The Fnc-2 System User's Guide and Reference Manual

Release 1.19 — August 1997

Martin Jourdan and Didier Parigot

INRIA
Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France

Telephone: [33] (1) 39 63 56 05
E-mail: {Martin.Jourdan,Didier.Parigot}@inria.fr
WWW:http://www-rocq.inria.fr/oscar/FNC-2/

Abstract : This volume provides the reader with complete information — that is, as complete as possible at the time of writing — about all the aspects of the use of the FNC-2 system, a powerful and versatile attribute grammar processing system. The *User's Guide* is an informal presentation of its practical use, mainly based on examples; it also contains much advice for programming with FNC-2 . The OLGA *Reference Manual* is a complete and precise specification of all — but only — the linguistic aspects of an attribute grammar specification to be input to FNC-2 . The FNC-2 *Reference Manual* is a complete and precise description of how to operate FNC-2 on your computer, and how the generated evaluators may be integrated in a more complete program such as a compiler. PPAT*: Pretty Printer for Attributed Trees* describes the PPAT language to decompile complex attributed trees providing a concrete representation of an FNC-2 -processed, abstract-syntax-based attributed system. *Generating Abstract Trees with the SYNTAX System* describes how to construct a tree for FNC-2 in parallel with the analysis of a source text by a parser generated by SYNTAX™. The *Appendices* include much relevant information such as a completely commented example, the summaries of the concrete syntax of the languages of FNC-2 , the bibliography, the index, etc.

Résumé : Ce volume fournit au lecteur une information complète — c'est-à-dire aussi complète que possible au jour de l'édition — sur tous les aspects du système FNC-2 , un système de traitement de grammaires attribuées puissant et souple d'emploi. Le *Guide de l'Utilisateur* est une présentation informelle de son utilisation pratique, basée principalement sur des exemples; il contient aussi beaucoup de conseils pour programmer avec FNC-2 . Le *Manuel de Référence d'*OLGA est une spécification complète et précise de tous les — mais seulement des — aspects linguistiques d'une spécification de grammaire attribuée destinée à FNC-2 . Le *Manuel de Référence de* FNC-2 est une description complète et précise de la façon de mettre en œuvre FNC-2 sur votre ordinateur, et de comment les évaluateurs engendrés peuvent être intégrés dans un programme plus complet comme un compilateur. PPAT*: imprimante intelligente pour les arbres attribués* décrit le langage PPAT pour la décompilation d'arbres complexes et pour la répresentation concrète d'un système basé sur d'arbres abstraits, traité par FNC-2 . *Engendrer des Arbres Abstraits à l'aide du Système SYNTAX* décrit comment construire un arbre pour FNC-2 en parallèle avec l'analyse d'un texte source par un analyseur syntaxique engendré par SYNTAX™. Les *Annexes* comportent beaucoup d'information associée, comme un exemple complètement commenté, le résumé de la syntaxe concrète des langages de FNC-2 , la bibliographie, l'index, etc.

Keywords : attribute grammar – attribute evaluator – language definition – semantics – applicative language – polymorphism – overloading – exception – semantic analysis – lexical analysis – – syntactic analysis – program tree – abstract syntax – incremental evaluation – structured editor – program editor – pretty printer

Mots-clés : grammaire attribuée – évaluateur d'attributs – définition de langage – sémantique – langage applicatif – polymorphisme – surcharge – exception – analyse sémantique – analyse lexicale – analyse syntaxique – arbre de programme – syntaxe abstraite – évaluation incrémentale – éditeur structuré – éditeur de programmes – imprimante intelligente

Contents

Foreword

This volume brings together all of the documents that are needed to understand and to use the FNC-2 attribute grammar processing system.

FNC-2 is a general system. It is used in a particular application by providing a description of the set of valid inputs and how they are to be processed. The inputs are statements in some formal language, and the processing involves extracting an understanding of what the statements "mean".

In traditional compiler terms, FNC-2 expects to be given the parse tree that is output by a parser, on which FNC-2 will perform semantic analysis to produce attribute values that are added to the tree. This (attributed, or decorated) tree will be used as input to the optimization and code generation phases of a compiler. FNC-2 can also be used in other non-compiler applications.

input statements

↓

syntactic and lexical analysis: SYNTAX

↓

parse tree

↓

semantic analysis: FNC-2

↓

attributed tree

↓

optimization

↓

attributed tree

↓

code generation

↓

translated statements

FNC-2 is designed to work with the output of a particular parsing system, named SYNTAX, which is part of the FNC-2 distribution and is also described here.

This volume is organized into several parts, as follows.

- Part I is a User's Guide. It describes how to obtain and install the software. It shows how to use the interface (XFNC2 tool or FNC2 shell-script) to all parts of the installed system by describing the example *simproc* (*sim*ple language with *proc*edures) that is part of the distribution. It also shows how to look at attributed trees that are generated by the system.

  This part includes a short tutorial on attribute grammars, the theory that underlies FNC-2 , and further references to the extensive literature on this theory for those readers who need more background.

- Part II is the reference manual for the OLGA language. OLGA is the notation that is used to describe attribute grammars for input to FNC-2 .

  This Part makes up about one half of this volume.

- Part III is the reference manual for the FNC-2 software system. The various components of the system and the algorithms they use are described in detail.

- Part IV describes PPAT, a program for printing the attributed trees that are generated by FNC-2 .

- Part V describes SYNTAX, a system for generating the parse trees that are processed by FNC-2 . SYNTAX uses descriptions of the lexical and syntactic aspects of a language. From these descriptions it produces tables that are combined with existing programs to produce lexical and syntactical analysers (scanners and parsers) that produce the parse trees required by FNC-2 .

- Part VI is a set of appendices containing technical specifications. These include detailed descriptions of *simproc*, the vocabulary for OLGA, the vocabulary for ATC (the notation used as input to SYNTAX), and the vocabulary for PPAT.

Credits

The global design of the FNC-2 system was done by Martin Jourdan and Didier Parigot. OLGA was designed by Martin Jourdan, after some preliminary work by himself and Joël Bonnet, with numerous comments and suggestions by Didier Parigot, Antoine Rizk, Bernard Lorho, Pierre Boullier, and many others. The basic algorithms used in the evaluator generator, the kernel of the FNC-2 system, were designed by Didier Parigot with the help of Martin Jourdan. The space optimization algorithms were designed by Catherine Julié, improving the work of Uwe Kastens. The implementation of FNC-2 is being carried out by Didier Parigot and Martin Jourdan. Catherine Ayrault devised the basics of the translation of OLGA into C. Pierre Boullier and Philippe Deschamp provided helpful support for the development of the lexical and syntactic parts of FNC-2 using the SYNTAX$^{\mathrm{TM}}$ system. Bernard Amilien implemented the graphical user interfaces.

Acknowledgements

Antoine Rizk, Bernard Lorho, Pierre Boullier, and the other members of the "Langages et Traducteurs" projects made valuable comments and suggestions about this volume. Special thanks go to Antoine Rizk and Olivier Durin for correcting our use of the English language. Tamás Gaál contributed to the first publicly available edition.

This work was supported in part by the Commission of the European Communities under ESPRIT Projects #956 "Components for Future Computing Systems" (COCOS) and #5399 "Compiler Generation for Parallel Architectures" (COMPARE).

Part I

User's Guide

Chapter 1

Installation

FNC-2 is an attribute grammar processing system available from INRIA (Institut National de Recherche en Informatique et en Automatique). FNC-2 includes

- *asx*: an attributed abstract syntax compiler,
- *fnc2*: the OLGA compiler (OLGA is the input language used by FNC-2 to describe attribute grammars),
- *atc*: an interface to the SYNTAX parser generator,
- *ppat*: a pretty-printer for attribute trees,

and a complete library. Building applications with FNC-2 requires the use of SYNTAX, which is a parser generator system from INRIA. SYNTAX includes

- *lecl*: the lexical constructor,
- *csynt*: the syntatic constructor, and
- *prio*: the disambiguation processor.

These two software systems are available in versions for Sun4 or Solaris systems via anonymous `ftp` from `ftp.inria.fr` (INRIA's general-purpose ftp server). The distribution is in the form of compressed `tar` files. You must retrieve the files and extract and install what is contained in them. You should first make a directory to hold the distribution files, and change into it.

```
% mkdir fnc2
```

```
% cd fnc2
```

Now establish an `ftp` connection to the INRIA server.

```
% ftp ftp.inria.fr
```

When you are asked for your "name", you should type:

```
anonymous
```

When you are asked for your "password", type your email address, for example:

```
jsmith@mach.univ.cy
```

Once you have successfully established a connection, you should give the following commands to FTP:

```
ftp> cd INRIA/Projects/oscar/FNC-2

ftp> binary

ftp> ls -l
```

You will see a list of the files available in the FNC-2 directory.

You are now ready to copy the FNC-2 distribution to your system. There exist two current distributions of the FNC-2 system :

- for SunOS system `fnc2SunOS.tar.gz` ;
- for Solaris system `fnc2Solaris.tar.gz`.

You can get it like this

```
ftp> get README

ftp> get fnc2SunOS.tar.gz

ftp> get f2manual.ps.gz
```

SYNTAX is in a subdirectory.

```
ftp> cd syntax

ftp> ls -l
```

There exist two current distributions of the SYNTAX system :

- for SunOS system `f2sxSunOS.tar.gz` ;
- for Solaris system `f2sxSolaris.tar.gz`.

You should verify the version of the SYNTAX distribution () from the directory listing before typing this command :

```
ftp> get f2sxSunOS.tar.gz

ftp> quit
```

You now must install FNC-2 on your machine.

```
% gunzip -c fnc2SunOS.tar.gz| tar xvf -
```

You now must install SYNTAX.

```
% gunzip -c f2sxSunOS.tar.gz | tar xvf -
```

Read the file `A_LIRE` for installation instructions. This version of SYNTAX is tailored for use with FNC-2 and is not the standard release. To use it essentially all that is required is to add the subdirectories created by the installation (`bin`, `incl`) to your appropriate paths, or to put the contents of those directories into your appropriate local directories.

FNC-2 is a system in active development and in use within INRIA and elsewhere. Accordingly, details of the system's behavior and the assumptions it makes about its environmental may change as it is used and developed. If you have any problems, send electronic mail to

Didier.Parigot@inria.fr `http ://www-rocq.inria.fr/~parigot`

concerning FNC-2 or to

Pierre.Boullier@inria.fr

concerning SYNTAX.

An alternative method of obtaining the software is to use the World Wide Web and a network browser. Go to the page

`http://www-rocq.inria.fr/oscar/FNC-2/`

using your network browser, and retrieve the files for FNC-2 and SYNTAX using the appropriate highlighted phrases. The installation steps that have already been described will still have to be carried out.

Chapter 2

The XFNC2 Interface to FNC-2 : An Example

2.1 What are *simproc* and XFNC2 ?

The distribution of FNC-2 contains a complete working example. This example is the programming language *simproc* (*sim*ple language with *proc*edures) that is defined in Appendix A. As a result of the installation of FNC-2 you will have a subdirectory named `simproc` in the subdirectory `examples` of FNC-2 where you did the FNC-2 installation. This subdirectory contains all of the files required to define *simproc*. This Chapter does not discuss *simproc* in detail (you can see that in the Appendix). Instead, the Chapter describes how to use the FNC-2 system. It assumes that you have *simproc* available to work with, and that you can experiment with the features described in this chapter using that example.

Developing an application using FNC-2 involves creating files containing specifications of the various aspects of the application: lexical structure of inputs, syntactic (grammatical) structure, attribute equations, and so on. These files are processed in various ways:

- first, to ensure that the specifications make sense, and
- second, to produce encodings of the specifications that can be used in an application such as a compiler.
- finaly, to choise the target language that the FNC-2 system can be generated; currently C, LISP, fSDL structure and ML are supported.

Although this concept is straightforward, managing the files and the required processing can be complicated in practice. XFNC2 [1] is an interactive interface that manages this process for you. It provides a number of high-level operations that can be selected with a mouse (and some text input), while hiding the details of file interdependencies and the required processing commands.

The example shows the use of FNC-2 and XFNC2 to build the front end of a traditional compiler, but other kinds of applications can also be specified using attribute grammars, and parts of those applications can be built using the FNC-2 tools. There exist a equivalent shell-script, call FNC2, of the XFNC2 X-windows tool. You need the GNU make utility and GCC compiler to use XFNC2 or FNC2 shell-script.

[1] Not yet upgraded to X11R6 and Solaris system

This Chapter does not attempt to explain all the details of XFNC2. Some of these (especially options that you can select) can only be understaood after you have become more familiar with the detailed operation of FNC-2 and its component parts as described later in this volume.

2.2 Operations in XFNC2

The chart in Figure 2.1 shows the general flow of a user's interaction with XFNC2. The important components of the chart, and the ideas they represent, are described here. These are subsequently described in more detail. Our purpose at this point is to give a general description of the system and the flow through it.

- A *specification* is a set of files containing the various kinds of specification needed by the system.

  There is usually a one-to-one correspondence between an application and a directory in your file system — that directory contains all and only the files for that application.

  The specification files are in the `spec` subdirectory.

  You can use any editor you like to create and modify the specification files.

  They are shown in a dashed box in the Figure, because they are not written by XFNC2, they are only read by it.

- An *application* is an FNC-2 application.

  Usually you will use XFNC2 to deal with a single application, such as the *simproc* language specification.

  After you have gained some familiarity with the system, you will be able to use XFNC2 to deal with several different applications in a single session if you want to do that ; the `Open` operation allows you to do this.

  For now, you can simplify the diagram in your mind by omitting the `Open` box and its associated arrows, and the `Close` box which allows you to stop processing some application.

- The `Quit` box simply indicates ending a session. This can be done when the specification of the application is completely written and has been fully processed, or at any time prior to that.

- The `Help` box provides some brief information about the system.

- The `Create` and `Init` operations are intended to be carried out once to establish the context in which an application will be developed.

- There remain two important repetitive loops in the Figure. These correspond to the two kinds of processing mentioned at the beginning of this Chapter.

  The first of these involves the `Construction` box. This represents a *construction* cycle that comprises checking the specifications – to ensure that each required specification is present, and that each satisfies the constraints of the particular processor for which it is designed (syntax, semantics, etc.).

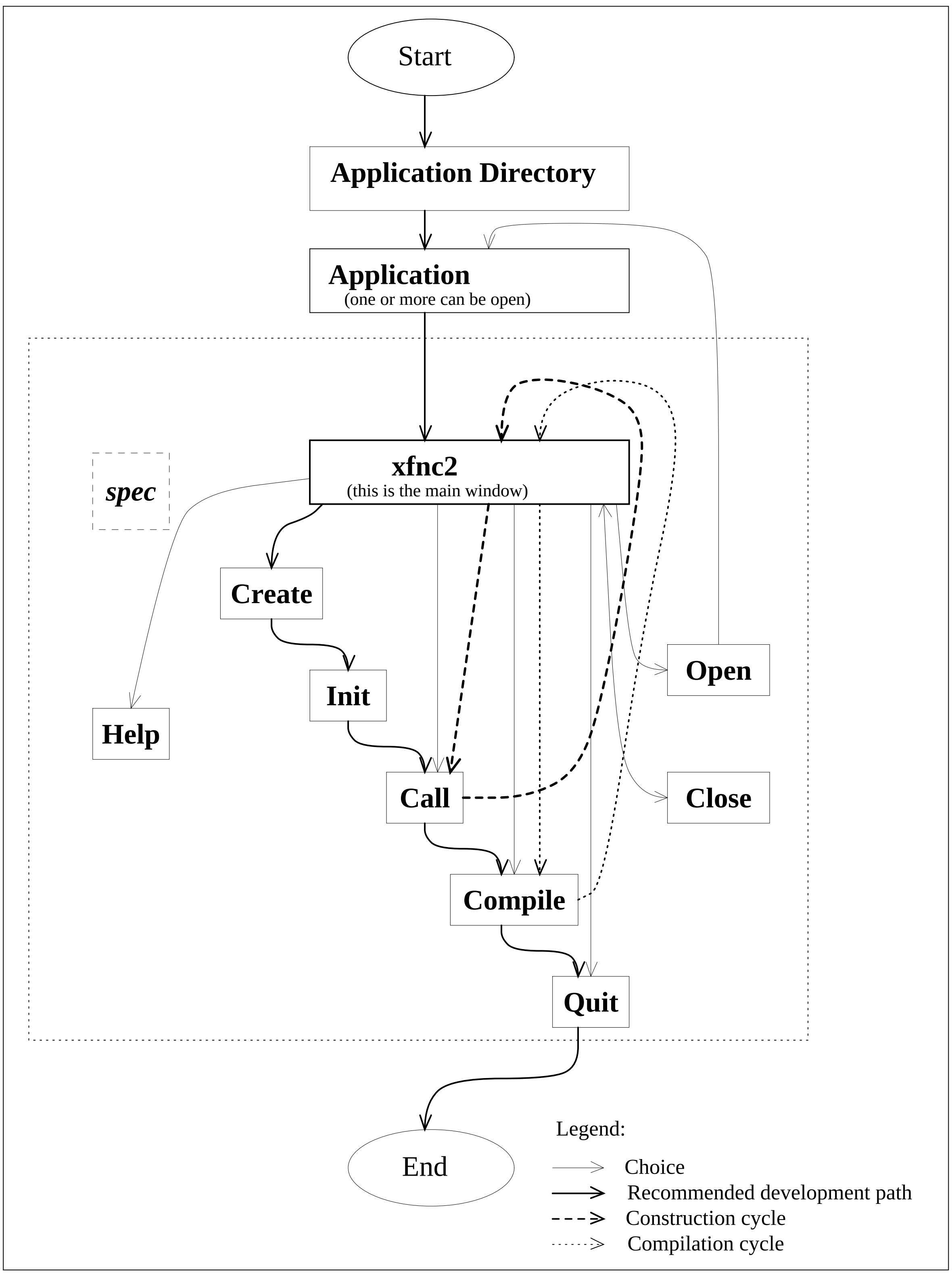


Figure 2.1: General flow–chart of an XFNC2 session

```
total 38
-rwx------  1 barnard     8057 Mar 19  1992 ag_simproc-check.olga*
-rwx------  1 barnard     6518 Dec  9  1991 ag_simproc-term.olga*
-rwx------  1 barnard     1138 Dec  9  1991 dec_simproc-types.olga*
-rwx------  1 barnard     1914 Dec  9  1991 def_simproc-types.olga*
-rwx------  1 barnard     2098 Dec  9  1991 simproc-adt.asx*
-rwx------  1 barnard     1674 Dec  9  1991 simproc-base.asx*
-rwx------  1 barnard      503 Dec  9  1991 simproc-in.asx*
-rwx------  1 barnard      717 Dec  9  1991 simproc-out.asx*
-rwx------  1 barnard     5788 Dec  9  1991 simproc-ppat.ppat*
-rwx------  1 barnard     1784 Dec  9  1991 simproc.atc*
-rwx------  1 barnard      603 Dec  9  1991 simproc.lecl*
-rwx------  1 barnard      114 Dec  9  1991 simproc.prio*
-rwx------  1 barnard     1542 Dec  9  1991 simproc.recor*
```

Figure 2.2: Contents of the `simproc/spec` Directory

The loop represents the process of editing the specifications, then checking them, editing again, checking again, and so on.

The direct manipulation of the specification files that is required on each traversal of this loop is not part of XFNC2 but is done externally using an editor.

- The second loop involves the `Compile` box. This indicates actual *compilation* of the engine that includes the encoded specifications and the processing software. For *simproc*, for example, this will make the front end of a compiler.

  Errors detected in the compilation cycle may require a return to the construction cycle to modify the specification.

2.3 Starting an XFNC2 Session

When XFNC2 begins, it wants to find a `spec` directory containing a (partial) set of specifications for an application. It is not necessary that the set of specifications be sound and consistent — XFNC2 will be used to manage getting to that state — but the directory should exist.

There is a complete set of specifications for *simproc* in the `spec` subdirectory of the `simproc` subdirectory of your installation directory. Go to the `simproc` subdirectory now to run XFNC2. If you list the contents of the distributed `spec` directory you will see something like what appears in Figure 2.2 (of course, your userid — and in other Figures in this Chapter, your pathname — will differ from what is shown). The programs that XFNC2 calls will produce derived files in `src_fnc2`. These files can be used by XFNC2 or by FNC-2 directly if you decide later not to use the interactive interface.

Now run XFNC2. When it begins, it will display the window shown in Figure 2.3. The directory name shown will be the name of the directory where XFNC2 was started.[2] If this is

[2]The situation is actually slightly more complicated than this. There are two environment variables that

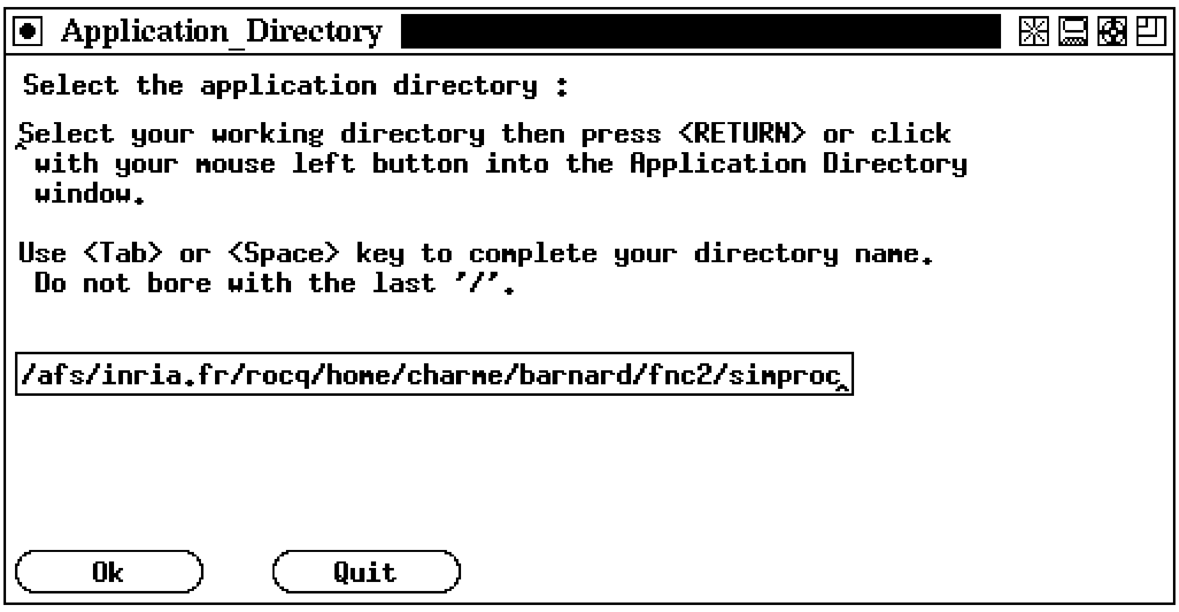


Figure 2.3: Application Directory Window

not the directory containing your `spec` directory, you can specify the correct directory here. For the example, if you started in the `simproc` directory, select "Ok" with the mouse.

The next window you will see is shown in Figure 2.4. The default application and language names are the same as the name of the directory where you started (or that you specified in the previous window if you changed it). For the example application, just select "Ok" with your mouse.

In other circumstances (once you know more about the system, and are not just working through the distributed example) you can change these names, but you will have to change the specification file names if you do so. The application name controls internal name generation (binary names) and the language name determines the file name prefixes used (such as `<languageName>.atc`). Once you leave this window these cannot be changed until you start a new session or open a new folder.

2.4 Developing an Application

You should now see the window shown in Figure 2.5. This is the main XFNC2 window. From here you can get help, carry out the initialization operations, and manage the construction and compilation cycles. You can scroll in this window using the scroll bar, and you can type text into it if you want to mark points in your script to be searched for later.

We will show this window in several different states as the operations of XFNC2 are applied to the example, so that you can see if your installed version is behaving as it should.

Selecting the `Help` button causes display of the window in Figure 2.6. It gives a brief description of the operations that can be invoked from the main window. Select the `Help`

---

can be optionally used. `APPLICATION` specifies the default working directory ; if it is unspecified the current directory (where XFNC2 was started) is used. The variable `XFNC2RESOURCES` may contain the full pathname (including the filename itself) of a window resource file. You can safely ignore this for now.

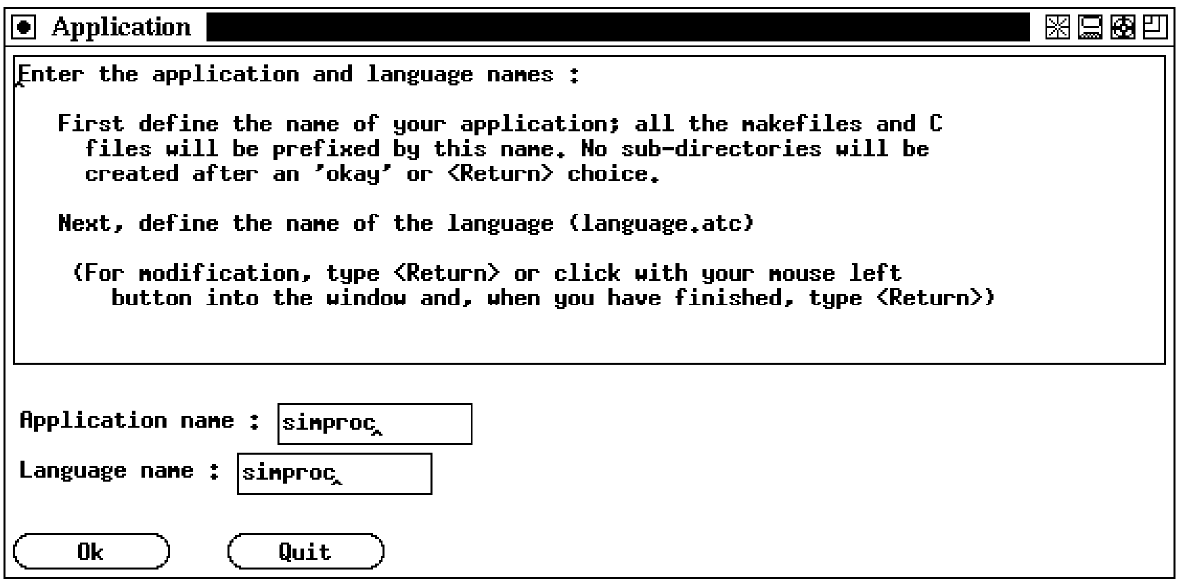


Figure 2.4: Application Window

button with your mouse to get the window, and dismiss it by selecting the "Ok" strip at the bottom of the window.

2.4.1 Choising the target language

FNC-2 can generate engines for different target languages ; currently C, LISP, fSDL structure and ML are supported. By the `Option` box, it is possible to choise the target language (see the Figure 2.16) with the `target language` menu.

2.4.2 Establishing the Development Context

To get ready for the development it is necessary to select the `Create` button. This results in the window shown in Figure 2.7 being displayed. For the example, select "Ok". The main window will now change, indicating the results of the operation. Some directories are created, and you are shown a list of them.[3] You should see output that looks like Figure 2.8.

Next select the `Init` button. This operation produces the file `simproc.MKFNC2` and displays it in the main window. You should see output that looks like that in Figure 2.9.

2.4.3 The Construction Cycle

Now the system is ready for you to develop a complete and consistent set of specifications for your application.

To continue with the *simproc* example, select the `Construction` button. This results in the window in Figure 2.10 being displayed. You have a chance to set some options here, but for the example simply select "Ok". This operation takes some time and produces a

[3] The directories created here are : `bin` (binary files), `f2aux`, `incl` (C or Lisp include files), `incl_fnc2`, `lib` (library files), `tmp` (temporary files), `src` (C or Lisp source files), `src_fnc2` (fnc2 source files).

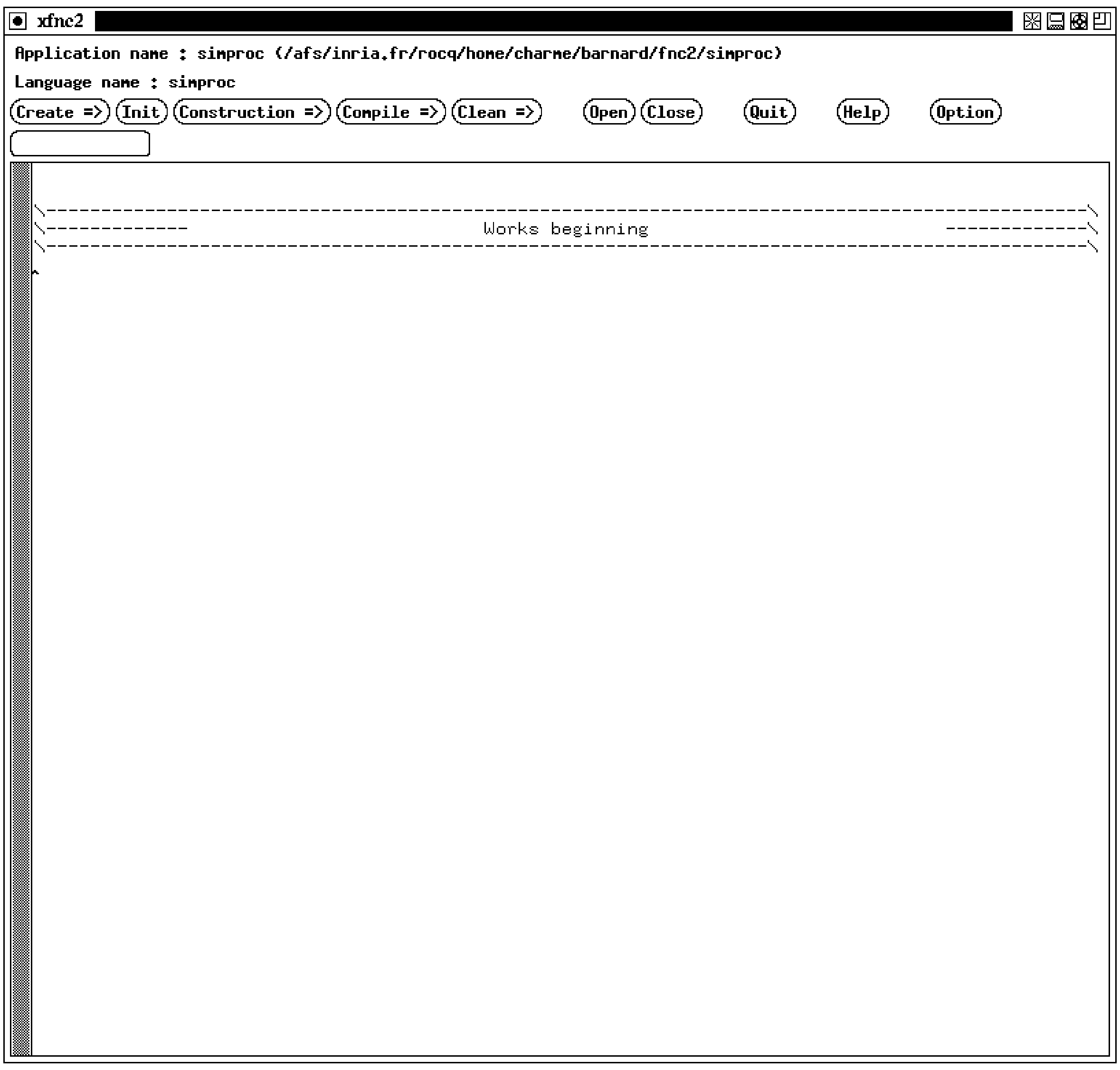


Figure 2.5: Main XFNC2 Window

considerable amount of output in the main window. When it is complete, you should see something like what is shown in Figure 2.11.

If you were starting with a new application you would use the construction cycle repeatedly at this point, refining and checking your specifications.

2.4.4 The Compilation Cycle

Once you have a set of specifications, you can use them to produce an engine that will process your application. This is done using the `Compile` button. When you select it, you will see the window shown in Figure 2.12. When you have learned more about FNC-2 you can use this window to set some options.

For the *simproc* example, again simply select "Ok".

Like construction, the compile operation also takes some time and produces a considerable

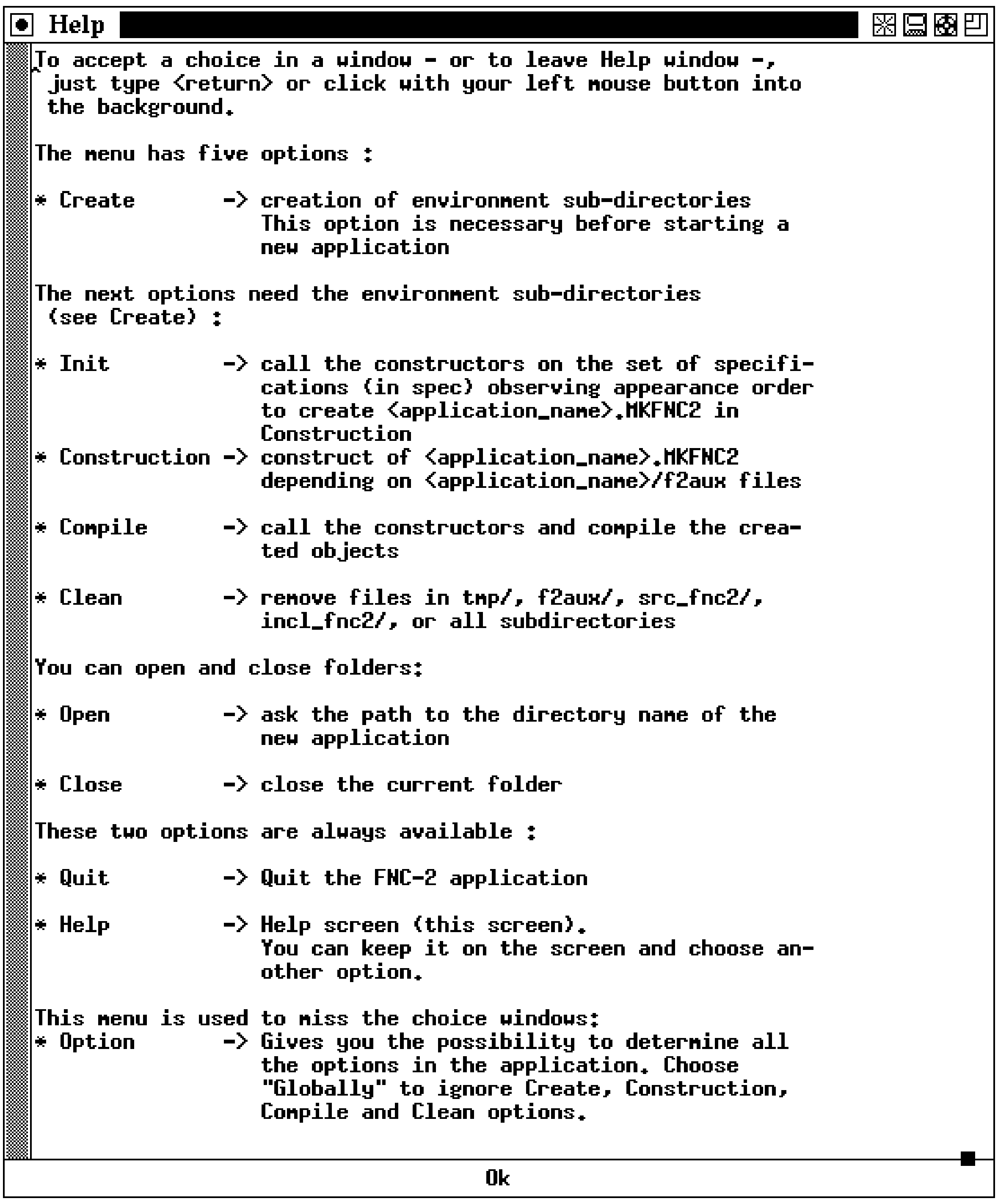


Figure 2.6: Help Window

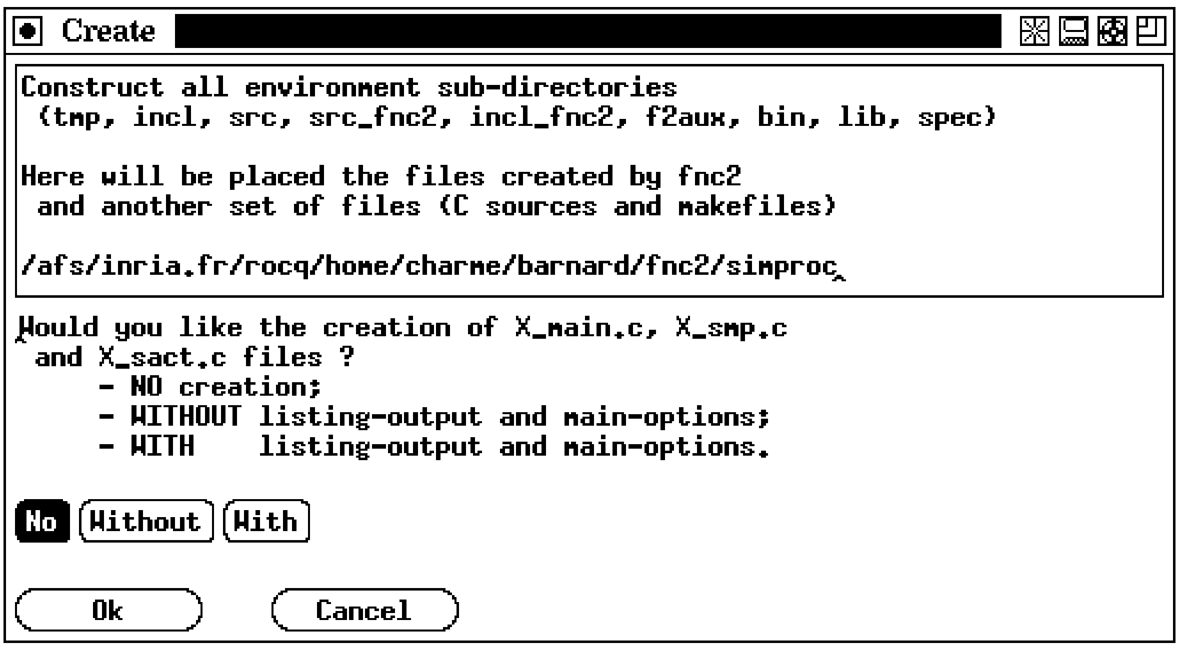


Figure 2.7: Create Window

amount of output in the main window. When it is complete, you should see something like what is shown in Figure 2.13.

If you were working with a new application you would use the compilation cycle repeatedly, perhaps also returning to the construction cycle, until you had an error-free compilation of the engines using a complete and consistent set of specifications.

When an engine is produced, it is in the `bin` directory with the name of your application. For the example, this means you will find the generated engine in `simproc/bin/simproc`. You can execute it using a distrobuted example program, by typing this command (in a shell window, not in XFNC2):

```
% bin/simproc examples/fact.1
```

You should see output like that in Figure 2.14.

When you are developing your own applications, at this point you could develop a set of test inputs (programs) for your generated engine.

FNC-2 can generate engines for different target languages; currently C and LISP are supported. The *simproc* example is based on the C language. As you can verify by looking at the generated files, FNC-2 provides a framework (primarily in the form of C skeleton programs) to perform some straightforward tasks. The user must provide a similar encapsulation of the generated FNC-2 objects in more complicated cases. In simple cases like *simproc*, the standard framework will suffice.

2.4.5 Other Operations

The remaining operations in the main window are these.

- `Clean` allows you to delete files that have been generated by XFNC2. This is useful if you get into some inconsistent or confusing state. If you remove generated files by mistake,

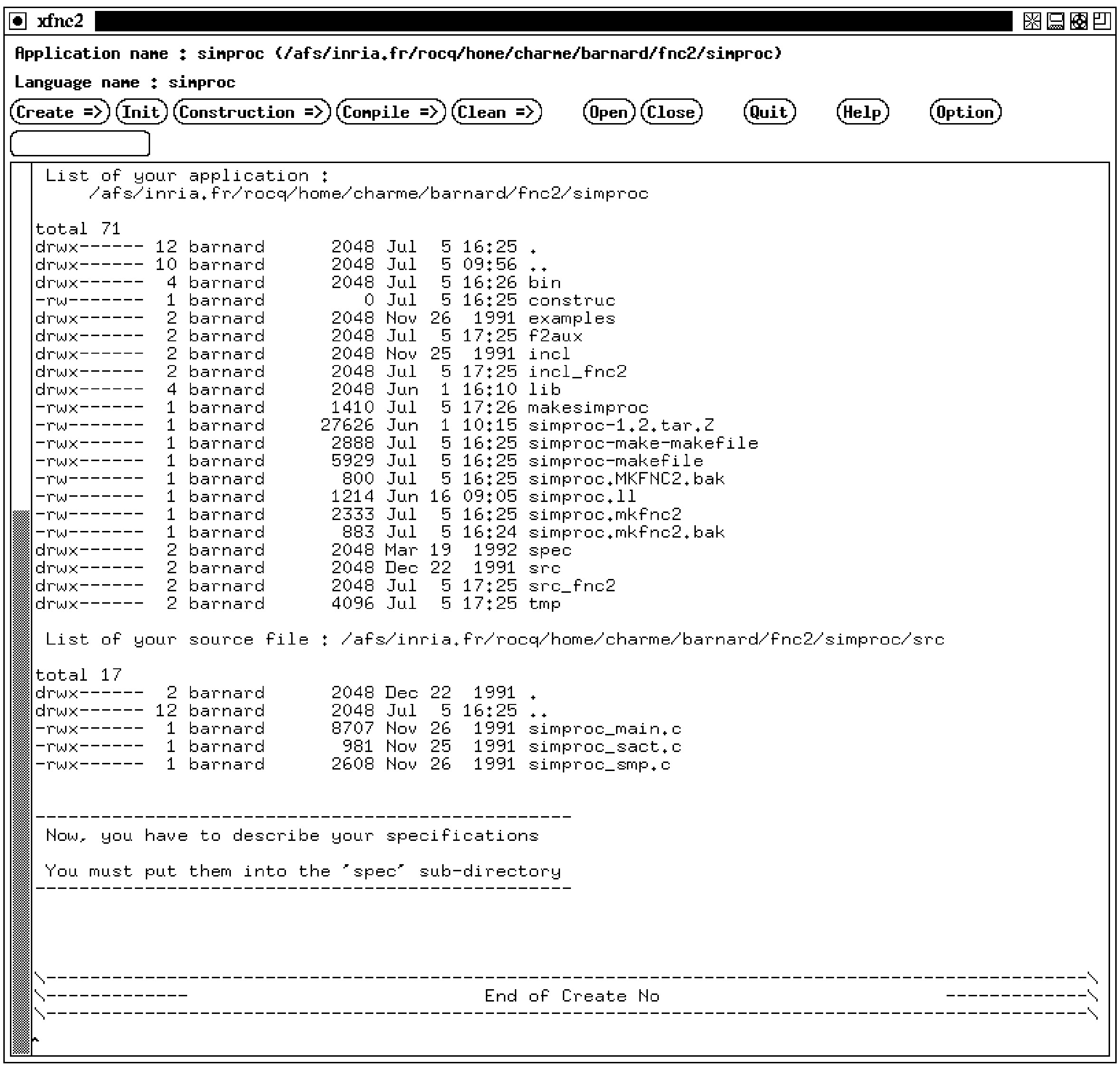


Figure 2.8: Results of Create

they can easily be reproduced (at the cost of a little time) using the `Construction` or `Compile` operations. These two facts taken together mean that you can experiement with confidence, since the files can be easily restored to a previous state. When you select `Clean` you will see the window shown in Figure 2.15.

- `Option` is for setting the default options that are displayed in the various windows used by XFNC2. When it is selected the window shown in Figure 2.16 is displayed.

- `Open` and `Close` allow switching among applications in a single XFNC2 session. Selecting `Open` displays the Application Directory Window (see Figure 2.3).

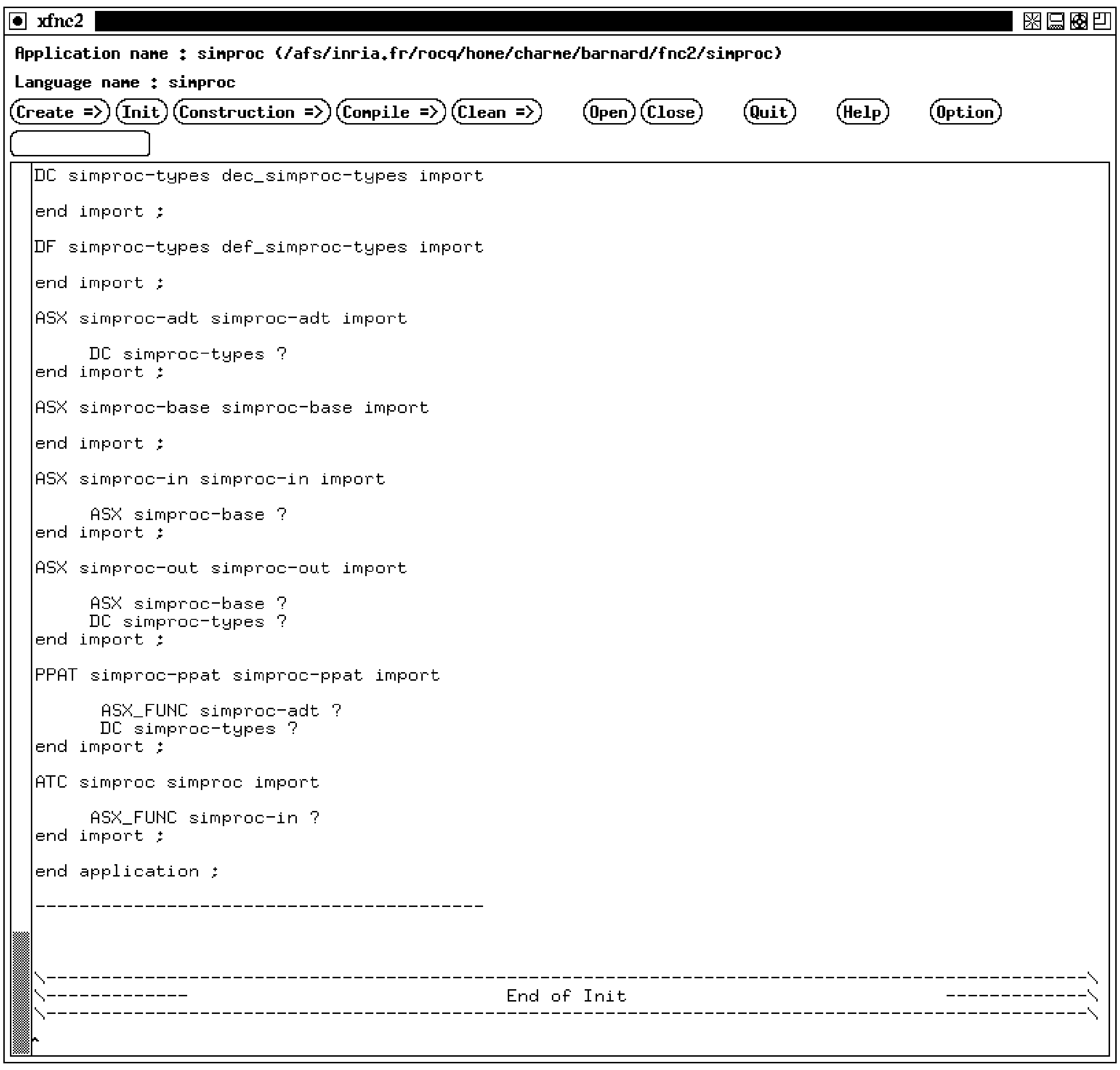


Figure 2.9: Results of Init

2.5 Recovering from Crashes

This application uses shared memory. In case of an unexpected error halting the program (`Bus error`) or destroying the main XFNC2 window, some manual cleaning up might be needed. If this happens to you, try the following procedure.

First, type

```
% ipcs -m
```

If a list (of at least 20 items) appears, continue with the following instructions. If you do not see such a list, you have some problem other than the one we are trying to solve here. Consult a local guru or send electronic mail to one of the authors of this volume.

If there remains allocated memory, type the following:

```
% ipcs -m | awk '{print "ipcrm -m ", $2}' | sh
```

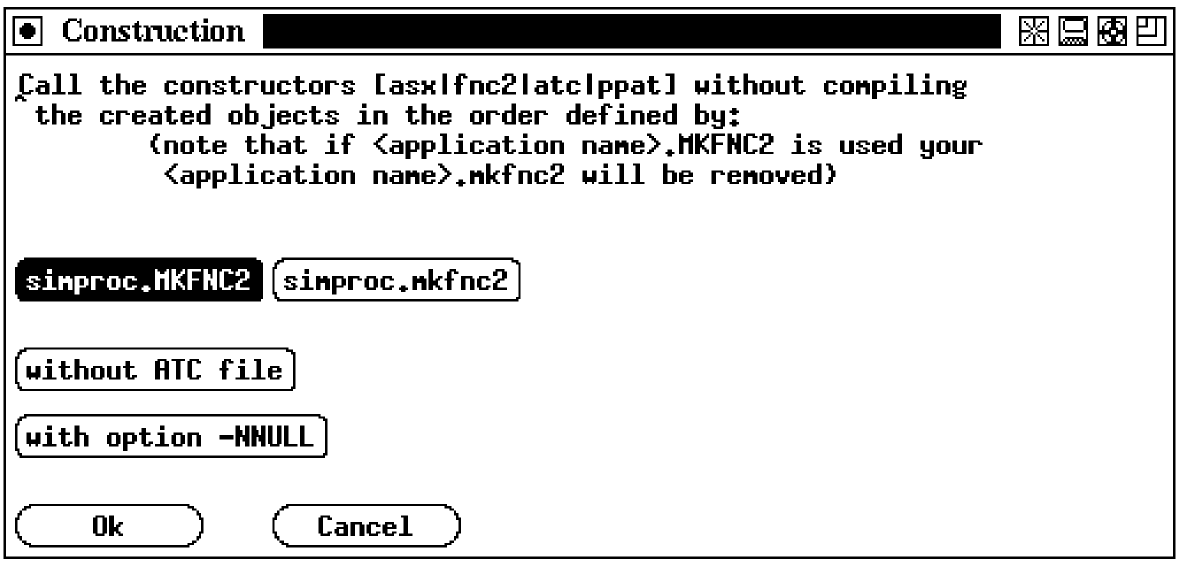


Figure 2.10: Construction Window

This is not elegant. The shell should not interpret the first 3 lines; if for some reason it does, this error message will occur three times:

```
'ipcrm:  shmid(0):  not found'
```

There may also be some forked shells still in existence. Type

```
% ps -aux
```

to see what processes you have and if there are any that look like they remain from your terminated session, type

```
% kill <process or shell number>
```

to get rid of them.

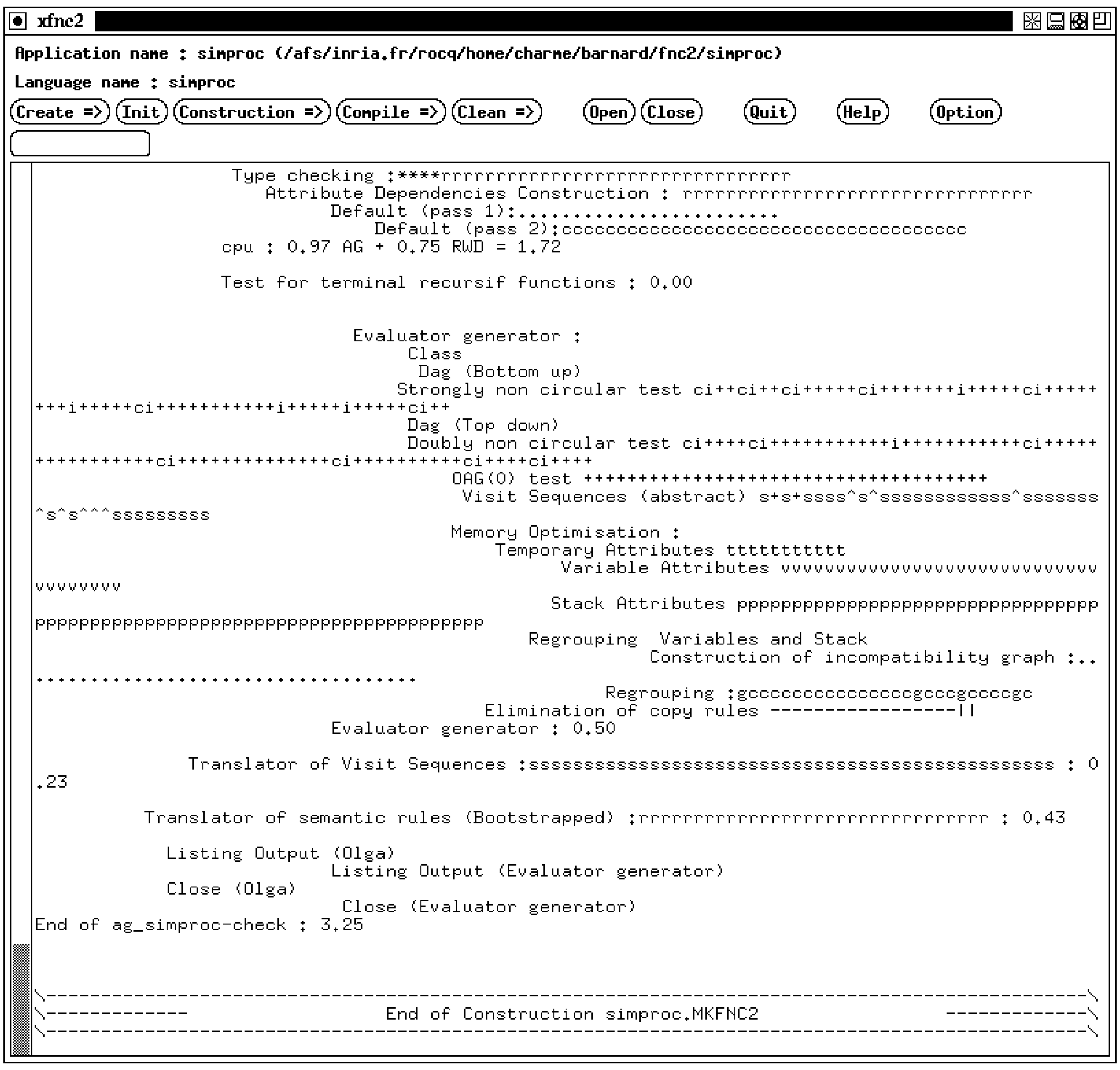


Figure 2.11: Results of Construction

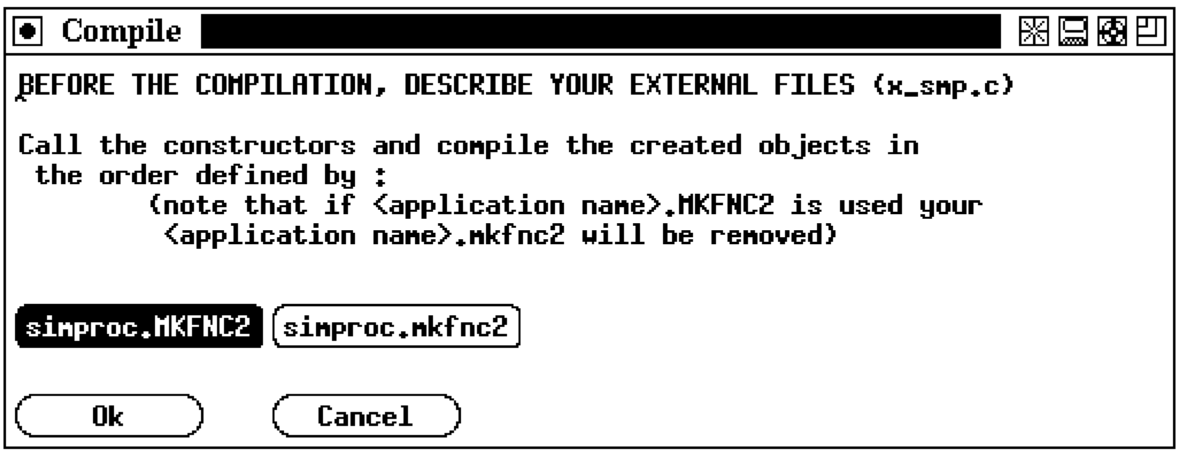


Figure 2.12: Compile Window

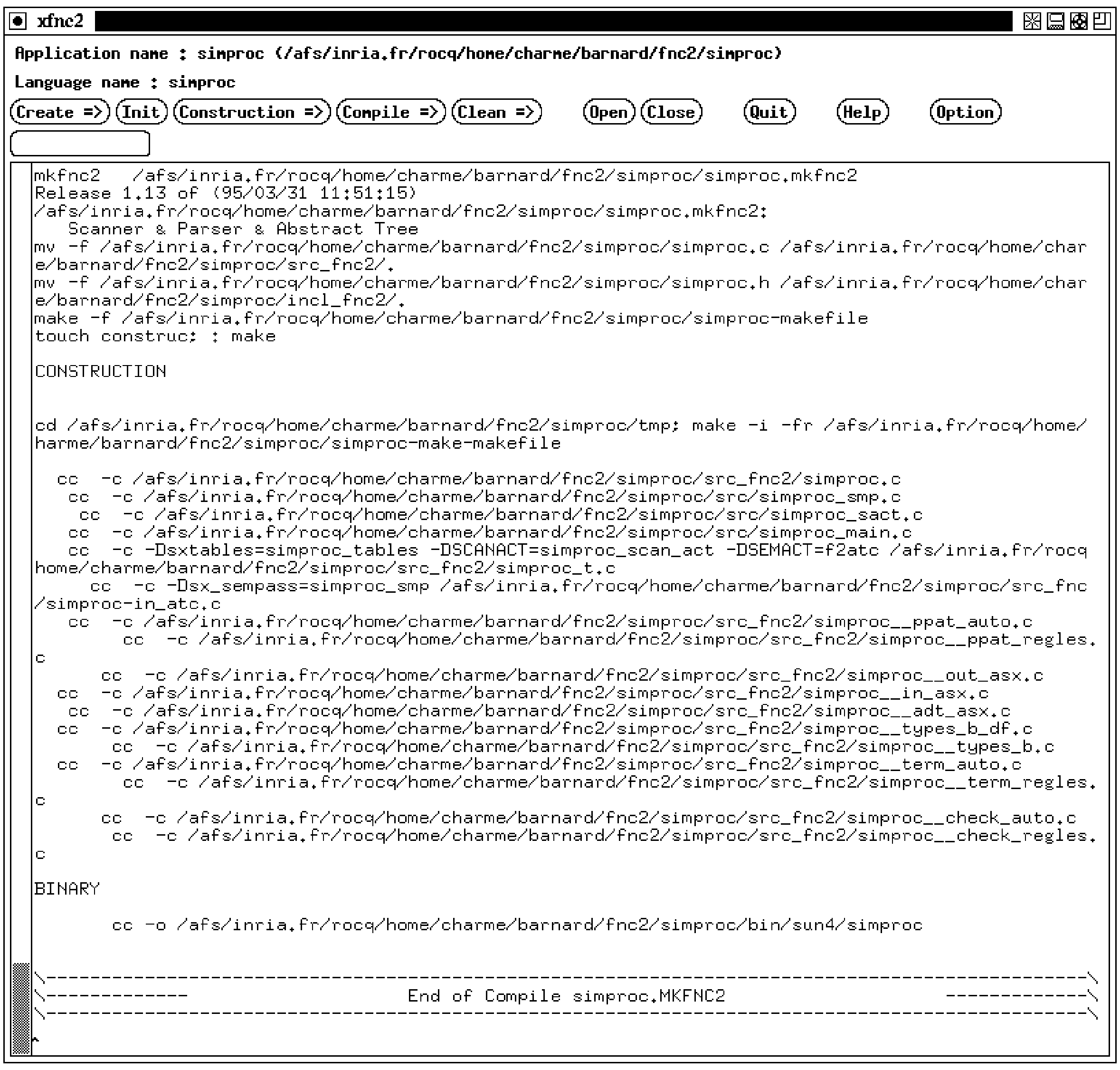


Figure 2.13: Results of Compile

```
version 0.0
examples/fact.1:
   Scanner & Parser
Execute
Prog
    Decl
    Intdecl J Prog
    Procdecl factorial I F Prog
        Decl
        Intdecl TEMP factorial
        Stmt
        Condit Designates I 0
        Then Assignint Refersto Refdesignates F 1
        Else Call factorial Difference Designates I 1 Designates
                 TEMP factorial
             Assignint Refersto Refdesignates F Times Designates
                 TEMP Designates I
    EndProddecl
    Stmt
    Call factorial 5 Designates J Prog
```

Figure 2.14: Constructed Engine Output from `examples/fact.1`

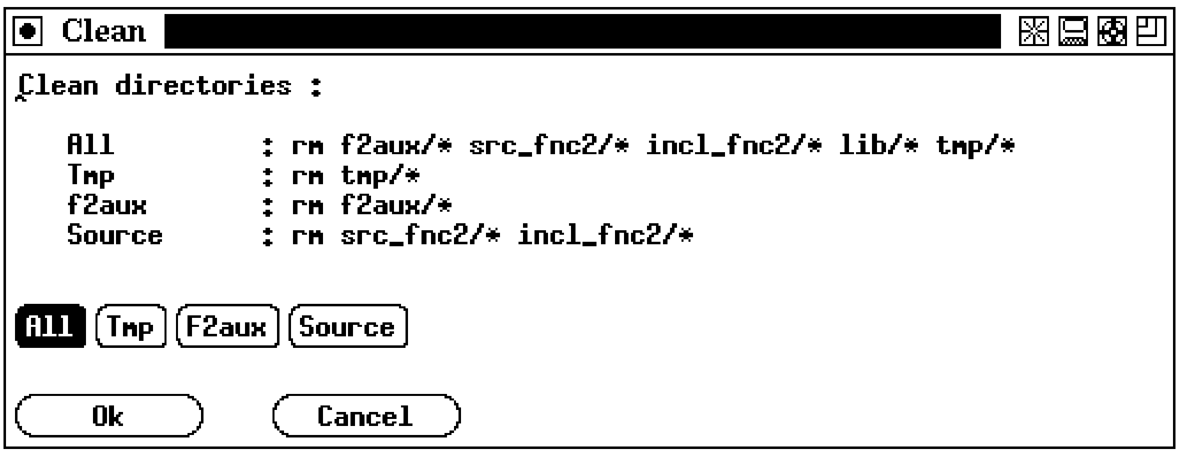


Figure 2.15: Clean Window

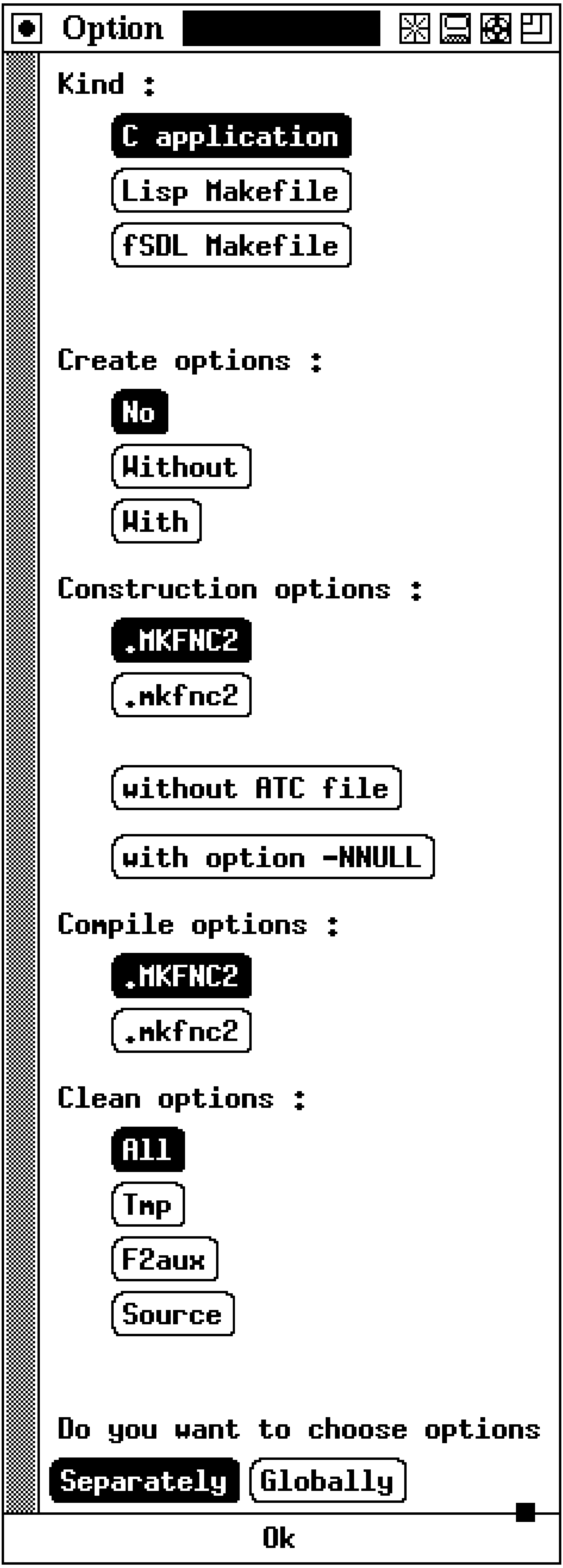


Figure 2.16: Option Window

Chapter 3

Visualization of SNC and non SNC grammars: XVISU

Introduction

XVISU is a practical tool to visualize dependency graphs of both SNC and non SNC OLGA grammars. Graph visualization is often delicate because of possible size and circularity constraints.

In the case of XVISU, the dependency graph is visited production by production, interactively. Each production, displayed by XVISU, consists of a parent node and its offsprings; the dependency graph can be walked upon either down– or upwards.

The FNC-2 component needed to this XVISU–based dependency-graph visualization is called EVALGEN (see section 18.1). XVISU and EVALGEN need synchronization: the former visualizes only what the latter produces.

3.1 XVISU example

To run an example, an application is needed in an FNC-2 environment including access in the PATH environment variable to the *xvisu* program and a properly set XVISURESOURCES environment variable. The specification of the respective OLGA grammar must have been processed without error (see the OLGA option **-agwrite** in section 18.1) and the result of the (OLGA) compilation must be in the respective intermediate file (created by the OLGA compiler) called `<grammar-name>.agst`. It is the *evalgen* program that performs circularity checks and *xvisu* cooperates with it. Though *xvisu* is independent of *evalgen*, the two programs should be in phase: the former can not receive data from the latter if not ready either on the synchronization or information type levels. Calling *evalgen* by either –visu or –circ option provides inter-task synchronization implicitly: the process server, *xvisu* is activated first then its client process, *evalgen*. Synchronization problems are indicated by the message:
"xxx.c: Connect() failed"
In such a case, the PATH and XVISURESOURCES environment variables should be verified.

The command

```
tutorial% evalgen -circ simproc-check
```

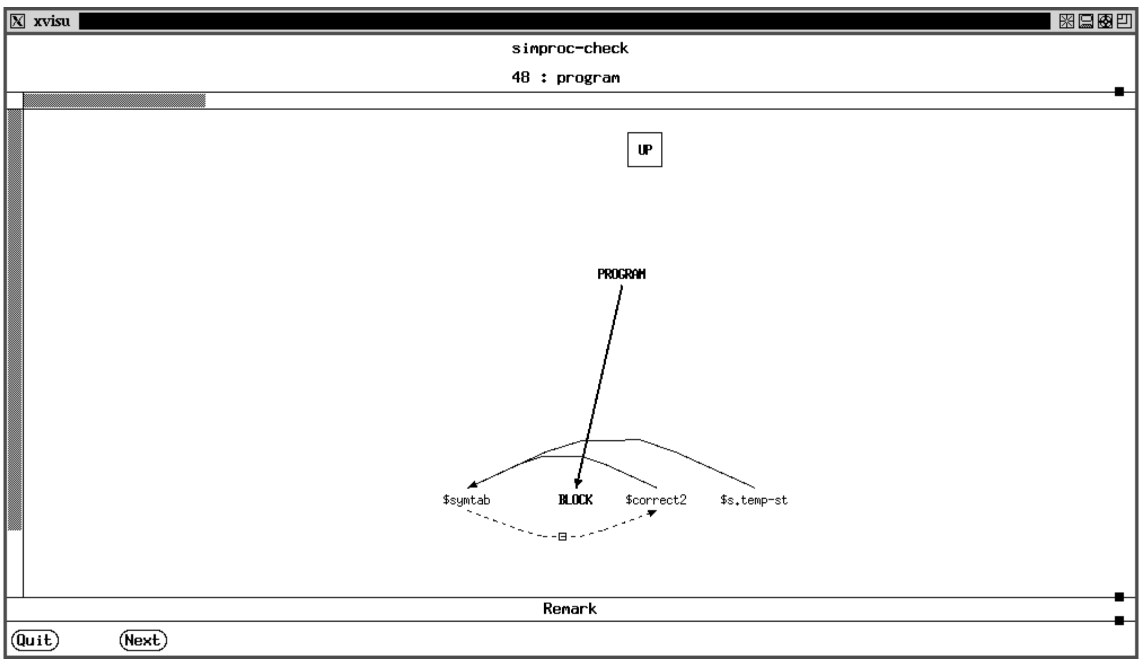


Figure 3.1: XVISU window: a production of the *simproc-check* (non-SNC) grammar

calls EVALGEN (now with the -circ option, see section 18.4) for an attributed grammar called `simproc-check` (the corresponding file is `simproc-check.olga`)[1].

On the standard output, the messages of EVALGEN are appearing (see section 18.1) and a new window pops up. The visualization window arriving (Fig. 3.1), four parts can be identified:

- The upper box contains several pieces of information about the studied grammar: its name, the relevant line number of the OLGA specification file involved, and the production name, too;
- The partial dependency graph of the production examined;
- Remarks about the graph;
- Commands.

If an offspring node is selected by the user, then it becomes a parent node, and its respective offsprings will be shown with all their dependencies. If the parent node is chosen next, the parent of the actual parent becomes the new parent node with all its offsprings (including the former parent among them) and the dependency relations. Tree grammars are allowed only so tree-walking is limited by both the root node and terminal nodes of the grammar.

3.1.1 Partial dependency graph

There is always a parent phylum and one or more offsprings can be attached to it. At the beginning, the graph displayed is the root production of the visited tree: the only possible

[1]Please note that this grammar is not identical to that of the *Simproc* example in appendix A. After some practice in attribute grammar manipulation, it is easy to se, why not!

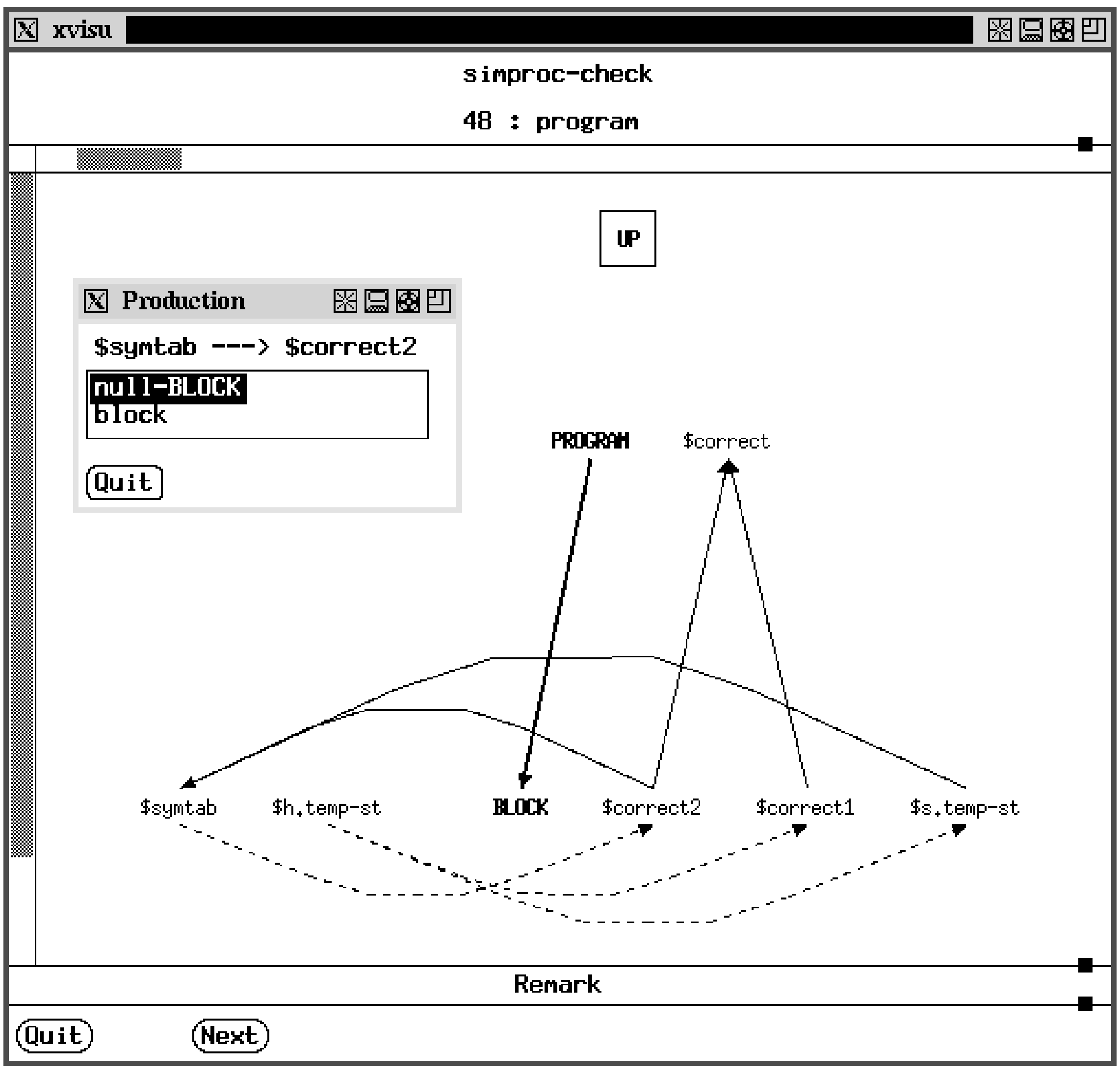


Figure 3.2: XVISU window: descending on an attributed tree

move is descending (if there is at least one offspring) by clicking on an induced dependency (see also chapter 4).

Phyla and their links are in **bold** (Fig. 3.2); attributes and their links are in normal width; induced dependencies are displayed by dotted lines. The **UP** sub-window means ascent: one-level ascent requires one click in this sub-window and further **UP**-s backtrack the path (as far as the initial phylum).

By using the mouse (i.e. clicking), additional information can be asked about a phylum, an attribute, or an induced dependency (in this case, the respective dotted arc should be clicked into) — but not about other links. To make this additional information window disappear, it is sufficient to click into it again.

If an induced dependency is clicked on, the productions of the phylum involved are displayed. To "descend" a level in the tree, it is sufficient to click on one of the productions displayed in a sub-window (Fig. 3.2). If this sub-window is not interesting any more, its own

"Quit" button should be clicked upon (and not that of the main window!).

3.1.2 Commands

- "Quit": The *xvisu* program can be exited without stopping *evalgen* — this latter continues execution;
- "Next": The visit of a particular tree can be finished without leaving *xvisu*. It serves to visit several trees of a grammar — if they exist. The direction is predefined by *evalgen*, that is, the visit is unidirectional.

Important note

As it should be clear by now, *xvisu* only displays tree representations (irrespective of their types), that is, nodes, attributes, and their relationships. In particular, it does not resolve any circularity: it helps the user only in visualizing a production of a dependency graph. All data used by *xvisu* (names, types of relations, etc.) are external, provided by *evalgen*; *xvisu* can not (and will not) modify them.

3.2 Xvisu implementation

Xvisu makes possible to visit a dependency graph without representing the whole graph. A production of the graph is represented and then it can be navigated down– or upwards.

3.2.1 Xvisu, the visualization program — technical details.

The production is represented by a two-level tree (a parent and its offsprings). The visit follows the user's intentions, that is, the immediate neighbouring "upper" or "lower" production can be displayed. Note that neither the root node can be bypassed ("upwards") nor terminal nodes ("downwards").

User communication to Xvisu is governed by clicks on the left mouse button. The program is based on the X11R4 graphical system, and is written in K&R C and SUN C++.

As mentioned before, the PATH and the XVISURESOURCES environment variables should be properly set; the former specifies the access path to *xvisu* to synchronize with *evalgen*, the latter describes the path to the *Xvisu* file describing graphical resources.

Sockets

In program communication, *xvisu* and *evalgen* use sockets: this sort of communication makes good type checking of interchanged information possible. BSD IPC is a collection of primitives to write interprocess communication. Both *xvisu* and *evalgen* are emitters and receivers (full duplex mode). The protocol UNIX DOMAIN (AF_UNIX) has been used; one communication path per file.

Chapter 4

A Tutorial Introduction to Attribute Grammars

4.1 Introduction

Attribute grammars were originally introduced by Knuth [Knu68] as a formal device to specify the semantics of programming languages. More precisely, an AG specifies a (partial) mapping—i.e. a translation—from the language described by a context-free grammar to some domain(s). AGs as a formal device have been studied by many authors, the most prolific of them being Engelfriet.

An important advantage of AGs over other formal methods for describing the semantics of a language is that it is an *executable* method, which means that it is generally possible to automatically construct a program which will implement the translation specified by a given AG. Finding efficient implementations of AGs was the topic of many many works, which were concretized into a number of systems to construct those programs, among which FNC-2 , the topic of this whole manual.

Lastly, it was recognized that AGs could actually be used not only as a formal device, but also as a very effective programming tool to describe and implement any syntax-directed computation, i.e. any computation on trees.

The purpose of this chapter is to introduce the reader to the use of AGs and to the specific terminology of this field of computer science. It will start with a simple example which will be fully commented in a step-by-step manner, just to give the flavor of the method, and then proceed with a brief review of the theory of AGs and of some methods to implement it. Unless explicitly stated, all what will be said is applicable to the general theory of AGs, and not only to its application in the FNC-2 system.

4.2 An example: binary numbers

This example appeared initially in the seminal paper on AGs by Knuth [Knu68], and has now been adopted as the standard example by most authors. It is indeed a very short example but exhibits most of the basic notions; it is hence quite pedagogical.

The purpose of this AG is to compute the decimal value of rational numbers written in fixed-point binary notation. It is so simple a problem that no one would ever think of writing an AG to solve it in practice, but an AG is a good specification for any implementation.

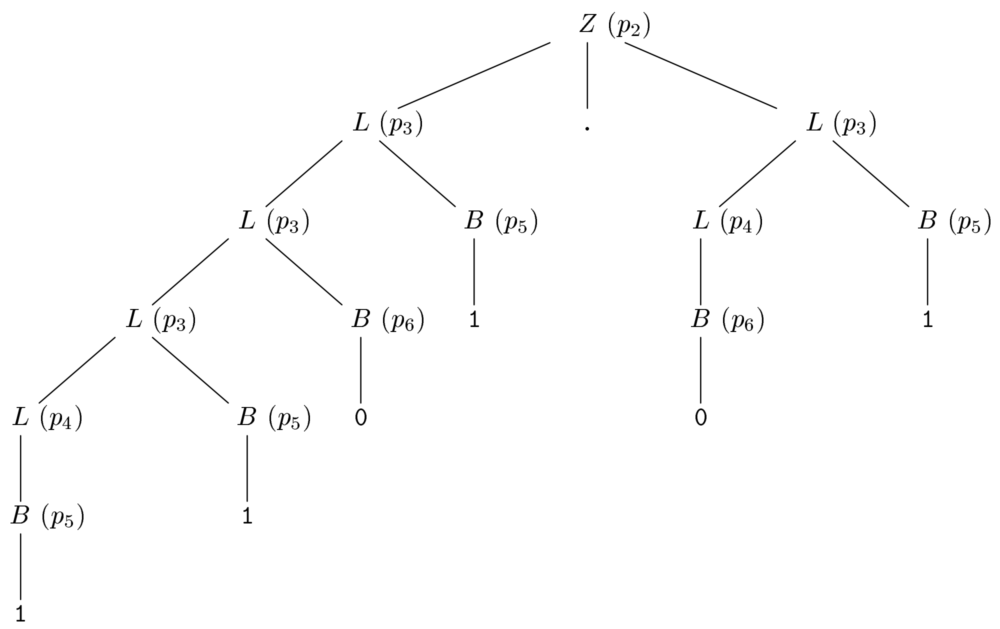


Figure 4.1: The parse tree of the string `1101.01`

Since AGs can be considered as an extension of context-free grammars, we first have to write down a grammar describing fixed-point binary numbers. The terminals of this grammar will be the bits "`0`" and "`1`" and the dot "`.`". The non-terminals are B, which represents a single bit, L, which represents a list of consecutive bits (a number), and Z, which is the start symbol and thus represents a whole number. The self-explanatory productions are as follows:

$$
\begin{array}{ll}
p_1\colon & Z ::= L \\
p_2\colon & Z ::= L\ .\ L \\
p_3\colon & L ::= L\ B \\
p_4\colon & L ::= B \\
p_5\colon & B ::= \mathtt{1} \\
p_6\colon & B ::= \mathtt{0}
\end{array}
$$

The reader will easily convince himself that the language generated by this grammar is indeed what we want, i.e. the set of fixed-point binary numbers. An example of such a number is the string `1101.01` which represents the decimal number 13.25 and whose parse tree according to our grammar is depicted in Fig. 4.1. Nodes of this tree are labelled either with terminals or with non-terminals; in the latter case, the production which applied at the node is written between parentheses close to the symbol name.

4.3 Attributes

An AG builds upon a context-free grammar by associating with each of its non-terminals a set of *attributes*, which can be considered as variables or slots or fields of these non-terminals. In the same way that non-terminals represent syntactic notions (structures) of the language, attributes represent semantic information associated with these notions. The information stored in the attributes can be of any type suitable for the problem at hand. In an AG describing a compiler for instance, we'll have attributes holding symbol tables, pieces of generated code, types of expressions, values of constant expressions, boolean flags specifying whether we require the value or the address of a variables, etc.

In our AG for binary numbers we have only attributes of simple types because the problem is simple. We first have an attribute called v (for value) representing the decimal value of the substring generated by each non-terminal; it is of type real and will be attached to the non-terminals Z, L and B. To represent e.g. attribute v of non-terminal Z we will write $Z.v$, to emphasize that an attribute is a "constituent" of the syntactic-semantic entity represented by a non-terminal. It is also an implementation hint: attributes can be implemented as fields of the record structure describing a node of the parse tree. This point will be further detailed below.

Note that, although they have the same name, attributes $Z.v$, $L.v$ and $B.v$ are formally distinct and bear no particular relationship, except that they correspond to the same semantic notion, i.e. the value of a substring; this is what is meant by giving them the same name. Also note that, as for non-terminals in a grammar, the names of the attributes are immaterial and can be chosen at will to enhance the readability of the specification.

4.4 Semantic rules

In addition to attributes attached to the non-terminals, the other addition of an AG to the underlying grammar is a set of *semantic rules*, attached to the productions. These rules are *equations* which specify how the values of all attribute occurrences appearing in a production are related to each other. For instance, since we have stated that attribute v of each non-terminal represents the exact value of the substring derived from it, we will have the following semantic rules:

$$\begin{array}{llll} p_1: & Z ::= L & p_3: & L_0 ::= L_1\ B \\ & \quad Z.v = L.v & & \quad L_0.v = L_1.v + B.v \\[1ex] p_2: & Z ::= L_1\ .\ L_2 & p_4: & L ::= B \\ & \quad Z.v = L_1.v + L_2.v & & \quad L.v = B.v \end{array}$$

In productions p_2 and p_3, we use subscripts to distinguish the various occurrences of non-terminal L. The semantic rules of production p_1 and p_4 are called *copy rules.*

The value represented by a "0" digit is always zero, so we can also write:

$$\begin{array}{ll} p_6: & B ::= \mathtt{0} \\ & \quad B.v = 0 \end{array}$$

but the value represented by a "1" digit depends on the position of this digit w.r.t. the decimal point. Since this latter information cannot be computed locally in productions p_5 and p_6, we must invent a second attribute whose role will be to transmit it to the nodes labelled by B. Let us call this attribute s for scale. It will have type integer. We can now write:

$$\begin{array}{ll} p_5: & B ::= \mathtt{1} \\ & \quad B.v = 2^{B.s} \end{array}$$

$B.s$ thus represents the position of the digit w.r.t. the decimal point. It must be zero for the digit immediately to the left of the dot, one for the second to the left, and so on increasingly to the left, and minus one for the digit immediately to the right of the dot and so on decreasingly to the right. We are now left with the problem of computing and transmitting this value to the B nodes.

This transmission must be performed through the L nodes. The simplest way to do it is to specify that the scale of a string of digits derived from an L node is that of its rightmost digit. This leads to the following semantic rules:

$$p_3\colon\ L_0 ::= L_1\ B \qquad B.s = L_0.s \qquad L_1.s = L_0.s + 1$$

$$p_4\colon\ L ::= B \qquad B.s = L.s$$

We now come to the end, that is computing the scale of the topmost lists of digits in productions p_1 and p_2. This is no problem for the L node to the left of the dot—or the unique L in production p_1—whose scale is zero, according to the discussion above:

$$p_1\colon\ Z ::= L \qquad L.s = 0$$

$$p_2\colon\ Z ::= L_1\ .\ L_2 \qquad L_1.s = 0$$

According to the same discussion above, the value of $L_2.s$ in p_2 must be the scale of its rightmost digit, that is the opposite of the length of its string of digits. But this information is not directly available in p_2! So we have to introduce a third and last attribute, l for length, attached to L and of type integer:

$$p_2\colon\ Z ::= L_1\ .\ L_2 \qquad L_2.s = -L_2.l$$

The rules needed or computing this length are very simple:

$$p_3\colon\ L_0 ::= L_1\ B \qquad L_0.l = L_1.l + 1$$

$$p_4\colon\ L ::= B \qquad L.l = 1$$

This concludes the specification of our AG, which is gathered in Fig. 4.2. In AGs describing (parts of) compilers, the semantic rules will be used to express the insertion of an identifier and associated information in a symbol table, the typing rules of the language, code generation, etc. Let us make a few additional (and general) comments.

4.5 Synthesized and inherited attributes

Attributes v and l are, in each relevant production, computed for the left-hand side non-terminal in terms of other attributes, generally attached to the non-terminals in the right-hand side. This means that they propagate information from the leaves of the parse tree up to the root. They are called *synthesized* attributes. By contrast, attribute s is defined in relevant productions on RHS non-terminals in terms of attributes attached to the LHS terminal, i.e. it propagates information from the context of a subtree down to where it is needed. It is called an *inherited* attribute. Each attribute of each non-terminal is either synthesized or inherited, i.e. it cannot behave as a synthesized attribute in some productions and as an inherited one in other productions.

The distinction between synthesized and inherited attributes is not only a useful classification means. It also has more theoretical grounds. But to explain them we must introduce the notions of semantic tree and attribute instance.

$$G = (N, T, P, Z)$$

$$N = \{Z, L, B\},\ T = \{\mathtt{0}, \mathtt{1}, \mathtt{.}\},\ P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$$

$$A = \{v, s, l\},\ A(Z) = \{v\},\ A(L) = \{v, s, l\},\ A(B) = \{v, s\}$$

$$\text{type}(v) = \text{real},\ \ \text{type}(s) = \text{type}(l) = \text{integer}$$

p_1: $Z ::= L$
  $Z.v = L.v$
  $L.s = 0$

p_2: $Z ::= L_1\ .\ L_2$
  $Z.v = L_1.v + L_2.v$
  $L_1.s = 0$
  $L_2.s = -L_2.l$

p_3: $L_0 ::= L_1\ B$
  $L_0.v = L_1.v + B.v$
  $B.s = L_0.s$
  $L_1.s = L_0.s + 1$
  $L_0.l = L_1.l + 1$

p_4: $L ::= B$
  $L.v = B.v$
  $B.s = L.s$
  $L.l = 1$

p_5: $B ::= \mathtt{1}$
  $B.v = 2^{B.s}$

p_6: $B ::= \mathtt{0}$
  $B.v = 0$

Figure 4.2: The complete binary number AG

4.6 More on trees and attributes

A *semantic tree* is a parse tree augmented with slots at each node corresponding to the attributes of the non-terminal labelling the node. The semantic tree corresponding to the parse tree of Fig. 4.1 is depicted in Fig. 4.3; conventionally, inherited attributes are shown on the left of the non-terminal and synthesized ones on the right. Each of these slots is called an *attribute instance*. The numbers on the left of or below the "records" corresponding to the nodes will be explained and used below.

Now, the theory of AGs—and also the common sense—requires that the value of each attribute instance be specified by one and only one semantic rule. But, apart from the root, every non-terminal node is "surrounded" by two productions, one "below" the node, for which the non-terminal is the LHS, and one "above", for which the non-terminal is one of the RHS symbols. Since semantic rules (which define *attribute occurrences* and hence a number of attribute instances) are attached to the productions, each attribute instance is potentially defined by two semantic rules: one in the production below the corresponding node, and one in the production above. Many ways to solve this problem and enforce the single definition rule could be thought of, but Knuth chose the simplest one: he decided that every attribute (representing a whole set of attribute instances) must always, whatever the context in which its corresponding instances appear, be defined by a semantic rule of either the production "below" or the production "above"; the former are the synthesized attributes, whereas the latter are the inherited ones. This explains why an attribute cannot be both synthesized and inherited. This also explains why the start symbol of the grammar cannot have any inherited attribute, because it has no production "above" to define them.

This also implies that each production must come with exactly one definition (semantic rule) for each synthesized attribute of the LHS non-terminal and one for each inherited attribute of each RHS non-terminal, and no more. These attribute occurrences are called the *defined occurrences* of the production, and only defined occurrences may (and must!) appear

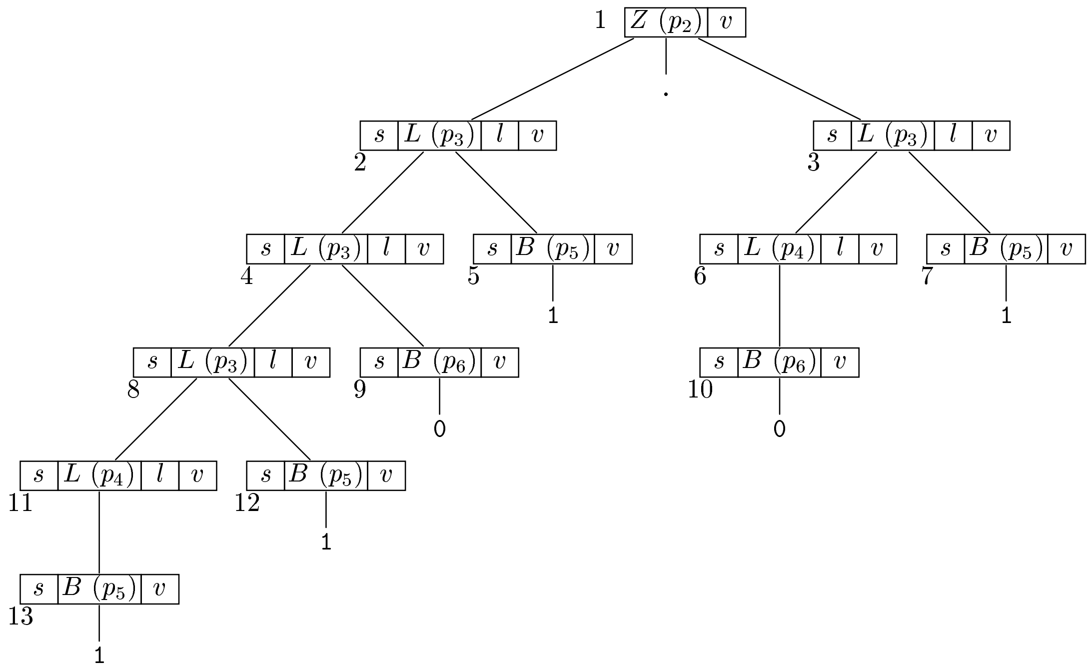


Figure 4.3: The semantic tree corresponding to the parse tree of Fig. 4.1

in the LHS of a semantic rule. Occurrences allowed to appear in the RHS of semantic rules are called *applied* or *used occurrences.* If the AG is in (Bochmann) *normal form* [Boc76] then applied occurrences are disjoint from defined ones, i.e. they are the inherited attributes of the LHS non-terminal and the synthesized attributes of the RHS non-terminals; otherwise, in addition to those, an applied occurrence may also be a defined one. Our example AG is in normal form. Normal form AGs are considered easier to read and easier to use in theoretical discussions; in addition there exists a simple textual macro-expansion transforming any AG into a normal form one, so imposing normal form is not a very stringent requirement.

4.7 Semantics of an attribute grammar

Now, what is the semantics of such a specification? That is, given a sentence belonging to the language generated by the underlying context-free grammar, how do we compute the "meaning" of that sentence according to the AG? First we parse the sentence and construct the corresponding semantic tree; how this is done is irrelevant to the theory of AGs, provided that we end up with exactly one tree. Then, we consider each attribute instance in that tree as a variable which must be given a value. These variables are defined by a system of equations derived from the semantic tree and the semantic rules as follows: first, each node in the semantic tree is given a unique number. Then, for each attribute a of the non-terminal labelling node i, we introduce the variable a_i. Now this variable a_i, which is exactly an attribute instance, corresponds to a unique defined attribute occurrence in a unique production, as explained above; we then introduce a new equation in the system, which is a copy of the semantic rule defining that attribute occurrence, with the LHS being replaced by a_i and the attribute occurrences in the RHS being replaced by the corresponding variables. The system of equations corresponding to our example AG and the semantic tree of Fig. 4.3

$$
\begin{array}{ll}
v_1 = v_2 + v_3 & v_6 = v_{10} \\
s_2 = 0 & s_{10} = s_6 \\
s_3 = -l_3 & l_6 = 1 \\
v_2 = v_4 + v_5 & v_7 = 2^{s_7} \\
s_4 = s_2 + 1 & v_8 = v_{11} + v_{12} \\
s_5 = s_2 & s_{12} = s_8 \\
l_2 = l_4 + 1 & s_{11} = s_8 + 1 \\
v_3 = v_6 + v_7 & l_8 = l_{11} + 1 \\
s_7 = s_3 & v_9 = 0 \\
s_6 = s_3 + 1 & v_{10} = 0 \\
l_3 = l_6 + 1 & v_{11} = v_{13} \\
v_4 = v_8 + v_9 & s_{13} = s_{11} \\
s_9 = s_4 & l_{11} = 1 \\
s_8 = s_4 + 1 & v_{12} = 2^{s_{12}} \\
l_4 = l_8 + 1 & v_{13} = 2^{s_{13}} \\
v_5 = 2^{s_5} &
\end{array}
$$

Figure 4.4: The system of equations for decorating the semantic tree of Fig. 4.3

(the nodes being uniquely identified by the numbers on their left or below them) is shown in Fig. 4.4.

Because of our condition on the "well-formedness" of an AG, we know that a system of equations constructed as explained above is well-formed, i.e. each variable is defined by one and only one equation. But this is not sufficient to ensure that it can be solved: we must also require that it be *non-circular*, i.e. no variable is used, directly or indirectly, to compute itself.[1]

An AG for which all the systems of equations corresponding to all its possible semantic trees are non-circular is itself called non-circular. AGs which are not non-circular are considered as specifying "impossible" computations and are thus not used. An interesting point to note here, although this will be discussed more at length below, is that the non-circular property is intrinsic to the AG, which means that it can be verified on the AG itself, without constructing all the possible trees and systems of equations, using a rather simple—although expensive—algorithm.

Our example AG is non-circular, and so is the system of equations of Fig. 4.3. We can hence solve it by a simple topological sort, beginning with variables defined by constants (e.g. s_2), then variables depending only on those latter variables (e.g. s_4), and so on, until every variable has a value. This resolution process, when thought of in terms of attribute instances in the semantic tree, is called *tree attribution* or *tree decoration*. A semantic tree in which all the attribute instances have a value which is correct w.r.t. the AG at hand is called a (*consistently*) *decorated tree* (or attributed tree). The attributed tree corresponding to the semantic tree of Fig. 4.3 is shown in Fig. 4.5.

The meaning, or semantic value, of the input sentence is then, depending on the authors or the context or the use, either the fully attributed tree or only the values of a subset of its attribute instances, generally the (synthesized) attributes of the root (i.e. the start symbol). In our case, the meaning of the sentence `1101.01` is the value of attribute v of the root, i.e.

[1] This is in principle not mandatory if the domains of the variables and the mappings used in the equations are such that the system admits a fixed point, but since we want to be able to deal with any domain and any mapping, we must impose more "syntactic" requirements.

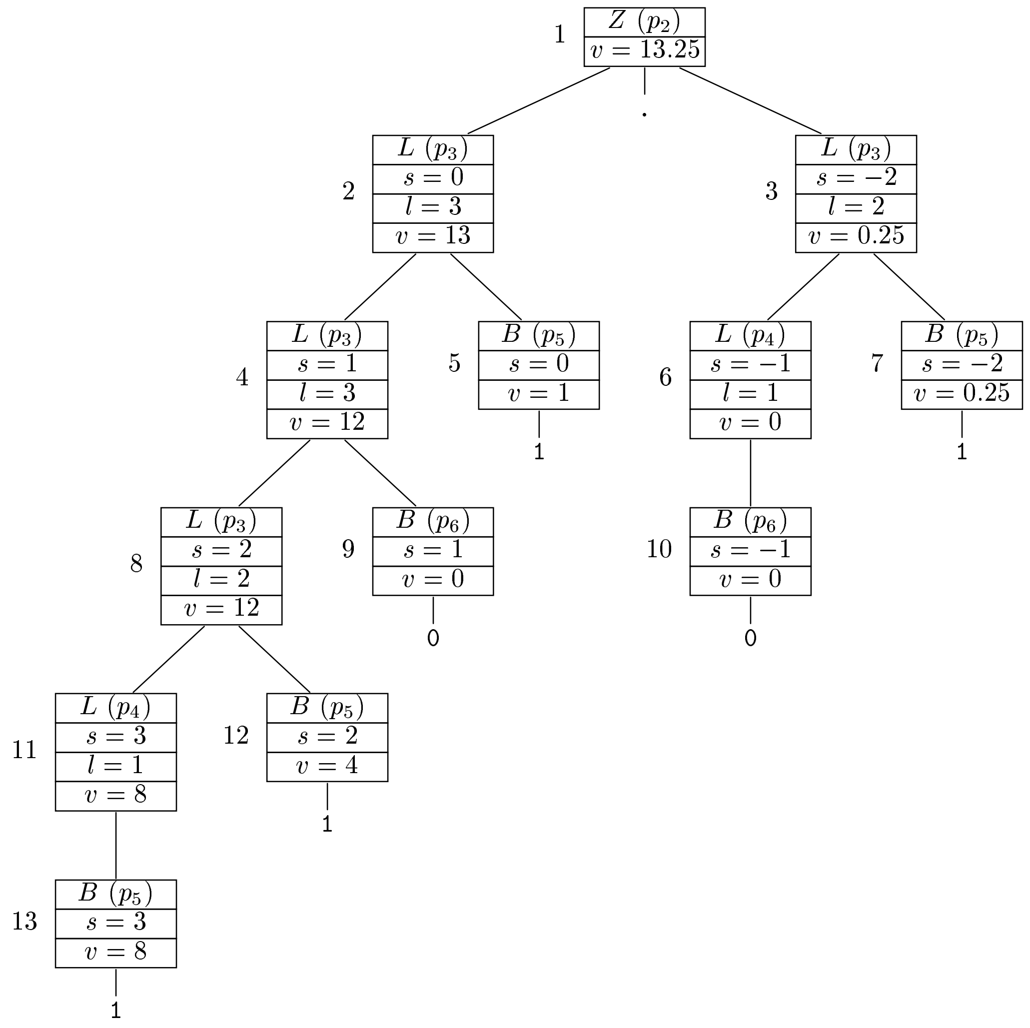


Figure 4.5: The fully attributed tree corresponding to the semantic tree of Fig. 4.3

13.25, which is, as expected, its decimal value (Gee, how great!).

This discussion about the system of equations gives us many insights about the notions of attributes and semantic rules. An attribute is a single-assignment variable, whose value cannot change after it has been set, and semantic rules are equations which relate the values of these attributes. The term "equation" should be given here its purely mathematical meaning, i.e. semantic rules should involve only pure, side-effect-free functions (although they also have an operational semantics, namely the assignment of some value to some variable). This requirement to avoid side-effects is justified by the (operational) resolution process: a given system of equations generally admits several resolution orders, i.e. the order in which the semantic rules are "executed" is generally not unique. If these semantic rules involve side-effects, their order becomes highly unpredictable, which makes their semantics not clearly defined;[2] to avoid this difficulty, we simply forbid side-effects. The language used to express the semantic rules should enforce this rule by being strictly applicative.

[2]Furthermore, it is also possible that a semantic rule be executed more than once or none at all.

4.8 Attribute grammars as a specification tool

We can now clearly see the advantages of AGs as a specification method:

- it is a *declarative* (non-procedural) method: you specify only *what* must be computed, and not *how* to do it; the semantics of an AG is thus independent of its implementation(s);

- it is a *structured* method: each production, together with its semantic rules, acts as a "black box" which is independent from the others and communicate with them through well-defined interfaces (the non-terminals and their attributes);

- AGs feature a strict *locality of reference*: in each semantic rule you can see and use only the attributes local to the production at hand, and the relations between the attributes are thus decomposed into small size, easily manageable pieces.

This explains why programming with AGs is so easy and highly productive: developing an AG is much quicker than writing by hand an equivalent program which is blurred by operational considerations.

The fourth advantage of AGs over other specification methods is that they are *executable*: it is possible to automatically construct program which will implement the translation specified by a given AG. We explain now how this is achieved.

4.9 Dependencies and first evaluation method

The key notion here is that of *dependency*. Remember the discussion about the resolution of the system of equations: we start with the variables defined by constants, i.e. those depending on no other variable, then we proceed with the variables depending only on the latter, and so on. We say that a variable a_i depends on a variable a_j if a_j appears in the RHS of the equation defining a_i. For each semantic tree and associated system of equations, we can draw the graph of this dependency relation, which is conventionally pictured upon the semantic tree; for instance, Fig. 4.6 represents this graph for the semantic tree of Fig. 4.3. This graph is called the *compound dependency graph* of the tree (or input sentence). Note that, for historical reasons, this graph is not exactly the dependency graph, but rather its inverse, the *graph of information flow*. However using one instead of the other causes no problem, so we will stick to the tradition.

This graph is exactly the one that we have to topologically sort to get the order in which the system of equations will be solved. This also gives us the first method to implement the translation defined by a non-circular AG, as follows:

1. given an input sentence, parse it and construct its semantic tree;

2. construct the compound dependency graph and the system of equations;

3. topologically sort the compound dependency graph, and use the generated order to solve the system of equations, storing the values in the semantic tree;

4. deliver the final semantic value.

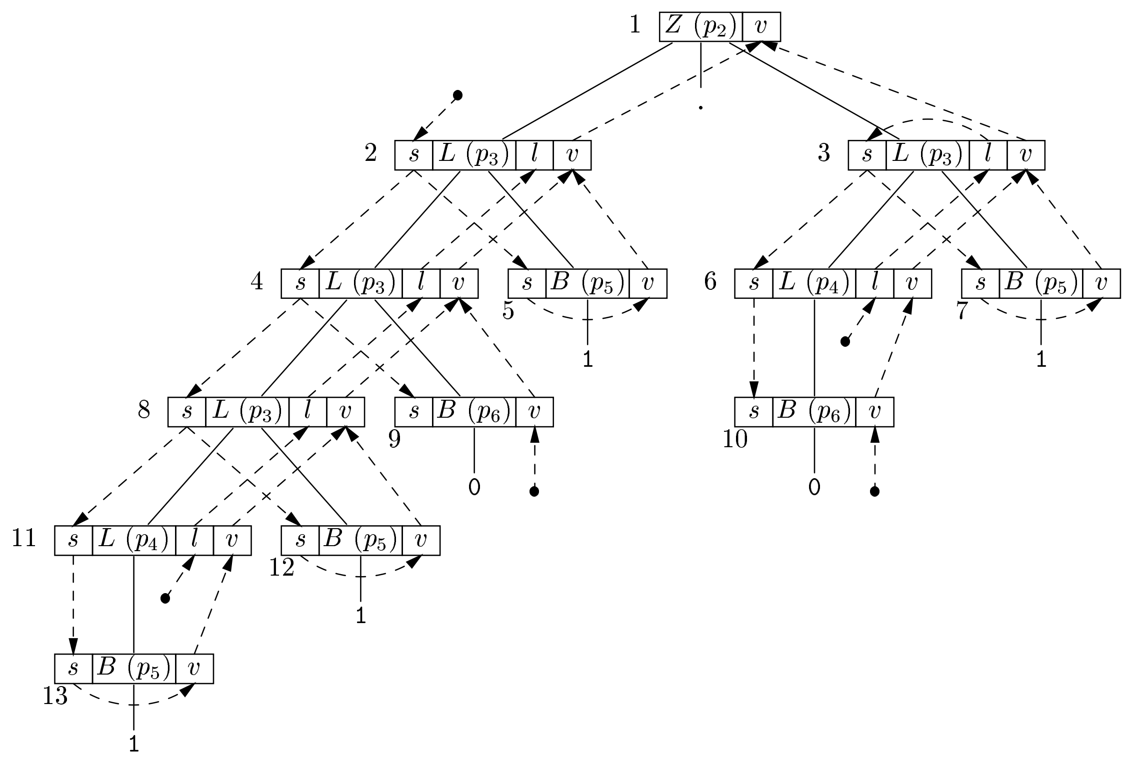


Figure 4.6: The compound dependency graph (directed edges) of the semantic tree of Fig. 4.3

Many variations of this algorithm are possible, to improve the efficiency of the process; for instance, there is no need to explicitly construct the system of equations, since the combination of the semantic tree, the compound dependency graph and the semantic rules gives all the necessary information.

This algorithm is easy to understand, and it is applicable to any non-circular AG. It was used, in some form or another, in several practical systems (e.g. DELTA, FOLDS, NEATS, see [DJL88] or [AM91] for more information). However it is not very efficient, because, for large source texts, the compound dependency graph becomes huge and topologically sorting it becomes very expensive w.r.t. the cost of the "execution" of the semantic rules (which any method must perform, of course). So many authors did a lot of research work in order to devise more efficient methods (see again [DJL88] or [AM91]). The key idea is to try to replace the dynamic evaluation order induced by the topological sort by a more static order, determined *a priori* or by static analysis of the AG itself.

4.10 Synthesized dependency graphs and a non-circularity test

Indeed, the notion of dependency between attribute instances (a dynamic notion) can be transposed into the notion of dependency between attribute occurrences, a static notion which we can study on the AG itself rather than on its semantic trees. Semantic rules induce dependencies between the various attribute occurrences appearing in a production. These dependencies can be represented by a graph which is called the *production dependency graph*. Fig. 4.7 represents the set of production dependency graphs for our example AG. The relation between the compound and production dependency graphs is that, to get the former, you

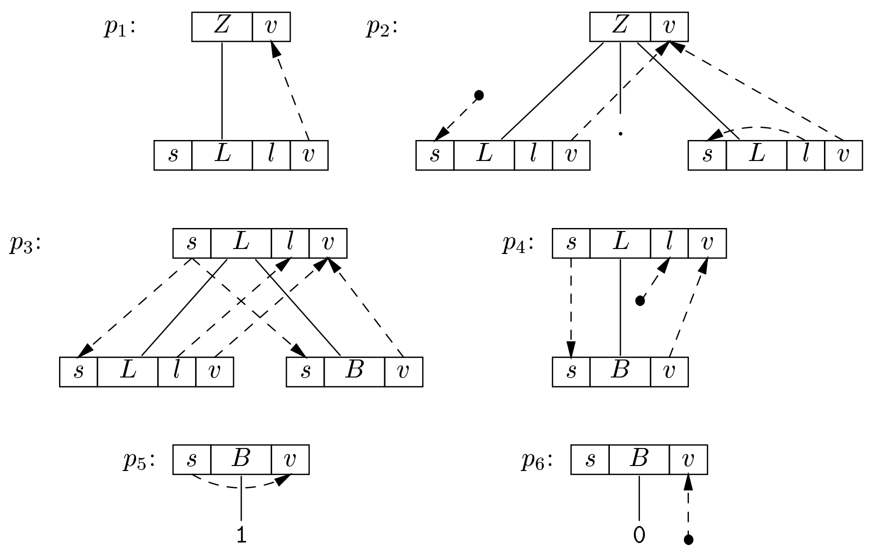


Figure 4.7: Production dependency graphs for the AG in Fig. 4.2

simply paste together copies of the latters, identifying common "interfaces", according to the structure of the parse tree; this gives both the semantic tree and the compound dependency graph. For instance, from Fig. 4.1 and 4.7 you can draw Fig. 4.6.

The fact that all possible compound dependency graphs are composed from statically-known production dependency graphs is important because it allows to statically simulate all possible dynamic situations. For instance, the already mentioned notion of non-circularity, which is defined in dynamic terms—the non-circularity of every possible compound dependency graph—can be tested statically on the AG itself by using the production dependency graphs. This is done by computing, for each non-terminal, a set of graphs on its attributes—another kind of dependency graphs!—, called *synthesized dependency graphs* or *inferior characteristic graphs*, which simulate the effects of all possible (parts of) compound dependency graphs issued from the non-terminal. Consider again the graph of Fig. 4.6; Fig. 4.8 shows the subgraph issued from node 4, labelled by non-terminal L, and the projection of the transitive closure of this subgraph on the attributes of L; this latter graph is one of the synthesized dependency graphs of L.

The sets of all possible synthesized dependency graphs for all non-terminals of an AG are constructed by a simple algorithm, to be presented soon, which involves the construction of *augmented production dependency graphs*. Such a graph is a production dependency graph to which we paste, on each non-terminal of the RHS, one of its synthesized dependency graphs. This process is exemplified in Fig. 4.9. The projection of the transitive closure of such an augmented graph on the attributes of the LHS non-terminal is a (new) synthesized dependency graph. The whole algorithm is presented in Fig. 4.10.

Although this algorithm is theoretically intrinsically exponential, it is quite tractable in practice if implemented with enough care [JP90]. The sets of sdg's for our example grammar are presented in Fig. 4.11. Note—and this is no accident—that edges in sdg's always flow from inherited to synthesized attributes (if the grammar is in normal form).

These sdg's can be used for many purposes; some of them will be explained below, but let us concentrate on testing non-circularity. Consider a circular compound dependency graph, and a particular cycle in this graph. This cycle is either wholly contained in a single production

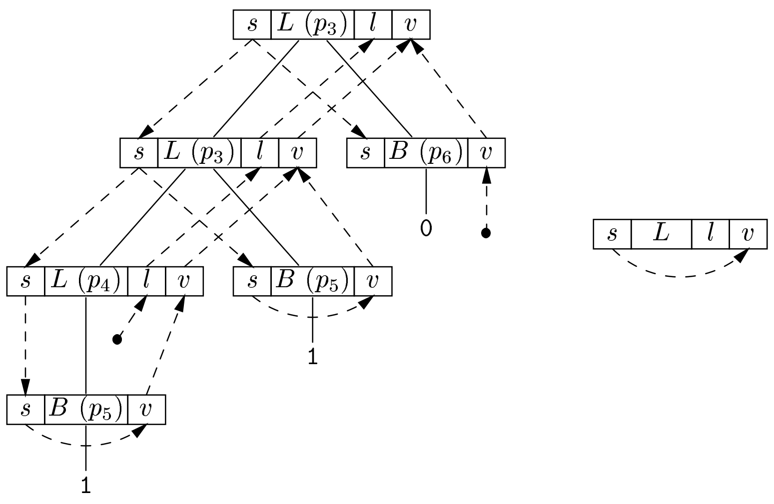


Figure 4.8: A subgraph of the compound dependency graph of Fig. 4.6 and its associated synthesized dependency graph for L

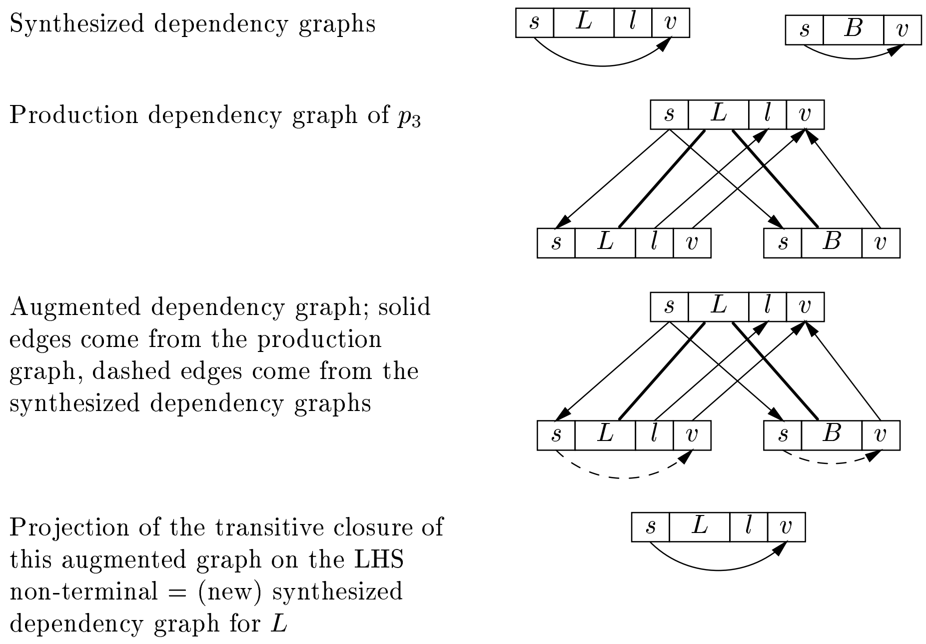


Figure 4.9: Construction of an augmented production dependency graph

```
1.  set the set of sdg's of each non-terminal to the empty set;
2.  repeat convergence ← true;
3.      for each non-terminal X do
4.          for each production p with X as LHS do
5.              for each combination of sdg's of the non-terminals
                        in the RHS of p do
6.                  construct the corresponding augmented dependency graph;
7.                  compute the transitive closure of this graph;
8.                  project this closure on the attributes of X;
9.                  if this new sdg is not already in X's set then
10.                     add it to that set;
11.                     convergence ← false
                    endif
                endfor
            endfor
        endfor
12. until convergence.
```

Figure 4.10: Construction of the whole set of sdg's

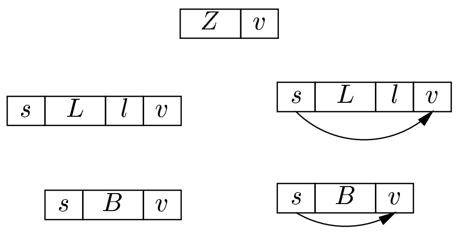


Figure 4.11: Synthesized dependency graphs for the AG of Fig. 4.2

dependency graph or spread over several productions. The former case is easily detected on the AG, and is an obvious mistake; note that this cannot happen in normal form AGs. In the latter case there always exists, in the "upper" part of the cycle, some production which contains an edge from some synthesized attribute of some non-terminal in the RHS to some inherited attribute of another (or the same) non-terminal in the RHS, such that the cycle is broken if we remove from it this production and associated dependencies.

Now the rest of the cycle can be "summarized" by a sdg for each of the non-terminals in the RHS of this particular production. Thus, if we construct the augmented graph for this production using those sdg's, the cycle will unavoidably show up again.

So a given AG is non-circular if and only if every augmented graph constructed in the course of the above algorithm is non-circular. This can easily be checked by inserting the following test between lines 7 and 8:

```
7'.                 if this graph is circular then the AG is circular;
```

4.11 Inherited dependency graphs

In the same way that we have defined sdg's which represent dependencies inside a pruned out semantic (sub-)tree, we can also define *inherited dependency graphs* which represent, for each non-terminal, the possible dependencies outside subtrees derived from it, as pictured in

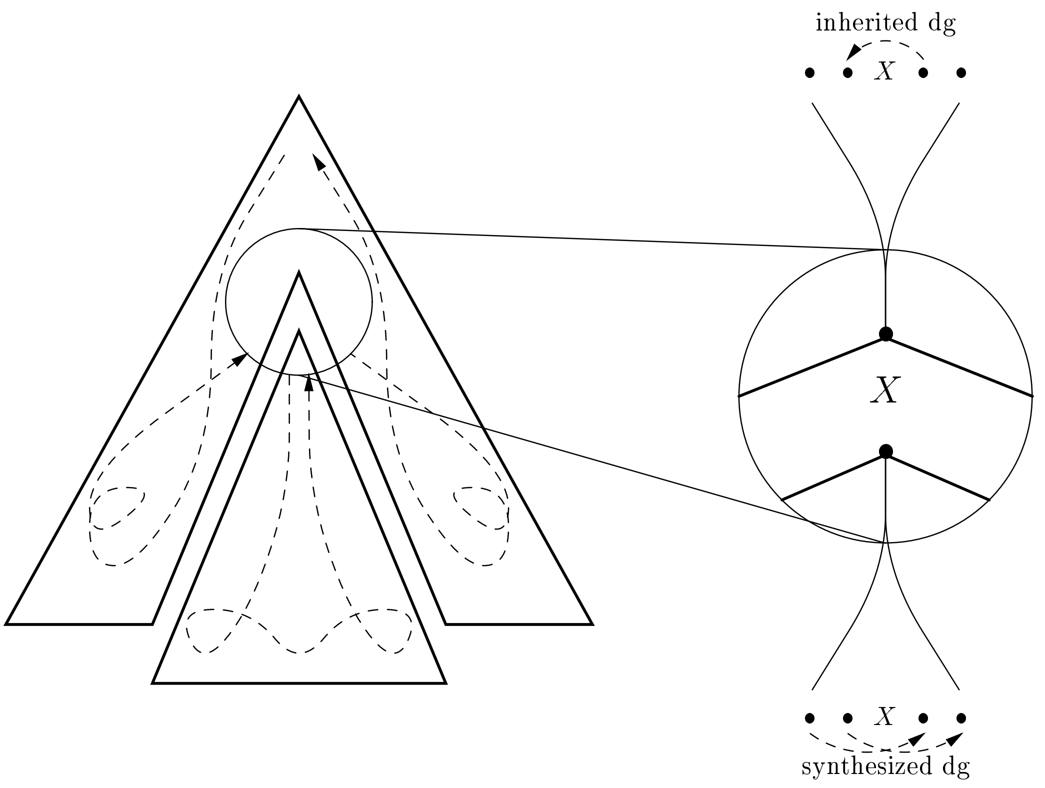


Figure 4.12: Inherited dependencies and their graphs

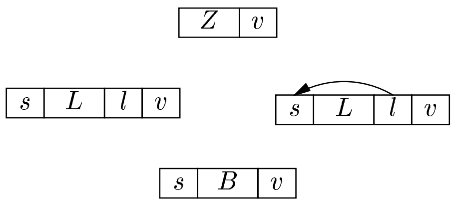


Figure 4.13: Inherited dependency graphs for the AG in Fig. 4.2

Fig. 4.12. Those idg's can be computed by an algorithm similar to that for sdg's; those for our example AG are presented in Fig. 4.13. If the AG is in normal form, edges in idg's flow from synthesized to inherited attributes.

4.12 Another evaluation method

The combination of sdg's and idg's is interesting because it gives the various orders in which the attributes of a given non-terminal may be evaluated. If we can find, for each non-terminal, a total order on its attributes which is compatible (in the sense of order relations) with each of its idg's and sdg's, then we say that the AG is *l-ordered* or *uniform*; in addition, those total orders must verify a property of *closure*.

This property states that when you augment any production dependency graph with the total orders on each non-terminal of the production (as in Fig. 4.9), then the projection of that augmented graph on each of the non-terminals is included in its given total order.

Z: v
L: $l \to s \to v$
B: $s \to v$

p_1: `BEGIN 1; VISIT 1, 1; EVAL` s`, 1; VISIT 2, 1; EVAL` v`, 0;`
`LEAVE.`
p_2: `BEGIN 1; VISIT 1, 1; EVAL` s`, 1; VISIT 2, 1; VISIT 1, 2;`
`EVAL` s`, 2; VISIT 2, 2; EVAL` v`, 0; LEAVE.`
p_3: `BEGIN 1; VISIT 1, 1; EVAL` l`, 0; LEAVE;`
`BEGIN 2; EVAL` s`, 1; VISIT 2, 1; EVAL` s`, 2; VISIT 1, 2; EVAL` v`, 0;`
`LEAVE.`
p_4: `BEGIN 1; EVAL` l`, 0; LEAVE;`
`BEGIN 2; EVAL` s`, 1; VISIT 1, 1; EVAL` v`, 0; LEAVE.`
p_5: `BEGIN 1; EVAL` v`, 0; LEAVE.`
p_6: `BEGIN 1; EVAL` v`, 0; LEAVE.`

Figure 4.14: Evaluation orders and visit-sequences for the AG of Fig. 4.2

Not every AG falls in the l-ordered class, but when this is the case, we can construct a very efficient deterministic evaluator, because no dynamic ordering is needed. Such an evaluator is an automaton driven by *visit-sequences*, one for each production. A visit-sequence is a sequence of instructions of four kinds:

`BEGIN` i indicates the start of the i-th visit to (the LHS non-terminal of) the production;

`EVAL` a, k performs the evaluation of attribute a of the k-th non-terminal in the production (0 for the LHS, 1, 2, ..., for the RHS); the attributes used in the RHS of the corresponding semantic rule are guaranteed to have been evaluated previously;

`VISIT` i, k is a sort of "procedure call" to begin the execution of the i-th visit of the k-th non-terminal in the RHS of the production; when this visit is complete, execution of the current sequence resumes after the `VISIT` instruction;

`LEAVE` marks the end of the current visit and returns to the calling sequence.

Our example AG is l-ordered; the total evaluation orders for each non-terminal and the visit-sequences for the corresponding evaluator are presented in Fig. 4.14. Obviously, visit-sequences can be implemented very efficiently.

Note that it is formally possible to construct, for any non-circular AG, an equivalent (but larger) l-ordered AG, but the resulting evaluator will be less simple because it must manipulate dynamic parameters in addition the the tree and the visit sequences.

4.13 Space optimizations

When the result of the translation is not the fully attributed tree but only a few values, we can use the fact that some attribute instances "die", i.e. become useless, before some other are computed so as to reduce the global space consumption of the evaluation process; this is achieved by sharing a same unit of storage between several attribute instances whose lifetimes are non-overlapping. In particular, if we can avoid storing a fairly large proportion of attribute instances in the tree, whose lifetime is virtually endless, this can be very effective. So we can try to store some attribute instances in global variables or global stacks (whose height is bounded by the height of the tree, rather than by the size of the tree if they are stored at the nodes).

Of course, this is not possible for any AG because, to avoid run-time overhead, decisions regarding where to store the attributes must be taken statically at evaluator generation time. But this is fairly easy for l-ordered AGs, because the visit sequences statically give us all the information on the lifetime of every attribute instance.

It is not the purpose of this tutorial—which is already too long!—to explain in every detail how this is done, but let us show the results of this lifetime analysis on our example AG. Attribute l can obviously implemented as a global variable, because each of its instances dies at the same moment another one is computed. Attribute s must be (and can be) implemented in a stack, because in production p_3 we must save the value of $L_0.s$ before visiting the L_1 subtree, so that it remains available for visiting B; the height of this stack is the height of the tree. Attribute v will be implemented as one global variable for all the instance of $B.v$ and a global stack for all instances of $L.v$ (because of production p_2).

Of course, this example is so small that these optimizations do not encounter any difficulty, but experience on large practical AGs show that they are actually quite effective. Note that they are the "automatic" counterpart of the analysis that a compiler writer does by hand.

4.14 Dead ends

An acute reader will have noticed that the compound dependency graph of Fig. 4.6 is not connected; in particular all the instances of attribute $L.l$ in the subtree to the left of the dot do not contribute to the computation of the final semantic value $Z.v$, and hence can be considered useless. These instances are called *dead-ends*. This situation is far from exceptional, and an efficient implementation must try to avoid loosing time in computing dead-ends.

This is easy to achieve in evaluators based on dynamic ordering. Traversing the compound dependency graph in reverse order, starting with the distinguished attribute instances forming the semantic value, will reach only the variables which are not dead-ends and which must actually be evaluated.

In visit-sequence based evaluators, this is more difficult to achieve because the evaluator cannot modify its behavior according to dynamic situations—apart from choosing the right visit sequence to execute depending on the production labelling the visited node. So, for instance, the evaluator of Fig. 4.14 will fail not to evaluate attribute s of B nodes deriving "`0`". However it is possible, after all the visit sequences have been constructed, to reanalyze them in order to detect when a whole visit is useless because it computes only definite dead-ends. As said above, it is the case of the first visit to the leftmost L son of the root, which computes $L.l$; we may hence delete the "`VISIT 1,` 1" instructions from the sequences of p_1 and p_2.[3]

This technique can be quite effective, but it must be used with great care and requires a considerable amount of analysis.

[3] However we must not modify the visit sequences of p_3 and p_4, because all their instructions are necessary for the subtree on the right of the dot.

4.15 Simpler evaluation methods

An evaluator based on visit-sequences is able to walk around the tree in any necessary order. But this was long considered as expensive and historically, after the first evaluators based on dynamic ordering (which were soon recognized as inefficient), many authors thought they could drastically reduce the running time by imposing *a priori* the evaluation order (and thus the tree walk order), and merely checking if the AG at hand could fit in this order.

The first orders which were considered were simple depth-first, left-to-right or right-to-left *passes*. The constraint imposed on the AGs was that, for each attribute, all of its instances could be evaluated in the same pass. For instance our example AG is evaluable in two passes; either left-to-right or right-to-left, the first one evaluating $L.l$ and the second one evaluating all the other attributes. Many systems were and still are based on such techniques, but their simplicity does not compensate their drawbacks, the main one being that the classes of accepted AGs are too restrictive (another "drawback" is that, contrary to what was expected, these techniques are not more efficient that those based on visit-sequences, while the l-ordered class is much larger than any pass-based class). Only two special cases deserve some attention:

- When the passes are strictly alternating, it is possible not to keep the whole tree in main memory, but instead hold its not currently used portions in secondary storage. This is exemplified by the old version of the LINGUIST system [Far84], which is able to construct non-trivial evaluators running on micro-computers with small memories.
- When the AG is evaluable in one left-to-right pass, it is possible to combine attribute evaluation with parsing and thus completely avoid building the parse tree. This yields very efficient evaluators, quite comparable to hand-written one-pass compilers, but the accepted AG class is also quite restricted (for instance our example AG, although very simple, does not fall in this class).

4.16 Hierarchy of AG classes

We have seen in the previous sections that the various evaluation methods were generally not applicable to every AG. More precisely, each method defines its own (set of) AG class(es). We have encountered three of them: non-circular, l-ordered, passes-based. The FNC-2 system accepts another class, named *strongly non-circular*, which lies between non-circular and l-ordered. There exists many more evaluation methods, leading to many more classes.[4] These classes actually form a complete hierarchy. Please refer to [DJL88] and [AM91] for more details.

4.17 Extensions

As with other fields of computer science, the basic theory of AGs was slightly modified or extended to make it usable in practice.

The first obvious limitation of pure AGs is that it is very difficult or cumbersome or tedious to completely specify the base language with context-free grammars, and syntactic analysis is generally supplemented by a lexical analysis phase. But then we need to be able

[4] Some authors working on the theory of AGs have even defined AG classes for which no evaluation algorithm is known.

to use, in the semantic rules, at least the actual text of the terminals of the grammar. This is achieved by considering that terminals have predefined attributes, whose value is set by the scanner. Every practical AG system includes such an extension.

Another less frequent extension of the same vein is the ability to process AGs based on extended CF-grammars, i.e. grammars with regular expressions in the RHS of the productions, or abstract grammars.

Another important extension is *incremental evaluation*. Remember the discussion in section 4.7 about *consistently attributed trees*, which are semantic trees such that all their attributes have a value and those values globally verify a system of equations derived from the semantic rules. Now imagine that you prune out of such a consistently attributed tree some attributed subtree, and graft in its place another attributed subtree pruned out of another consistent tree.[5] The resulting tree is fully attributed, but it is probably not consistently attributed. To make it consistent again, you can restart a complete evaluation from scratch, but this is expensive. An alternative is to try to reevaluate only those attributes which are inconsistent w.r.t. the new system of equations. This is what is done by incremental (re)evaluators. This technique has proven very useful in adding semantic capabilities to syntax-directed program editors, as exemplified by the Synthesizer Generator [Rep84].

The FNC-2 system embodies all these useful extensions.

4.18 Programming with AGs

The development of large AGs with systems which "naïvely" implement the basic theory set up by Knuth has uncovered methodological problems which are not compensated by the advantages of AGs as a specification tool. As an extreme example, let us quote a compiler front-end for the Ada language [UDP$^+$82] developed with the GAG system: it is a single, monolithic AG which fills up a complete volume of more than 500 pages! In these cases, the basic structuring device of an AG, the grammar productions, is too fine, and there is a need for other modularity tools.

The FNC-2 system provides two other ways to cut down a large application into manageable pieces:

1. An AG is seen as specifying a mapping which takes an attributed tree and produces another attributed tree; this implies that two or more evaluators can be "piped" in sequence to produce a larger application. Furthermore, this is achieved with no loss of efficiency since there exist techniques to build, from two "piped" AGs, a single AG which performs the same translation but without constructing the intermediate tree.

2. An AG itself can be structured into phases in which semantically related semantic rules can be grouped. This purely syntactic device is very useful to enhance the modularity and readability of an AG.

4.19 Further readings

The most comprehensive survey on AGs to date is the proceedings of a Summer School on AGs held in Prague in 1991 [AM91]. It contains more than 20 chapters on the theory,

[5] Of course, the non-terminals labelling the root of both these subtrees must be identical, so that the resulting tree is syntactically correct.

practice, applications and implementations (systems) of AGs, plus theoretical and practical relationships of AGs with other formalisms, written by the best specialists in the world. This book can be usefully complemented by the one by Deransart, Jourdan and Lorho [DJL88]; the latter is older but it takes a different perspective that may be of interest; in particular, the second part describes more then 30 practical systems dealing with AGs and the last part is a near-exhaustive bibliography with about 600 references.

A shorter but still useful reading is the twin papers by Courcelle [Cou84] and Engelfriet [Eng84].

Lastly, other chapters in this manual contain interesting discussions which apply not only to FNC-2 but also to the whole field of AGs.

Good work with attribute grammars!

Part II

Olga Reference Manual

Chapter 5

Introduction and Rationale

> "The language designer should be familiar with many alternative features designed by others, and should have an excellent judgement in choosing the best, and rejecting any that are mutually inconsistent. He must be capable of reconciling, by good engineering design, any remaining minor inconsistencies or overlaps between separately designed features. He must have a clear idea of what the scope and purpose and range of application of his new language, and how far it should go in size and complexity. (...) One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation."
>
> C.A.R. Hoare [Hoa73].

> "(...) a programming language designer needs the expertise of a scientist, the precision of a mathematician, and the taste of an artist as well as the pragmatism of an engineer (...)"
>
> R.D. Tennent [Ten81].

OLGA—an acronym for the French wording "Ouf! un Langage pour les Grammaires Attribue'es", which roughly means "At last! a Language for Attribute Grammars"—is the input language of the FNC-2 system and, as such, was especially designed to describe attribute grammars. This chapter introduces its main principles and the history of its design.

5.1 Background

Since Knuth's historical paper [Knu68] introducing attribute grammars, they have become a method of choice for describing and implementing syntax-directed computations, in particular compilers, translators and syntax-directed editors. Many systems have been developed around the world which produce efficient attributes evaluators [DJL88]. However, surprisingly few of these works were concerned in designing languages suited to attribute grammars writing. Indeed, most systems use as input language a variant of their implementation language (e.g. Lisp for FNC/ERN [Jou84], Algol-68 for HLP [RSS$^+$83], Pascal for MUG2 [Gie79], etc.) including notations to deal with attributes declaration and access.

The most achieved work in this domain is certainly the GAG system developed by Uwe Kastens [KHZ82] and its input language ALADIN ("A Language for Attributed DefINitions"). ALADIN is an applicative, strongly typed language, which includes many constructs which ease writing an attribute grammar. Indeed, GAG has been used in a large number of projects, including the development of very large attribute grammars such as a front-end for an Ada compiler [UDP+82]. ALADIN is translated into Pascal, but is not just an extension of Pascal: the only common features deal with types and type constructors. In our opinion, this is a good language, however some of its features are useless or even harmful, and some useful others are missing.

For a more complete review of ALADIN and GAG, see [KHZ82]. For a more complete comparison of the linguistic aspects of FNC/ERN, MUG2 and GAG, see [Bon86].

Our background also lies in the development of large attribute grammars—a full ISO Pascal to P_code compiler and a Pascal to Ada translator [Jou84]—with the FNC/ERN system, in which semantic rules must be written in Lisp. This helped us determine which features of the different existing systems are useful and which are inconvenient and/or must be banned. Several ideas can be found in [Par88].

5.2 Goals

This section lists a number of qualities which, in our opinion, a language for describing attribute grammars should have. The first set of features stems from the fact that this language must be, from the point of view of the attribute grammar writer, a useful and pleasant *tool*, and not a set of more or less awkward constraints.

This ideal language should be *easy to learn*, which implies that its constructions have a well defined, non ambiguous semantics, and that it respects most rules and habits of usual programming languages.

It should also provide the writer with as much *programming safety and reliability* as possible, while remaining very *versatile.* As many errors as possible should be detected at compile time. The language should also provide *helpful* features, including all the necessary structures regarding either control structures or the representation, manipulation and control of data.

It should also allow the development of very *readable* and *modular* programs.

The last set of features stems from the desire to make that language the best suited partner to the FNC-2 system, from the internal point of view.

It should be as *close* as possible *to Knuth's theory of attribute grammars*, providing all their expressive power but no more, so that algorithms developed in that frame (e.g. evaluation algorithms) be immediately applicable to an attribute grammar written in that language. To achieve this, it is necessary to ban any deviant construct, in particular those leading to side effects which could be interpreted differently by various systems.

The language should remain *independent from the evaluation method* which will eventually be chosen. Similarly, it should remain independent from, but be *easily translatable* into, most implementation languages (first targets to be considered are C, Lisp and Ada).

Lastly, it should provide constructs offering many opportunities for *optimizing* time and space consumption of the generated evaluators.

And, in the end, when all the above requirements are satisfied, the last feature of that ideal language should be *simplicity.*

5.3 Main Concepts

This section presents the main concepts of OLGA, together with a justification of their introduction in the language. Those justifications are related to the requirements listed in the previous section.

First of all, OLGA is an *applicative* language, i.e. there is no assignable variable, just expressions and strict functions. This means that it is impossible to make *side-effects* in OLGA. This is the first step towards programming safety on one hand, and, on the other hand, this formalism is very well suited for attribute grammars since it deals only with *values*. However, using pure attributes might prove cumbersome in some (rare) cases, such as generating fresh label numbers in a code generation phase for that purpose, OLGA provides connections with externally-defined functions which may make side-effects, but it must be strongly stressed that this falls entirely on the programmer's responsibility. Also, OLGA is not a *functional* language, in which functions are first-class objects, because this formalism is not well suited to compiled programs. So it is impossible in OLGA to pass functions as parameters, return them as function results or use function-valued attributes.

Secondly, OLGA is a *strongly typed* language, the other necessary condition for programming safety. *Type checking* is performed on any OLGA expression, in order to detect at compile time most inconsistencies and errors. However, this strong typing is not necessarily much constraining, since type declarations are in most cases *optional.* A powerful and versatile *type inference* mechanism is used rather than mere type checking to determine at compile time the type of every object, and functions may be *polymorphic* as in ML [CHP85], which means that they accept parameters of different types. Basic types include integers, reals, booleans, strings and (lexical) symbols. Type constructors include enumeration, ranges, records, unions, sets and (homogeneous) lists. Types may be recursive. OLGA provides for type tests and type conversions.

The syntactic base of attribute grammars written in OLGA is not a concrete syntax as in the classical theory [Knu68] but rather an *abstract syntax* as in [VM82], that is a tree grammar expressing the structure of notions of the language being described rather than the textual appearance of those notions. This frees the grammar writer from cumbersome syntactic constraints which may arise due to the parsing method chosen—e.g. the *dangling else* problem in the LR method, which leads to the duplication of nearly all the structured statements non-terminals and productions—and, on the other hand, allows smaller trees since keywords, simple productions and so on are excluded from the trees. The abstract syntax formalism we chose is very close to the one used in the programming environment Mentor, which is named Metal [KLM+83]: it provides for heterogeneous, fixed-arity and homogeneous, variable-arity *(lists)* operators, and comes with powerful tree construction actions to be either used in semantic rules or associated with parser reductions (see the corresponding part in this volume). OLGA provides special constructs for defining attributes of list productions.

Modularity is a basic feature of OLGA. Constants, types and functions definitions may be written in separate *modules.*. An attribute grammar itself may be structured into *phases* and, of course, *productions.* Each of these structuring levels supports the definition of local objects (constants, types, and functions). The visibility of local objects to other blocks is specified, as in Ada [ANS83], by declaring them in declaration modules, while other hidden objects may be defined in definition modules. Type checking is performed across modules frontiers. Import clauses also imply *dependencies* between modules, which are used by an (external) library manager to insure that an OLGA specification is always in a consistent state (see the corresponding chapter in *the* FNC-2 *Reference Manual*).

As the FNC/ERN system [Jou84], OLGA and FNC-2 provide a mechanism for generating missing attribute declarations and semantic rules. Attributes are divided into four kinds—inherited, synthesized, global and compound—which all have a set of *default rules* for determining how an undefined attribute of a given production should be computed. Also, it is possible to make attribute declarations optional in this case, default rules allow to determine the kind of an undeclared attribute and the phyla to which it is attached, whilst type inference determines its type. These mechanisms prove very useful when (incrementally) developing large attribute grammars.

Other miscellaneous features of OLGA, designed for ease of use, versatility and efficiency, are as follows:

- The only construction of OLGA involving side effects—apart of course from computing and storing attribute values—is the production of *error messages*; it was necessary to include it in OLGA—rather than leaving it to externally defined procedures—because, on one hand, it is very commonly used in applications of attribute grammars (e.g. in static semantic checkers in compilers), on the other hand, the attribute grammar is the place where all the necessary information is available—in particular the symbol to which the message should be attached—and, lastly, we needed to enforce the applicative character of this construct.

- Function names may be *overloaded* as in Ada [ANS83], i.e. two or more functions may have the same name provided they have different *profiles* (list of parameter types and result type); this proves to be a useful feature because it improves the semantic consistency of an OLGA specification—two similar operations on different types need not have different names—and the semantic consistency of OLGA predefined functions themselves. This notion is orthogonal to polymorphism.

- The **case. . . is** construct is extended to provide selection based on *pattern matching*.

- Declared constants may in some cases have their value computed at run-time when the block in which they are declared is entered; in this case they are named declared values. This is very convenient for block-wide common subexpressions. Since a production is a block, those values can play the role of *local attributes*, attached to operators (productions) rather than to phyla (non-terminals).

- It is allowed to refer to non-local attributes attached to nodes which are "upward" and "downward" in the tree this reduces the size of the grammar, because such global attributes need not be propagated into a whole subtree, and also offers opportunities for storage optimizations, allowing them to be implemented as (non-assignable) global variables.

- An *exception handling* mechanism is built into OLGA to deal with abnormal conditions.

- *Tree types* and *tree construction operations* are available in OLGA, allowing the result of an attributes evaluation phase to be a (possibly decorated) tree which may be input to another evaluator generated from an OLGA specification.

We hope that the above features are such that OLGA—and FNC-2 , for extra-linguistic considerations—meet all the requirements listed in section 5.2.

However, this volume describes only a preliminary version of OLGA and FNC-2 . We are aware that it has a number of imperfections. But we believe that the users will be sufficiently appealed by this version that they will be willing to participate in its improvement, by suggesting new features or changing existing ones.

5.4 How to Use this Manual

The lexical and syntactic descriptions of OLGA which appear in the following chapters of this manual, and are summarized in appendix B, are expressed in the two meta-languages of SYNTAX, *lecl* for lexicon and *bnf* for syntax. These languages are described in the *SYNTAX Reference Manual* [BDL87]. Note that the identifiers used in next chapter to designate lexical entities of OLGA, and in other chapters to designate syntactic entities (non-terminals), are used just for description and specification purposes and are NOT part of the OLGA language itself.

This manual is a reference manual, not a tutorial; the reader is thus assumed to have a working knowledge of attribute grammars and of at least one programming language among Pascal, C, Ada, Modula-2, CLU, Lisp, ML. For a tutorial on attribute grammars, see the first chapter of the *User's Guide* in this volume.

This manual is divided into two main parts, after chapter 6 which describes the lexical structure. The first part introduces those features of OLGA which make it a general-purpose applicative language: chapter 7 describes types and type constructors; chapter 8 describes functions and type checking; chapter 9 deals with expressions, constants and "values" (q.v.); chapter 10 introduces exception handling and, lastly, chapter 11 describes visibility rules, modules structure and import rules. The second part describes the other features of OLGA which make it an AG description language: chapter 12 describes attributed abstract syntaxes, chapter 13 deals with AGs proper, including attribute declarations and semantic rules, chapter 14 describes tree construction operations ans chapter 15 deals with the structure of AGs.

The information contained in this manual is impossible to present in a strict linear order. Forward references are hence unavoidable, although it was tried to keep them infrequent.

Chapter 6

Lexical Structure

This chapter contains complete information about the lexical structure of OLGA. The basic alphabet of OLGA is the ASCII character set on 7 bits.

6.1 Character Set

Characters are divided into a number of *classes*. Each character in a class plays the same lexical role as the others in the same class the difference between them is only semantical. Classes may be named or unnamed. Named classes are specified below; the other classes appear directly in the definition of lexical units and are denoted as the list of their member characters written between double quotes. A given character may belong to several classes.

```
LETTER             = "a".."z" + "A".."Z";
DIGIT              = "0".."9";
OCTAL              = "0".."7";
TRULY_PRINTABLE    = " "..#176 - "^\'\"\\";
                   -- printable ASCII characters except those
                   -- participating in escape sequences
CONTROL_CHAR       = "A".."_";
                   -- printable characters which correspond to
                   -- control characters with the "↑" notation
                   -- except @ (^ @ == NUL character, forbidden)
```

6.2 Comments

Comments are any sequence of spaces, horizontal and vertical tabs, form feeds, line feeds and newlines (carriage returns), and "meaningful" comments, which are (possibly nested) Pascal comments in which any ASCII character is allowed.

```
BLANKS             = SP + HT + VT + FF + LF + NL + CR;
                   -- the above identifiers are predefined in the
                   -- lexical metalanguage of SYNTAX

MEANINGFUL         = "{" @1 {^"{}"& True | "{"& True @1 | "}"&1 @2}*
```

```
                    ("}"&2 @2 | @7);
-- initialization:  comment-level = 0
-- &1:  true if comment-level > 1
-- &2 = not &1
-- @1:  increase comment-level
-- @2:  decrease comment-level
-- @7:  (on EOF only) complain if comment-level!= 0
```

Comments `= -{BLANKS | MEANINGFUL}+;`

Comments may be inserted anywhere between two other lexical units or left out, except when their presence is necessary to separate two units (e.g. two identifiers) whose meaning would be altered by their concatenation. Of course, comments may also appear before the first and after the last lexical unit of a text.

6.3 Identifiers

Most OLGA objects are designated by *identifiers.* An identifier begins with a letter, possibly followed by any number of letters or digits, optionally preceded by an underscore "_" or an hyphen "-".

```
ID                  = LETTER {["_-"] (LETTER | DIGIT)}*;

%ID                 = ID;
```

The case of letters, upper or lower, is significant. All characters in an identifier are significant (i.e. participate in comparisons) and there is no limit to their length (except maybe for implementation-dependent limitations).

6.4 Integral Numbers

Integral numbers are unsigned constants of the predefined type *int.* They are denoted by strings of (decimal) digits.

```
INT                 = {DIGIT}+;

%INT                = INT;
```

Lexically correct integral numbers may nevertheless be refused by the OLGA compiler if they denote values which do not fit in the (implementation-defined) range of type *int.* Integral values may also be expressed in other bases by using string conversion functions (see section 7.11); those functions will be computed at compile time if possible.

6.5 Real Numbers

Real numbers are unsigned constants of the predefined type *real*. They are denoted by strings of (decimal) digits, followed by a dot and a fractional part, and optionally an (optionally signed) exponent. Neither the integral part nor the fractional part may be empty.

```
SIGN             = [-"+" | "-"] ;

%REAL            = INT "." INT [("e" | "E" @Lowe_Case) SIGN INT];
```

The letter introducing the exponent may be written in lower- or uppercase. This is the only situation in which the case of a letter is not significant.

Lexically correct real numbers may nevertheless be refused by the OLGA compiler if they denote values which do not fit in the (implementation-defined) range of type *real*.

The value denoted by such a real number, when it is accepted, is the closest machine-dependent approximation of the mathematical value it represents.

6.6 Character Constants

Character constants are constants of the predefined type *char*, values of which are characters of the ASCII alphabet on 7 bits. They are denoted by one printable ASCII character embedded in simple quotes. Several *escape sequences* are used to express non-printable characters in a printable form.

```
PRINTABLE        = TRULY_PRINTABLE
                 | -"↑" CONTROL_CHAR @4
                 | -"\\" (TRULY_PRINTABLE @5 |
                         "↑\'\"\\" |
                         OCTAL&True [OCTAL& True[OCTAL& True]] @6);
                 -- @4:  convert a control character from "↑" notation
                 -- @5:  recognize and convert \c escape sequence
                 -- @6:  convert a character from its octal value
                 -- and complain if NUL

%CHAR            = -"'" PRINTABLE -"'";
```

An escape sequence of the form ↑`c` where `c` is a character in the ASCII alphabet between `"A"` and `"_"` (inclusive) denotes the character whose rank in the ASCII alphabet is that of character `c` minus that of character `"@"`; for instance ↑`A` denotes the character with rank 1 (SOH), ↑`H` denotes the character with rank 8 (BS) and _ denotes the character with rank 31 (US). The NUL character (rank 0) is forbidden in any form in character and string (see below) constants, because it has a special meaning in some implementations.[1].

Escape sequences of the form `\c` allow to denote characters introducing character constants, strings and escape sequences, and also to denote some other characters as in C. The following sequences are recognized:

[1]NUL is the end-of-string marker in C [KR78] The NUL character can always be forced in a character or string value by using the expression `char(0)` (see section 7.11.1) but, if it used, it is not guaranteed that the semantics of functions dealing with strings is correctly implemented, in particular if the target language is C.

Sequence	Denotes	Rank
`\b`	BS	8
`\t`	HT	9
`\n`	NL (LF)	10[2]
`\v`	VT	11
`\f`	FF	12
`\r`	CR	13
`\s`	SPACE	32
`\"`	"	34
`\'`	'	39
`\\`	\	92
`\↑`	↑	94

An escape sequence of the form `\npq` where `n`, `p` and `q` are octal digits between 0 and 7 (except `\000` which is illegal) denotes the ASCII character whose rank has octal value `npq`. For instance `\001` is the SOH character (rank 1), `\040` is the space character (rank 32) and `\177` is the DEL character, the one with highest rank (127). If the denoted octal value is greater than 127 (decimal), then it is truncated to fit on 7 bits. Leading zeroes may be omitted.

Any other unrecognized escape sequence `\c` denotes the character `c`.

6.7 Character Strings

Character strings are constants of the predefined type *string*. They are denoted by a (perhaps empty) sequence of printable characters embedded in double quotes. The same escape sequences as for character constants are allowed.

```
%STRING            = -"\"" {PRINTABLE} -"\"";
```

Note that a string constant with length 1 is different from a character constant.

6.8 Attributes

Attributes are designated in OLGA by special identifiers prefixed with a dollar sign.

```
%ATTR              = "$" ID ;
```

6.9 Type Denoters

Type denoters, which are explicit forms of type variables (see section 8.1), are designated with special identifiers prefixed with an "at" sign.

```
%TVAR              = "@" ID ;
```

They can used in type expressions (see chapter 8).

6.10 Operators and Separators

The following characters and character combinations are used as operators or separators in OLGA:

`- !  < > [ ] .  -> | & >= <= !~ ~ != % / * + ..  = # , ) ( := :  ; [| |]`

The syntactic use of those operators and separators will be described in the next chapters.

6.11 Keywords

The following attribute names and identifiers are keywords used for syntactic purposes and cannot be used as normal identifiers:

$arity $position $scx all and arity attribute case catch compound const declaration declare define definition else elsif end exception external first from function global grammar if import in inherited inline is last left let letrec list map match message module of or other phase position raise range record right set severity synthesized test then tree type undef union use value where with

The syntactic use of those keywords will be described in the next chapters.

6.12 Utility Non-Terminals

The following non-terminals are used throughout the rest of this manual. Their meaning is purely syntactic.

Syntax:

```
<ID>                  =  %ID ;
<INT>                 =  %INT ;
```

Briefly said, `<ID>` is used in the definition of a new identifier, whereas `<id>`, to be defined in chapter 11, designates a previously declared identifier.

Chapter 7

Types and Type Constructors

7.1 Introduction

OLGA is a strongly typed language as stated in the rationale, this is a necessary condition for ensuring programming safety. However, as OLGA uses a powerful *type inference* mechanism rather than a simpler type checking scheme, most type declarations for working attributes (in attribute grammars) and function parameters and results are optional (see chapters 13 and 7 respectively).

Type declarations are nevertheless useful to invent new types when the few predefined types are insufficient, which will be the case of nearly all applications written in OLGA. Also, most type declarations involve the implicit declaration of other OLGA objects such as constants of enumerated types and construction or conversion functions for all types.

This chapter aims at providing all information dealing with the definition of types in OLGA. Type constraints and the type inference mechanism are presented in next chapter, together with functions. Types and functions associated with the construction of decorated abstract trees described by an attributed abstract syntax are presented in chapter 14.

7.2 Type Declarations

Every type *definition* must be associated with a name which will be used to refer to it where useful (e.g. in attributes declarations), and which will also be used to construct the names of implicit construction and conversion functions associated to that type: this association is a type *declaration*. This means that anonymous types do not exist. See also the type expressions in next chapter and their use in other chapters, e.g. 9 and 10.

Syntax:

```
<type part>          =  type
                        <type decls>;
<type decls>         =  <type decls>
                        <type decl>;
<type decls>         =  <type decl>;
<type decl>          =  <simple type decl>;
<type decl>          =  <recursive types>;
<simple type decl>   =  <ID> "=" <type> ";";
```

```
<recursive types>      =  <recursive types>
                          and
                          <simple type decl>;
<recursive types>      =  <simple type decl>
                          and
                          <simple type decl>;
<type>                 =  <type id>;
<type>                 =  <enumerated type>;
<type>                 =  <subrange>;
<type>                 =  <record type>;
<type>                 =  <union type>;
<type>                 =  <set type>;
<type>                 =  <list type>;
<type id>              =  <id>;
<type id>              =  %TVAR ;
```

Example:

```
type
day-of-week = (Monday, Tuesday, Wednesday, Thursday, Friday,
            Saturday, Sunday);
date =   record
            year  :  range 0..2000;
            month :  range 1..12;
            day   :  range 1..31;
            day-name :  day-of-week;
         end record ;
```

Although type declarations of the form:

```
type A = B;
   C = @T;
```

are syntactically allowed by the above grammar, and are used (at least the first form) in languages such as Pascal to denote the declaration of synonym types, they are forbidden in OLGA and will be flagged as errors by the OLGA semantic checker. The others define brand new types as described in the next sections. Note that type names do not belong to the same name space as grammars, modules, constants or functions names (see chapter 11 for more details).

Because of the visibility rules of OLGA which impose that a name be declared before it is used (see chapter 11), and because each type must be named, the declaration of mutually recursive types must use the third form of type declaration, with the **and** keyword, which temporarily relaxes the declaration-before-use rule in a list of type declarations; this allows to declare mutually recursive types. This is exemplified in this chapter and in appendix A.

Type denoters (see chapter 8) can be used as any other type name in type definitions, provided they are visible in the scope in which the type definition appears, that is in the local declarations of a polymorphic function with explicit type expressions for the parameters. They can be involved in the declaration of any structured type.

7.3 Predefined Types

7.3.1 The type "int"

Values of type *int* are signed integers, in a range which is implementation defined. However, since most UNIX machines on which FNC-2 will run are based on 16- or 32-bits hardware, and since the OLGA translators will choose the longest integral type available in the target language to implement the type *int*, it can generally be assumed that this range will be at least $-2^{31}+1$ to $2^{31}-1$.

Operations available on *int* values are for instance binary operations such as addition, subtraction, multiplication, (Euclidian) division and remainder, the unary opposite, all usual sorts of comparisons, and a number of conversions to other types. See next chapter for more details.

7.3.2 The type "real"

Values of type *real* are implementation defined approximations of real numbers in an implementation-defined range. Operations on these values are the usual arithmetic operations, comparisons, a number of conversions to and from *ints* and *strings*, and a number of predefined mathematical functions.

7.3.3 The type "bool"

Values of type *bool* are the usual truth values `false` and `true`. Operations available on *bool* values are the binary conjunction and disjunction, the unary negation, comparisons (truth values are ordered as `false` < `true`), and conversions to other types. Binary operations exist in strict and non-strict (short-circuit evaluation) version. *Bool* values also drive the polymorphic **if** ... **then** ... **else** conditional.

7.3.4 The type "char"

Values of type *char* are characters of the ASCII alphabet. Operations available on these values are comparisons (according to the ASCII alphabet) and conversions into other types.

7.3.5 The type "string"

Values of type *string* are dynamic-length character strings. Operations available on these values are concatenation, getting the length of a string, taking a substring of a longer string, taking all of a string except for a substring, inserting a string into another one, searching a given substring in a longer string, and many others. Strings are ordered by the lexicographic ordering derived from the order of the ASCII alphabet. Strings can also be converted to other types. In some implementations of FNC-2 , the character NUL (rank 0) has a special meaning in strings (as in C); in this case, the semantics of most strings operations is undefined if one of their arguments is a string containing a NUL character. This explains why it is not possible in OLGA to write manifest strings of that kind (see chapter 6).

7.3.6 The type "token"

Values of type *token* are opaque values corresponding to lexical units of the languages described in OLGA. Only attributes of type *token* can be set at lexical and syntactic analysis time (see appendices A and B). It is guaranteed that, in a given execution of an FNC-2 generated evaluator, two lexical units have the same *token* value if and only if they have the same text after lexical analysis. The only operations available on values of type *token* are conversions to and from *strings*, and comparisons: *tokens* are totally ordered according to some otherwise unspecified order (in particular, it is NOT related to any order defined on *strings*). The type *token* has a predefined constant named `error-token`, which, in particular, plays a role in error recovery (see Part V and appendix A).

7.3.7 The type "source-index"

Values of type *source-index* represent positions in source texts of the language being described. These positions comprise in particular the name of the source file, the line number and the column number, and maybe other implementation-dependent information. The type *source-index* is an opaque type whose implementation is not specified. Each grammar symbol has the predefined attribute `$scx` of type *source-index* which represents the position in the source text of the beginning of the terminal sentence derived from it. Values of type *source-index* can be used in the **message** construct which outputs error messages (see chapter 9). Other operations available on this type are the retrieval of the name of the source file, the line number and the column number. The type *source-index* has two predefined constants named `beginning-of-text` and `end-of-text`, whose value is respectively the position before the first and after the last character of the source text.

7.3.8 The type "void"

The type *void* comprises a unique value named `null`. This type has no interest by itself, but is very useful—or even necessary—because it provides a means to terminate recursion in recursive types (see further in this chapter). This idea is borrowed from Algol-68 and ML [CHP85]. It is also possible to use *void* as the result type of some functions to make them procedures, although this seems rather useless in a strictly applicative, side-effect-less language as OLGA.

7.4 Enumerated Types

An *enumerated type* is a type defined in OLGA by listing the ordered list of its values (constants), which are represented by identifiers. This is the same as in Pascal [BSI82].

Syntax:

```
<enumerated type>     =  "(" <identifier+> ")";
<identifier +>        =  <identifier +> "," <ID>;
<identifier +>        =  <ID>;
```

Example:

```
type
day-of-week = (Monday, Tuesday, Wednesday, Thursday, Friday,
         Saturday, Sunday);
```

Operations available on enumerated types values are predecessor and successor, and comparisons: the values of an enumerated type are strictly ordered as they are listed in the type declaration (increasing order). They can also be converted to and back from *ints* as follows: the first constant in the list corresponds to integer 0 (zero), the next one to 1, and so on.

In addition to the type name and the constants, each definition of an enumerated type ET defines the following functions:

```
function int (x: ET): int raiseUNDEFINED;

function ET (i: int): ET raise RANGE-ERROR, UNDEFINED;

function string (x: ET): string raise UNDEFINED;

function ET (s: string): ET raise CONVERSION-ERROR, UNDEFINED;

function pred (x: ET): ET raise RANGE-ERROR, UNDEFINED;

function succ (x: ET): ET raise RANGE-ERROR, UNDEFINED;

function < (x, y: ET): bool raise UNDEFINED;

function > (x, y: ET): bool raise UNDEFINED;

function <= (x, y: ET): bool raise UNDEFINED;

function >= (x, y: ET): bool raise UNDEFINED;
```

For more details, see section 7.11.1 for the first two, section 7.11.2 for the next two and section 8.3 for the other ones.

The *bool* predefined type is another example of an enumerated type:

Example:

```
type
   bool = (false, true);
```

However it is predefined rather than user-defined because of its usage in some basic constructs of OLGA.

7.5 Scalar and Simple Types

The notion of *scalar types* is not a type constructor as enumerated types, but rather a means of classification. A type is a scalar type if the set of its values is homomorphic to the set of integers or a subrange thereof, which means that the notion of predecessor and successor applies to it. Scalar types in OLGA are *int*, *bool*, *char*, *void* and the user-defined enumerated types, and only those types. This notion was first introduced in Pascal.

The notion of scalar types is important for three uses:

- the notion of *subrange* of a type (see next section) is only applicable to scalar types;
- so is the **set of** type constructor (see section 7.9);
- the **case ... is** selection construction (see section 9.6) has a special form for values of a scalar type.

Although, in every implementation, *real* values are implemented as discrete (countable) values, the type *real* is NOT considered as scalar. This is because the notion of predecessor and successor does not apply to the mathematical *reals*, and thus does not apply to their computer science approximations as well.

Another notion in OLGA is that of *simple* type, i.e. types whose values are *atomic* as opposed to structured types. Simple types in OLGA are scalar types plus type *real*.

7.6 Subranges

A *subrange* type is defined as a subtype of a scalar type, named the *host type* of the subrange type, by specifying its smallest and largest values. Both of those values must belong to the same host type, and the former must be less than or equal to the latter.

Syntax:

```
<subrange>            =  range <constant expr> ".." <constant expr>;
```

Example:

```
range 1..100;            {a subrange of int}
range -10..+10;          {idem}
range '0'..'9;           {a subrange of char}
range Monday..Friday;    {a subrange of the above defined day-of-week}
```

A subrange type is not a new type in the sense of type checking: values of a subrange type are considered as values of the host type. It is only an indication that some range tests must be dynamically performed in some situations involving subrange types: assignment of a value to a subrange-typed attribute occurrence, passing a value as a subrange-typed parameter in a function call, etc. These situations are detailed where necessary in this manual.

Subranges can appear where any other type can, and also in the **test type** construct.

7.7 Records

Values of *record* types associate into a single value a finite, immutable list of values of different types; each of these component values is accessible by a *field name* independently from the others. This is the OLGA counterpart of the *cartesian product* mathematical operation, except that component values are accessed by a name rather than by their position.

Syntax:

```
<record type>        =  record
                           <fields>
                        end record;
<fields>             =  <fields>
                        <field>;
<fields>             =  <field>;
<field>              =  <identifier +> ":" <type id> ";";
```

Example:

```
type date = record
               year  :  range 0..2000;
               month :  range 1..12;
               day   :  range 1..31;
               day-of-week :  day-of-week;
            end record ;
```

The field names appearing in a record type definition must be different from one another. However a record type definition opens a local scope in which these field names are declared; this local scope applies only to field names and not to field types, so that the above declaration is quite valid. Field names are thus invisible outside the record definition, except in the field selection operation (see chapter 9: then the same local scope is reopened to search the field name.

Operations available on record values are construction, selection of a field, and equality and inequality comparisons.

Record types may not be directly recursive because there is no possibility to terminate the recursion and this would lead to infinite values which are not supported in OLGA.[1] However record types may be used in conjunction with union types and type *void* to describe recursive structures.

7.8 Unions

Union types are a means to associate several types into a single one. Values of a union type are pairs composed of a value belonging to one of those types and of a (hidden) *type tag* indicating to which type this former value belongs to. This means that values of union types

[1] Most of OLGA constructions are strict and evaluated eagerly, in particular the construction of values.

contain only one "interesting" value at a time, but different values of the same union type may have this interesting value belong to different types.

Syntax:

```
<union type>        =  union
                          <union-types>
                       end union ;
<union-types>       =  <union-types>
                       <union-type>;
<union-types>       =  <union-type>;
<union-type>        =  <type id> ";";
```

Example:

```
type
point =  record
            X, Y : int;
         end record ;

stupid =    union
               string; int; date; point;
            end union {this is a completely stupid union};
```

The types names listed in the definition of a union type must be different from one another.

Union types may be recursive, although direct recursion is totally useless. However, the combination of records, unions and the *void* type allows to describe "linked" data structures, for instance a binary tree:

Example:

```
type
binary-tree-node =   record
                        key   :  key-type;
                        info  :  info-type;
                        left-son, right-son :  binary-tree;
                     end record;
and
binary-tree =  union
                  void;
                  binary-tree-node;
               end union;
```

The only operations available on values of union types are construction (from a single value of one of the component types), equality and inequality comparisons, and the **case ... is** construct when used to test the tag.

7.9 Sets

A *set* type definition introduces a type which is the powerset of some other *scalar* type, which is named the *basic* type.

Syntax:

```
<set type>          =  set of <scalar type>;
<scalar type>       =  <type id>;
```

Example:

```
type
letter = range 'a'..'z';
letter-set = set of letter;
```

Values of such types are subsets of the set of all values of the basic type. Any set, whatever its size, can be represented, except when system-dependent limitations arise. However it is not guaranteed that their implementation be very time and/or space efficient[2]...

Operations available on set values are construction, binary operations as union, intersection and difference, comparisons (in particular the test for inclusion), and the membership test.

As the set construction operation (see section 7.11.5 below) is always tagged with a set type name which must have been declared elsewhere, the type of a set value is always precisely known, and there is no implicit set types. This allows OLGA to have a much simpler notion of operand type compatibility for set operations than Pascal [BSI82]: both operands must belong to the *same* declared set type (but see also polymorphism on sets in chapter 8).

7.10 Lists

Values of *list* types are ordered, linear lists, the elements of which belong to a same type (which may be different from or identical to the list type), named the *element* type. There is no restriction on what OLGA types may apply as a list element type.

Syntax:

```
<list type>         =  list of <type id>;
```

Example:

```
type
symbol-table-entry = record
                        key   :  key-type;
                        info  :  info-type;
```

[2]As a rule of thumb, the space used by a set value in the current implementation is proportional to the difference between the largest and the smallest elements in the set. This is most efficient when the sets are dense rather than sparse.

```
                        end record ;
symbol-table = list of symbol-table-entry;
```

List types may be recursive since there is an obvious way to terminate the recursion, namely the empty list.

Operations available on list values include construction, efficient non-destructive concatenation, length, test for emptiness, and accessing the first *(head)* and the other elements *(tail)* of a list.

7.11 Construction and Conversion Operations

The *declaration* of a type, i.e. the association of a name with a type, also declares one or more functions which have the same name as the type being declared, and which are conversion (for simple types) or construction (for structured types) operations. However the declaration of a *subrange* type does not involve the declaration of such conversion functions because values of subrange types behave in any respect as values of their host type (see section 7.6). The **test ... range** and **test ... type** constructs (see section 9.7) can be used when it is needed to obtain a value of a subrange type from one of its host type.

The type denoted by a type denoter is totally unknown, so such a type has no conversion on construction function in fact, the only operations available on values of such a type are the tests for (in)equality and (non)-identity.

Conversion operations names can be overloaded as normal functions (see chapter 8).

Although this is not specified explicitely in the declaration (for sake of clarity), every function listed in the present section raises the predefined exception `UNDEFINED` when any of its arguments is the **undef** value of the appropriate type. See section 9.9.2 for more details.

7.11.1 Scalar Types

Values of each scalar type, either predefined *(bool, char)* or user-defined (enumerations) can be converted to and back from *int* values. To convert any scalar value to an integer, use the overloaded *int* functions: for each scalar type `ST` there exists one such function declared as

```
function int (x:  ST): int;
```

returning the ordinal *rank* of value `x` in type `ST`. *Char* values have a rank equal to their rank in the ASCII alphabet. Enumeration types, including *bool*, are such that the first constant in their definition list has rank 0 (zero), the next one 1, and so on.

For each scalar type `ST` there exists a function declared as

```
function ST (x:  int):  ST raise RANGE-ERROR;
```

returning the value of type `ST` with rank `x`, provided that such a value exists. If this is not the case, the `RANGE-ERROR` predefined parameterless exception is raised during the computation of that function (see chapter 10). The expression `char(0)` is the only way in OLGA to obtain the NUL character (rank 0).

The definition of a scalar type also implies the implicit definition of the four comparison predicates `<`, `<=`, `>` and `>=` for this type. See section 8.3 (note 11) for more details.

Note: in the discussion above, `int` refers to the predefined type *int* even in scopes where `int` has been redeclared as a new type (which is bad practice in any case). The same applies to `int` and `string` in section 7.11.2 below.

7.11.2 Other Predefined Types

Values of types *int*, *char*, *real* and *token* and values of any enumerated type `ET` (including *bool*) can be converted to values of type *string* by using the following functions:

```
function string (i: int): string;

function string (i: int; base: range 2..16): string;

function string (c: char): string;

function string (r: real): string;

function string (r: real; total-width: int): string;

function string (r: real; total-width, frac-width: int): string;

function string (x: ET): string;

function string (t: token): string;
```

The first one returns the string of digits forming the decimal value of integer `i`, preceded by a minus sign if `i` is negative. The second one returns the string of digits forming the value of integer `i`, expressed in the specified *base*; just enough digits are present in the returned string. This specification is sufficient for positive numbers; what is returned for negative integers is implementation-dependent. The third one returns a string of length 1 whose only character is `c`. The next three functions produce string representations of their first parameter analogous to what is output by the respective Pascal statements [BSI82]:

```
write (r);
write (r: total-width);
write (r: total-width: frac-width );
```

Also, it is guaranteed that the produced representation is a lexically correct OLGA real number (see chapter 6); in particular, the obtained string can be passed back to the inverse function *real* described below. The *string* function applied to a value of an enumerated type returns a string containing the identifier representing that value in that type, as specified in the declaration of the enumerated type; for instance, `string (true)` returns `true`. The last function returns a string which contains the text of the lexical unit designated as `t`, as defined by lexical analysis (i.e. after all processing performed during scanning such as conversion to uppercase, removal of useless quotes, etc.: see V of this volume). This string may be the empty string; in particular, the string value associated to the predefined constant `error-token` is the empty string.

Integer values can be obtained from strings by using the two following functions:

```
function int (s: string): int
        raise CONVERSION-ERROR, RANGE-ERROR;

function int (s: string; base: range 2..16): int
        raise CONVERSION-ERROR, RANGE-ERROR;
```

These functions try to interpret the contents of `s` as an integer value, respectively in decimal notation (including an optional leading sign) or according to the specified *base*. In the second case, no sign is accepted and a positive integer is always returned; for bases greater than ten, letters `'A'` to `'F'` will be used for digits with values greater than nine, either in upper or lower case. If string `s` contains other characters than digits and, according to the specified base, those letters (and the optional sign in the first form), then the parameterless exception `CONVERSION-ERROR` will be raised; if `s` represents a legal number with respect to the base, but its value is too large to fit in the range of type *int* (as defined by the implementation), then the parameterless exception `RANGE-ERROR` will be raised.

The function declared as

```
function real (s: string): real
        raise CONVERSION-ERROR, RANGE-ERROR;
```

accepts a string representing a real number and returns the value of type *real* which most closely approximates this real number, provided that, first, the parameter is a lexically correct OLGA real number as defined in chapter 6—if this is not the case, the parameterless exception `CONVERSION-ERROR` is raised—and, secondly, the denoted value lies in the implementation-defined range of type *real*—if not so, the parameterless exception `RANGE-ERROR` is raised.

Integers can be converted to real numbers by the function declared as

```
function real (i: int): real;
```

which returns the value of type *real* closest to that of its integer parameter. No exception can be raised by this function because it is guaranteed that in every implementation the range of type *real* includes that of type *int*.

Reals can be converted back to integers by the two function declared as

```
function round (r: real): int raise RANGE-ERROR;

function trunc (r: real): int raise RANGE-ERROR;
```

`Round` returns the integral value closest to that of its parameter provided it lies in the implementation-defined range of type *int*; otherwise the parameterless exception `RANGE-ERROR` is raised. `Trunc` returns the integral part of its parameter, provided it lies in the implementation-defined range of type *int*; otherwise the parameterless exception `RANGE-ERROR` is raised.

Individual characters in strings can be accessed by the function declared as:

```
function char (s: string; x: integer): char raise RANGE-ERROR;
```

which returns the `x`-th character of string `s` if it exists, i.e. if `x` is greater than zero and less than or equal to the length of `s`. Otherwise the `RANGE-ERROR` parameterless exception is raised. The first character of a string is numbered 1 (one).

Values of any enumerated type `ET` can be obtained from strings by using the function declared as:

```
function ET (s: string): ET raise CONVERSION-ERROR;
```

If string `s` contains an identifier naming one of the constants of type `ET`, as specified in the declaration of this type, then the function returns this value; otherwise the parameterless exception `CONVERSION-ERROR` will be raised.

Values of type *string* can be converted to values of type *token* by using the two functions declared as:

```
function token (s: string; check: bool): token;

function token (s: string): token;
```

The semantics of the first function is as follows: if `check` is false this function returns the unique value of type *token* whose text is `s`; if that token did not exist before then it is created; if `check` is true the function returns the unique value of type *token* whose text is `s`, unless that token does not yet exist, in which case it returns the predefined constant `error-token`. The semantics of the second function is the same as the first with argument `check` being equal to false.

Values of types *source-index* and *void* cannot be converted to or from values of any other type, and no function with these names are predefined.

7.11.3 Records

For each record type `RT` with field list `F1 : T1, F2: T2, ..., Fn: Tn` there exists a function declared as:

```
function RT (F1: T1; F2: T2; ...; Fn: Tn): RT;
```

which returns a record value of type `RT` whose field values are those passed as arguments. See next subsection on unions for an example.

7.11.4 Unions

For each union type `UT` with component types `T1, T2, ..., Tn` there exist n functions declared as:

```
function UT (x: T1): UT;
function UT (x: T2): UT;
      ...
function UT (x: Tn): UT;
```

which take a value `x` of one of the component types `Ti` and return a value of the union type `UT` with type tag `Ti` and value `x`. For instance, given the declaration of type `binary-tree` as in section 7.8 above, a "leaf" of such a binary tree could be constructed with the following expression:

Example:

```
binary-tree (binary-tree-node (a-key, an-info,
         binary-tree (null), binary-tree (null)))
```

in which the first `binary-tree` function application is an instance of

```
function binary-tree (x: binary-tree-node): binary-tree;
```

and the others are instances of

```
function binary-tree (x: void): binary-tree;
```

7.11.5 Sets

The construction of values of *set* types is not performed by functions applications, but rather by using a special OLGA construct derived from the set constructor of Pascal [BSI82], the main difference being that a set constructor is always tagged with its type name. The rationale for that is the desire to provide a constructor with a variable number of arguments, some of them being subranges; since both of these features do not fit in the function definition formalism used in OLGA, we preferred to introduce a special construct.

Syntax:

```
<set constructor>     =  <set type name> "[|" <set elt *> "|]";
<set type name>       =  <type id>;
<set elt *>           =  ;
<set elt *>           =  <set elt +>;
<set elt +>           =  <set elt +> "," <set elt>;
<set elt +>           =  <set elt>;
<set elt>             =  <expression>;
<set elt>             =  <expression> ".." <expression>;
```

Example: (assuming that `char-set` has been declared as **set of** `char`):

```
char-set [|'a'..'z', 'A'..'Z', '_'|]
```

A set constructor specifying no set element returns the empty set belonging to the given set type. A set constructor specifying a unique set element which is a single expression returns the value of the given set type which is the singleton containing the value of the expression; that value must belong to the basic type of the given set type, and must verify all necessary range constraints. If this is not so, the `RANGE-ERROR` parameterless exception is raised. A set constructor specifying a unique set element which is a range specified by two expressions returns the value of the given set type which contains all the values of the basic type which are greater than or equal to the lower bound of the range (first expression) and lower than or equal to its upper bound (second expression); if the lower bound proves (at run time) to be greater than the upper bound, then the empty set is returned. Both bounds values must belong to the basic type of the given set type, and must verify all necessary range constraints (which implies that all values in the specified subrange also verify those constraints). If this is not so, the `RANGE-ERROR` exception is raised. A set constructor specifying several set elements returns the union of the sets which would have been constructed for each set element taken separately.

For each set type `ST` with basic type `BT`, and assuming that `HT` is the host type of `BT`, there also exists a polymorphic conversion function declared as:

function `ST (s:` **set of** `HT): ST` **raise** `RANGE-ERROR;`

which takes as argument a value `s` of any set type based on `HT` and which returns the *same* value but with type `ST`, provided that `s` contains only elements in the range defined by `BT`. If this is not so, the `RANGE-ERROR` exception is raised.

7.11.6 Lists

For each list type `LT` with element type `T`, there exists four functions declared as:

function `LT (): LT;`

function `LT (x: T): LT;`

function `LT (x: T; l: LT): LT;`

function `LT (l: LT; x: T): LT;`

The first one returns a list of type `LT` with no elements in it, while the second one returns a list of type `LT` containing one element with type `T` and value `x`. The third one adds a new element `x` in front of the list `l` and returns the new list; the fourth one adds a new element `x` at the end of the list `l` and returns the new list; in both cases the original list `l` is not modified. Longer lists may be constructed by concatenation.

There is also a special list constructor:

Syntax:

```
<list constructor>    =  <list type name> "[" <actual *> "]";
<list type name>      =  <type id>;
```

such that with the above conventions:

LT [x_1, x_2, ..., x_n] (n $\geq$ 0)

is equivalent to

LT (x_1, LT (x_2, ... LT(x_n, LT())...))

i.e. it constructs a list of type LT containing the element x_1, x_2, ..., x_n. Each x_i must of course be of type T.

Chapter 8

Functions and Type Checking

8.1 Function Definitions, Overloading, Polymorphism and Type Inference

New functions may be defined in OLGA. Such a definition involves the specification of the function name, of the number, names and optionally types of its parameters, of optionally the type of its result, of optionally the list of exceptions which can arise during its computation, and of the expression which will be used to compute the result. It is also possible to declare local objects such as constants, types and functions (see chapter 11).

Syntax:

```
<function def>         =  <simple function def>;
<function def>         =  <recursive functions>;
<recursive functions> =     <recursive functions>
                          and
                             <simple function def>;
<recursive functions> =     <simple function def>
                          and
                             <simple function def>;
<simple function def> =  <function header> is
                             <function body> ";";
<function header>      =  <inline> function <function id>
                             <formal parameters>
                             <type constraint> <exc +>;
<function header>      =  <inline> function <function id>
                             <formal parameters>
                             <type constraint>;
<inline>               =  inline ;
<inline>               =  ;
<function id>          =  %ID ;
<function id>          =  <operator>;
<operator>             =  "+";
<operator>             =  "-";
<operator>             =  "*";
```

```
<operator>            =  "/";
<operator>            =  "%";
<operator>            =  "=";
<operator>            =  "!=";
<operator>            =  "~";
<operator>            =  "!~";
<operator>            =  "<";
<operator>            =  ">";
<operator>            =  "<=";
<operator>            =  ">=";
<operator>            =  "&";
<operator>            =  "|";
<operator>            =  "!";
<formal parameters>   =  "(" <param *> ")";
<param *>             =  ;
<param *>             =  <param +>;
<param +>             =  <param +> ";" <param group>;
<param +>             =  <param group>;
<param group>         =  <identifier +> ":" <type expr>;
<param group>         =  <ID>;
<type constraint>     =  ;
<type constraint>     =  ":" <type expr>;
<type expr>           =  <type id>;
<type expr>           =  list of <type expr>;
<type expr>           =  set of <type expr>;
<function body>       =     <top expression>
                         end function ;
<function body>       =  declare
                            <decl +>
                         in
                            <top expression>
                         end function ;
<function body>       =  external ;
<function body>       =  external <ID>;
<top expression>      =  <expression>;
<exc +>               =  raise <exc id>;
<exc +>               =  <exc +> "," <exc id>;
<exc id>              =  <id>;
```

When the header of a function definition begins with the keyword **inline**, it is an indication that the OLGA compiler should try to expand the function body at each of its call points rather than generate a standard function call. However this is only an indication (a *pragma* in the sense of Ada [ANS83]) which the compiler may ignore at will; conversely it may decide to expand in line some functions which were not so qualified. Most of the predefined functions are declared **inline**.

Function names may be *overloaded* as in Ada, which means that the same name may designate several different functions, provided that one can infer unambiguously from the context which function is intended. More precisely, two functions with the same name cannot be visible together at the same point if they have the same *profile*, i.e. if they have the same parameter number and types and result type. Functions which are implicitely defined as a side effect of the declaration of a type (see chapter 7) can be overloaded as user-defined ones.

Function names can be OLGA identifiers or operators, as listed above. In the former case, the corresponding functions will be designated as *normal functions* and will be called as described in next section; they may have any number of parameters. In the latter case, the corresponding functions will be designated as *operator functions* and their calls must appear in an OLGA program as the application of an infix (for binary operators) or prefix (for unary ones) operator, which means that the number of their parameters and their syntactic priority level and associativity rules are fixed (see chapter 9 for more details); all operators except `!` are binary, and `!` and `+` and `-` are (also) unary.

The formal parameter list defines the number, names and (optionally) types of the parameters. The specification of a parameter group with more than one name:

$\mathtt{id}_1$`,` $\mathtt{id}_2$`, ...,` $\mathtt{id}_n$`: ty`

is equivalent to the sublist:

$\mathtt{id}_1$`: ty;` $\mathtt{id}_2$`: ty; ...;` $\mathtt{id}_n$`: ty`

If `ty` is a type expression (see below) involving type variables (type denoters in this case), the *same* variables are used in each occurrence of `ty` in the second form, so that all $\mathtt{id}_i$ actually have the same type.

A new scope is opened for the body of the function, in which the names and (optionally) types of the parameters are visible; inside this expression, the parameters behave as declared values (see chapters 9 and 11) computed prior to the call. The description of parameter passing is detailed in next section. If the function body contains local declarations—types, constants, values, exceptions and functions definitions are allowed—, those declarations are added to that scope. See chapter 11 for more information. The function itself is declared in the enclosing scope and is thus visible in the function body (unless redeclared), so that it may be called recursively.

If a parameter group has a type constraint, then this must be a *type expression* (see below); however the type variables occurring in such an explicit type term are special—hence their name of *type denoters*—in the sense that they cannot be bound during the type inference process (see below); in fact they denote opaque types. Otherwise, that is when a parameter group is reduced to a single name with no type constraint, a fresh *type variable* is associated to that name (it is a normal one, i.e. it can be bound as described below) and *type inference* [Mil78, Car87] is then performed on the function body, using *unification* [Rob65, MM82], to determine for each of those variables the *most general type* compatible with their use in the body. Some of these types may be *polymorphic* types [Mil78, Car87], in which case the function is known as a polymorphic function. If every parameter group has an explicit type constraint, then type inference reduces to mere type checking, because none of the type variables appearing in the type expressions may be bound; however the process is essentially the same.

The same applies to the result of the function, except that if an explicit type expression involving variables is given, then these variables **MUST** also appear in some of the type expressions for the parameters. Otherwise, a new type variable is associated to that result, and participates in the type inference performed on the function body.

Type inference relies on the unification of three distinct type kinds:

- *actual types* which are declared types and have a name; those types are either predefined or user-defined;
- type variables and type denoters described above;
- *polymorphic type constructors*: **list of** and **set of**.

Those type kinds can be combined to form *type expressions* in the following way:

- an actual type is a type expression;
- a type variable is a type expression;
- a type denoter is a type expression;
- if "TT" is a type expression, then "**list of** TT" and "**set of** TT" are type expressions;
- no other expression is a type expression.

When all the function names in an expression are not overloaded, i.e. they designate only one function, type inference proceeds by first determining the set of all type variables in that expression. These variables may be associated to typeless parameters and/or typeless result if the expression is a function body, or may be associated to polymorphic type constructors of parameters and result of applied functions. Then the expression is traversed bottom-up to determine *(infer)* by unification the type of each of its components. Some of those components have a type which is known a priori: typed parameters, manifest and declared constants, etc. The type of other components (e.g. function calls) must be inferred from the context. If a given unification trial fails, then the *type checking* of the whole expression fails and type inference cannot proceed. If a unification trial succeeds, it will in some cases *bind* a type variable to some actual type; this binding is recorded globally for the whole expression and may be used in further unification trials. See [Car87] for more details.

Overloading somewhat complicates this process. To each of the overloaded function names is attached the *set* of their possible profiles; if some of these are polymorphic, then different type variables are used for each of them. The above type inference mechanism is then modified as follows: for each overloaded function only the profiles which match the types of the actual parameters, i.e. those which have the same number of parameters and such that the types of the formal and actual parameters unify, are kept in the set of profiles. This may imply that in the bottom-up traversal of the expression, some function calls may have a set of types rather than a single type; if such a set becomes empty after matching the parameters, then the expression is badly typed. Matching a set of profiles with actual parameters having a set of types reduces to finding all the combinations where types unify. However the whole expression forming the function body should be a *top-level expression*, that is an expression which should have one and only one type. If, at the end of the bottom-up traversal, the whole expression has an empty set of types, then it is badly typed. If it has more than one type, then the expression contains unresolved overloaded references; type constraints should be added to provide more information and help make the choice. If it has one and only one type, then the expression is traversed top-down to supply context information, so that the set of types for each subexpression can be reduced to one type and overloaded references be resolved. This mechanism is a straightforward adaptation of the well-known overloading resolution algorithm [PDM80] initially developed for Ada; the only difference is in the matching of two types, where equality is replaced by unification. See also the language Hope [BMS80] for a combination of overloading and polymorphism.

Note that the same type variable may appear several times in a given function profile: for instance the predefined function *head* has one parameter of type "**list of** @T" and a result of type "@T". This means that the type of the parameter and the type of the result are *bound* through that variable: for instance, in some context, the *head* function could be called with an actual parameter of type "**list of** string"; this would bind the variable @T to "string" and the result type would then be "string".

Unification rules are commutativity and the following rules.

- Two actual types unify only if they are the same type with the same name *(name equivalence)*.
- Any attempt to unify a type denoter appearing in an explicit type expression for some parameter with any type expression other than itself will fail; this behavior is similar to that of "opaque" actual types (see section 11.3.3).
- An actual type and a (yet unbound) type variable unify; the variable becomes bound to the type name.
- More generally, an unbound type variable and a type expression unify; this binds the variable to the type expression.
- Two (different) type variables unify: they become the same variable.
- A non-opaque actual type and a type constructor with a (yet unbound) variable (e.g. **list of** @T) unify only if the actual type is constructed with the same type constructor; in that case the variable is bound to the argument of the constructor according to the declaration of the actual type; for instance, if one tries to unify the actual type "list-of-strings" which has been declared as "**list of** string" with "**list of** @T" where @T is an unbound type variable, then the unification succeeds and binds @T to "string"; however the unification of the same actual type "list-of-string" with "**set of** @T2" fails. Note that this means that unification with such a type expression uses the *structure* of an actual type, and not only its name. The **set of** constructor is rather special in that unifying the type expression "**set of** @T" with an actual type ST declared as "**set of** BT" does not bind the variable @T to BT but rather to the host type of BT; note that unifying "**set of** @T" with anything will fail if this would lead to binding @T to a non-scalar type. A type denoter appearing in an explicit type expression as argument of a **set of** constructor is assumed to be scalar, but this does not mean that anything more is known on the type it denotes. In particular the functions which are available for every scalar type (e.g. `pred` or `<`) are **NOT** available for such a type denoter.

- An actual type and a less general type expression such as "**list of set of** @T" unify only if the actual type is constructed with the same type constructor and if, recursively, the arguments of the type constructor on both sides unify.
- An actual set type ST with basic type BT and a ground (i.e. variable-less) type expression "**set of** TT" unify if the *host type* of BT (rather than BT itself) and TT unify.
- Two "constructor" type expressions unify only if they are headed by the same type constructor and, recursively, if their arguments unify.
- A bound type variable and a type expression unify if and only if the value of the variable (i.e. the type expression to which the variable is bound) and the former type expression unify.

If the type inference mechanism succeeds, all subexpressions in the function body have a type; this type is either an actual type or a type expression involving variables. This is especially true of the function parameters and result; this type information is recorded as the profile of the function. If any of the types appearing in the profile of a function is a type expression with variables or a "**set of** HT" expression, then the function is polymorphic.

If a function with the same name and the same profile—up to a consistent renaming of the type variables, if any—as already been declared in the same block, then the new definition is a double definition error.

Type denoters which appear in explicit type expressions in the function header (for parameters) are visible in the function body and can be used to declare local types, local values, etc. This is described in more details in chapters 7 and 11.

The differences between an (explicit) type denoter and an (implicit) type variable apply only in the *definition* of a function and not when type-checking the *application* of the function.

In all issues regarding type inference and type checking, actual types which are subranges behave as their host types. The only difference is that some range tests are performed at run-time in some places. In particular, if the result of a function is declared to be of a subrange type, the function body is type checked (at compile time) and computed (at run time) as if it belonged to the host type; only when the actual result has been computed is a range check performed on it to check whether it fits in the specified subrange. Similarly, if a formal parameter has a subrange type, unifying an actual parameter with it is done using the host type; at run time a range check will be performed on the value of the actual parameter to check whether it fits in the specified subrange.

For the same reason as for mutually recursive types, mutually recursive functions must be defined together using the **and** connector. Type inference is then performed at the same time on all the function definitions appearing in the same group.

The header of a function definition should also list the names of the exceptions which are possibly raised during its computation. Presently this is not really mandatory (except for functions passed as parameters to some parameterized module, see section 11.4.2) and has only useful documentary purposes, but in future releases it may happen that this becomes mandatory because it will be used by the OLGA compiler to perform sophisticated consistency analysis. So please take up this good habit...

Functions declared to be **external** are described in section 8.5 below. Just note here that external functions must have explicit type constraints for all of their parameters and for their results, and that the corresponding type expressions must be reduced to actual type names; this implies that these functions cannot be polymorphic.

8.2 Function Calls and Parameter Passing

Normal function calls are specified with a quite common syntax. The syntax of operator function calls is described in next chapter.

Syntax:

```
<function call>       =  <id> "("<actual *> ")";
<actual *>            =  ;
<actual *>            =  <actual +>;
<actual +>            =  <actual +> "," <actual>;
<actual +>            =  <actual>;
<actual>              =  <expression>;
```

The type checking performed to match actual and formal parameters of a single function has been described above: first, the number of parameters is checked, and then the type of each actual parameter is unified with that of the corresponding formal parameter. As said before, when type checking a call to a polymorphic function which was declared with explicit type expressions and explicit type denoters, then the latter behave as normal type variables, i.e. they may be bound. If the function name is overloaded, this process is repeated for each profile in the set of visible profiles, to determine the set of profiles which match and the corresponding set of return types; these sets are later reduced to one member during the top-down pass. Subrange typed parameters are also described above.

All parameters are passed by value this is obvious since the concept of a (modifiable) variable does not exist in a purely applicative language such as OLGA. The order in which the expressions constituting the actual parameters are computed is unspecified. Once all the parameters have been computed, control is transferred to the function and a new environment is set up in which the names of the formal parameters behave as declared values whose values are those of the actual parameters; explicit type denoters in the function profile are also visible in this environment, and their value is determined for each by the above type checking mechanism; the optional local declarations and the expression which constitutes the function body are evaluated in that environment. The value and type of the expression are then the value and type of the whole call.

8.3 Predefined Functions

8.3.1 Predefined functions and operators

Apart from the conversion and construction functions described in chapter 8, the following functions and operators are predefined in OLGA, where `@T` is a type variable (but see notes 10 and 11):

Name	Type of first or only operand	Type of second operand (if any)	Result Type	May raise[1]	Description	Notes
+	int	int	int	RANGE-ERROR	addition	1
+	real	real	real	RANGE-ERROR	addition	1
+	string	string	string		concatenation	
+	**list of** @T	**list of** @T	**list of** @T		concatenation	
+	**set of** @T	**set of** @T	**set of** @T		union	
+	int		int		identity	
+	real		real		identity	
-	int	int	int	RANGE-ERROR	subtraction	1
-	real	real	real	RANGE-ERROR	subtraction	1
-	**set of** @T	**set of** @T	**set of** @T		set difference	3
-	int		int		sign inversion	
-	real		real		sign inversion	
-	**set of** @T	**set of** @T			complementation	
*	int	int	int	RANGE-ERROR	multiplication	1
*	real	real	real	RANGE-ERROR	multiplication	1
*	**set of** @T	**set of** @T	**set of** @T		set intersection	
*	@T	**set of** @T	bool		membership test	
/	int	int	int	ZERO-DIVIDE	integral division	2
/	real	real	real	ZERO-DIVIDE	real division	2
%	int	int	int	ZERO-DIVIDE	remainder/modulus	2
=	@T	@T	bool		equality	10
!=	@T	@T	bool		inequality	10
~	@T	@T	bool		identity test	10
!~	@T	@T	bool		non-identity test	10
<	(scalar) ST	ST	bool		ordering test	11
<	real	real	bool		ordering test	
<	string	string	bool		ordering test	
<	token	token	bool		ordering test	
<	**set of** @T	**set of** @T	bool		set inclusion test	6
>	(scalar) ST	ST	bool		ordering test	11
>	real	real	bool		ordering test	
>	string	string	bool		ordering test	
>	token	token	bool		ordering test	
>	**set of** @T	**set of** @T	bool		set inclusion test	6
<=	(scalar) ST	ST	bool		ordering test	11
<=	real	real	bool		ordering test	
<=	string	string	bool		ordering test	
<=	token	token	bool		ordering test	
<=	**set of** @T	**set of** @T	bool		set inclusion test	6
>=	(scalar) ST	ST	bool		ordering test	11
>=	real	real	bool		ordering test	

[1] Every predefined function listed in this section raises, in addition to the listed exceptions, the predefined exception UNDEFINED if any of its operands is the **undef** value of the appropriate type (see section 9.9.2). The only exceptions to this rule are the (in)equality and (non-)identity predicates, and the functions listed in note 14.

>=	string	string	bool		ordering test	
>=	token	token	bool		ordering test	
>=	**set of** @T	**set of** @T	bool		set inclusion test	6
&	bool	bool	bool		conjunction (and)	4
&	int	int	int		bitwise AND	5
|	bool	bool	bool		disjunction (or)	4
|	int	int	int		bitwise OR	5
!	bool		bool		negation	
!	int		int		bitwise complement	5
pred	(scalar) ST		ST	RANGE-ERROR	predecessor	7
succ	(scalar) ST		ST	RANGE-ERROR	successor	7
round	real		int	RANGE-ERROR	rounding	12
trunc	real		int	RANGE-ERROR	integral part	12
cons	@T	**list of** @T	**list of** @T		list cons. (front)	14
tcons[2]	**list of** @T	@T	**list of** @T		list cons. (end)	14
ncons	@T		**list of** @T		one element cons.	14
empty	**list of** @T		bool		emptiness test	
empty	**set of** @T		bool		emptiness test	
empty	string		bool		emptiness test	
head	**list of** @T		@T	EMPTY-LIST	first element of a list	8
tail	**list of** @T		**list of** @T	EMPTY-LIST	all elements but first	8
length	**list of** @T		int		number of elements	
card	**set of** @T		int		number of elements	
length	string		int		number of characters	
length	token		int		number of chars	13
file	source-index		string		source text name	9
line	source-index		int		source line number	9
column	source-index		int		source column	9

Notes:

1. The RANGE-ERROR exception is raised when the result of these operations does not lie in the implementation-defined range for types *int* or *real*; it is also implementation-defined whether the exception is actually raised or not, and what the result is when the exception should be raised and is not (in particular when the target language is C, since in that language it is not possible to check overflow on arithmetic operations).

2. The ZERO-DIVIDE exception is raised when the second argument (divider) is zero. Whether the % operation is the modulus or the remainder, and what its result is if either of its operands is negative, is implementation-dependent.

3. A - B contains all elements of A which are not elements of B, and no other.

4. The evaluation of these operations is strict; for a non-strict (short circuit) version, use A **and** B and A **or** B (see chapter 9).

5. The result of these operations is implementation-defined, however it is what could be expected according to their names.

[2] The name means "tail cons", not "tagged cons" as in Le_Lisp.

6. `A < B` is true iff all elements of `A` are also elements of `B`, but there is also at least one element of `B` which is not in `A`; `A <= B` is true iff all elements of `A` are also elements of `B`; `A > B` is equivalent to `B < A`, and `A >= B` is equivalent to `B <= A`.

7. `pred (x)` returns the element of `ST` with rank one less than that of `x`, except if `x` has lowest rank in `ST` (zero for enumeration types), in which case the `RANGE-ERROR` exception is raised; `succ (x)` returns the element of `ST` with rank one more than that of `x`, except if `x` has highest rank in `ST`, in which case the `RANGE-ERROR` exception is raised. For type *int*, `pred` can be defined as:

```
function pred(x: int): int raise UNDEFINED, RANGE-ERROR is
   x - 1
end function ;
```

   and if `T` denotes type *char* or an enumerated type, as:

```
function pred (x: T): T raise UNDEFINED, RANGE-ERROR is
   T(int(x) - 1)
end function ;
```

   The same applies to `succ`, plus being used instead of minus. These functions are overloaded, i.e. there is one pair of such functions for each scalar type, and not polymorphic (see section 7.11.1).

8. The `EMPTY-LIST` exception is raised when the argument is an empty list.

9. See section 7.6 for an explanation of those functions.

10. The `=` and `!=` functions test the *structural equality* of two values—that is, roughly speaking, whether they would have the same printable representation, if such a notion existed in OLGA—, whereas the ~ and `!`~ functions test the *identity* of two values, that is whether they are physically the same object. The difference between equality and identity is implementation-defined, but it is guaranteed that:

    - identity implies equality;
    - identity and equality are equivalent for values of simple types (see section 7.5).

    As the above specification suggests, all these functions are polymorphic.

    The equality and inequality functions are **NOT** guaranteed to terminate for recursive types. It is always possible to write a terminating equality test for a given recursive type using the identity test and the destructing constructs. Weaker forms of equality may also be implemented as normal functions.

11. Unlike the (in)equality and (non-)identity predicates, these functions are *not* polymorphic; rather, they are overloaded, and one such function is (implicitly) defined for each (scalar) type, in the same way that construction or conversion functions are implicitly defined when a type is defined.

12. This is a recall of section 7.11.2, since these conversion functions do not have the same name as their result type.

13. `length(x)`, where `x` is of type token, is the same as `length(string(x))` but is computed more efficiently.

14. These functions **DO NOT** raise the `UNDEFINED` exception when their "element" argument —first for `cons`, second for `tcons` and only one for `ncons`—is **undef**, so that it is possible to have **undef** elements in lists; this means that `ncons` never raises `UNDEFINED`. The other two raise `UNDEFINED` if their "list" argument is **undef**.

These functions are summarized in Appendix C.

8.3.2 The *string-functions* Module

The *string-functions* module contains several mathematical functions operating on *strings*. Since these functions are not expected to be used very often, they are defined in a separate module; see chapter 11 for more informations on modules. The *string-functions* DCM is as follows:

Other predefined functions dealing with *strings* not listed above are as follows:

```
declaration module string-functions is
   function substr (s: string; start: int; lgt: int):string
            raise RANGE-ERROR, UNDEFINED ;
   function substr (s: string; start: int): string
            raise RANGE-ERROR, UNDEFINED ;

   function omit (s: string; start: int; lgt: int):string
            raise RANGE-ERROR, UNDEFINED ;
   function omit (s: string; start: int): string
            raise RANGE-ERROR, UNDEFINED ;

   function insert (s1, s2: string; start: int): string
            raise RANGE-ERROR, UNDEFINED ;

   function index (s1, s2: string): int
            raise UNDEFINED ;
   function index (s: string; c: char): int
            raise UNDEFINED ;

   function search (s: string; chars: set of char):int
            raise UNDEFINED ;
   function verify (s: string; chars: set of char):int
            raise UNDEFINED ;
end module ;
```

As every other predefined function, the functions in *string-functions* raise the `UNDEFINED` exception if any of their operands is the **undef** value of the appropriate type.

The `substr` functions return substrings of their first argument `s`, starting with the character with rank `start` (where the first character has rank one). Both of them raise the `RANGE-ERROR` exception if `start` is less than one or greater than `length (s)`. The first function returns a string with `lgt` characters if `start + lgt - 1` $\leq$ `length (s)`, that is if there exist enough characters in `s`, else it raises the `RANGE-ERROR` exception. The second one returns a string with all characters of `s` from `start` up to `length (s)` (inclusive).

The `omit` functions return the string derived from their first argument `s` by omitting the substring defined as above by `start` and, if present, `lgt`, with the `RANGE-ERROR` exception being raised in the same conditions as above. The resulting string may be empty.

The `insert` function returns the string derived from its first argument `s1` by inserting the string `s2` at rank `start`, i.e. the first character of `s2` will be the `start`-th character of the result. The `RANGE-ERROR` exception is raised if `start` is less than one or greater than `length (s1)`.

The `index` functions return the rank of the first occurrence of their second argument (either a string or a single character) in their first one (a string) if such an occurrence exists, else zero.

The `search` function returns the rank of the first occurrence in `s` of any character belonging to `chars`, or zero if no character of `chars` appears in `s`. The result is zero if `s` is empty. The `verify` function returns the rank of the first occurrence in `s` of any character *not* belonging to `chars`, or zero if all characters of `s` belong to `chars`. The result is zero if `s` is empty.

The corresponding DFM does not exist in OLGA but only in compiled form.

8.3.3 The *math-functions* Module

The *math-functions* module, which is inspired of the predefined functions in Pascal [BSI82], contains several mathematical functions operating on *reals*. Since these functions are not expected to be used very often, they are defined in a separate module; see chapter 11 for more informations on modules. The *math-functions* DCM is as follows:

```
declaration module math-functions is

    const
       e, pi:  real;

    function abs (x:  real):  real; {absolute value }
    function sqr (x:  real):  real raise RANGE-ERROR ;
          { square; RANGE-ERROR is raised when result is
             out of the range of type real }
    function sqrt (x:  real):  real raise RANGE-ERROR ;
          { square root; RANGE-ERROR is raised when x < 0 }
    function sin (x:  real):  real; {sine; x in radians }
    function cos (x:  real):  real; {cosine; x in radians }
    function arctan (x:  real):  real raise RANGE-ERROR ;
          { principal value, in radians, of x's tangent arc;
             RANGE-ERROR is raised when result is
             out of range }
    function exp (x:  real):  real raise RANGE-ERROR ;
```

```
        { exponential; RANGE-ERROR is raised when result is
           out of range }
    function ln (x:  real):  real raise RANGE-ERROR ;
        { natural logarithm; RANGE-ERROR raised when x ≤ 0
           or when result is out of range }
end module ;
```

The corresponding DFM does not exist in OLGA but only in compiled form, and is implementation-dependent.

8.4 Examples

The following definition:

```
function plus (x, y) raise RANGE-ERROR is
    x + y;
end function ;
```

is illegal since the top-level expression `x + y` has, in the absence of type constraints on `x` and `y`, and because `+` is overloaded, more than one type. However, given an environment in which only the predefined functions are visible, the following definitions:

```
function plus (x, y:  int) raise RANGE-ERROR is
    x + y;
end function ;
function plus2 (x) raise RANGE-ERROR is
    x + 2;
end function ;
```

are well typed: the first one has profile `(int; int): int` and is a synonym of the integer addition, whereas the second one has profile `(int): int` because `2` is a manifest constant of type `int` and there exists only one function `+` with second parameter of type `int`.

In the same environment, the following definition:

```
function length (l) is
    if empty (l) then 0
    else 1 + length (tail (l));
end function ;
```

is well typed, has polymorphic profile `(`**list of** `@T): int` and is a synonym of the predefined function `length` computing the length of a list. The type inference mechanism proceeds as follows, where the `@Ti` denote type variables:

1. `length` is given profile `(@T1): @T2`, where `@T1` and `@T2` are new type variables; `l` is thus given type `@T1`;

2. `empty` has profile **(list of** `@T3`**) :** `bool`; `@T1` is unified with **list of** `@T3`, which succeeds and produces the binding `@T1` $\equiv$ **list of** `@T3`;

3. `0` and `1` are manifest constants of type `int`;

4. `tail` has profile **(list of** `@T4`**) : list of** `@T4`; `@T1` and **list of** `@T3` are unified with **list of** `@T4`, which succeeds and replaces `@T4` by `@T3`; this unification is actually performed twice: once to check the parameter of `tail`, and once to check the parameter of `length`;

5. `+` is overloaded, however it has only one profile with first parameter of type `int`, namely `(int; int): int`; then `@T2`, the result type of `length`, is unified with `int`, the type of the second parameter of `+`, which succeeds and produces the binding `@T2` $\equiv$ `int`;

6. now the **if ... then ... else** construct is type checked: `bool`, the result type of `empty`, is unified with `bool`, the expected type of the condition; this of course succeeds; then the type of the true part, `int` (that of the constant `0`), is unified with the type of the false part, `int` (that of the `+`), which also succeeds; the whole expression has thus type `int`; this type is unambiguous, which is necessary for a top level expression;

7. then that latter type `int` is again unified with the result type of `length`, `@T2`, which succeeds since `@T2` was already bound to `int`;

8. combining the results of all those unifications gives the final profile of `length`, **(list of** `@T3`**) :** `int`.

Note that, in our specification, *length* does not raise the `RANGE-ERROR` exception, because we assume that no one will construct so long lists that their length is out of the range of type *int*. It does not raise either the `EMPTY-LIST` exception which could arise in the computation of *tail* because we know that we don't apply the latter to empty lists.

8.5 Connection with External Languages

8.5.1 Introduction

A purely applicative language such as OLGA completely prevents a computation to make side-effects and forbids the use of global variables. This feature is very valuable for attribute grammars specification because it fits perfectly well with the notion of a declarative specification, and it provides the evaluator generator with complete control over the way it will organize the individual computations and offers many opportunities for optimizations. However, in some rare cases, this formalism is so restrictive that the expression of some notions is rather awkward. A good example is the generation of fresh labels when generating assembly code: the standard way to achieve this is to maintain a global counter specifying the value of the next label to be generated, and to increment it when such a label is requested during the code generation phase. Since global modifiable values are absent from pure attribute grammars, the implementation of this counter could be a compound attribute to be propagated throughout the whole tree and updated on the nodes where a new label is required. This is rather cumbersome to write, and, even with the powerful space optimizations provided by the FNC-2 evaluator generator (see *the* FNC-2 *Reference Manual*), would probably be far less efficient than the standard implementation using a global variable.

To overcome this kind of problems, OLGA provides a way to interface an attribute grammar with external functions written in other languages. This section describes only the linguistic and semantic aspects of that interface, viewed from the OLGA side. Implementation aspects, in particular those depending on the target language, are discussed in the relevant chapters of *the* FNC-2 *Reference Manual*.

8.5.2 Syntax and Semantics

On the OLGA side, this interface is achieved by the declaration of external functions, as defined in section 8.1.

A function definition whose body is reduced to the word **external** is the declaration of an external function; such a declaration specifies the OLGA name, the external name and the types of the parameters and result of an external function.

The OLGA name is the one following the keyword **function**, as expected. The external name is the one following the keyword **external** or, if none is present, the same as the OLGA name; however this facility is not applicable to operator functions. This external name will be used in the generated evaluator or module to call the external function. The OLGA name can be overloaded if overloading rules are respected (see section 8.1 above). Whether the external name can be overloaded is not an OLGA issue the grammar writer must ensure himself on the target language side that the external name unambiguously designates the desired function. However it is guaranteed that names generated by the OLGA compiler to designate in the target language the implementations of OLGA objects will be different from the names of declared external functions.

All the formal parameters and the result of an external function must have explicit actual types; in particular they cannot be polymorphic. Furthermore only a limited number of types are usable in communicating with external functions: all the simple types, including *void* and user-defined enumerated types, *token* and *string*. The mapping of those OLGA types to types of the target language is described in the relevant chapters of *the* FNC-2 *Reference Manual*. External *procedures* can be declared as functions with a return type of *void*, but they may be rather difficult to integrate in an applicative specification... How external functions may raise and catch exceptions is also described in *the* FNC-2 *Reference Manual*.

The declaration of an external function may appear in any block (see chapter 11). However OLGA scope rules do not apply to external names, which are global to an OLGA specification. The writer must ensure that different external names are used for different external functions, even if they are used in disjoint blocks. The same external function may have different OLGA names in different blocks; conversely, different external functions may have the same OLGA name in different blocks.

The **inline** specification will always be ignored for external functions, for the obvious reason that the OLGA compiler does not know their body.

The semantics of a call to an external function is similar to that of OLGA function calls: the actual parameters are evaluated in an unspecified order, fresh copies of them are created and converted to target data types if necessary, then the external function is called in a target language-dependent manner with those copies as parameters. Thus the parameters are passed by value, even if the external function modifies some of them: it will modify only the copies. Note that those copies are not created when the OLGA compiler can determine that the function cannot modify them, e.g. if the target language implements only parameters passed by value, such as C. The result of that external call, if any, is then converted back to the OLGA result type and is the value of the call.

If an external function performs side-effects —most of them will normally do so, since OLGA is better suited to write pure functions—, it falls entirely on the writer to ensure that those side-effects may be safely repeated several times since it is not guaranteed that a given function call will be executed only once; this is especially true in incremental evaluators. Similarly, since the order of the computation of the parameters of a function call and the order of evaluation of the attributes is not specified in OLGA, care must be taken regarding the order in which those side-effects will be executed.

8.5.3 Example

Here are two examples which solve the problem stated in the introduction, i.e. the generation of fresh labels for assembly code. It is assumed that those labels must be OLGA *strings* of the form "Lnnn" where "nnn" is a decimal number, different for each of those labels the target language is C.

In the first version, the external function returns a new integer each time it is called; the conversion into a string is expressed in OLGA.

In C:

```
static int label_num = 0;

int new_label_num ()
{
   return label_num++;
}
```

In OLGA:

```
function new_label (): string is
   declare
      function new_label_num (): int is
         external;
   in
      "L" + string (new_label_num ());
end function ;
```

In the second version, the external function directly returns the requested string.

In C:

```
static int label_num = 0;
static char buffer [16];
/* should be large enough to contain any integer plus letter "L" */

char *new_label_string ()
{
   sprintf (buffer, "L%d", label_num++);
```

```
    return buffer;
}
```

In Olga:

```
function new_label (): string is
    external new_label_string;
```

This second version will be probably less efficient due to the necessity to copy the returned string at each call, whereas the Olga compiler would generate the same `sprintf` call as above to implement the body of the function `new_label` of first version (this saves the copy operation).

Chapter 9

Expressions and Constants

In an applicative language such as OLGA, the main objects which can be manipulated are *values*, which are computed by evaluating *expressions*. This chapter describes most of the expressions which may be used in functions. Chapter 13 describes other expressions which may be used only in semantic rules to define attribute values, and chapter 10 describes expressions which deal with exceptions.

In this chapter and the sequel of this OLGA *Reference Manual*, the word “type” will denote any valid type expression, as defined in previous chapter.

9.1 Atomic Operands

As of expressions are manifest constants, which are lexical representations of constants of predefined types, identifiers which may designate function formal parameters, declared constants, etc., set and list constructors as presented in chapter 7, normal function calls, accesses to fields of record values, and more complicated expressions embedded in parentheses.

Syntax:

```
<expression>          =  <INT>;
<expression>          =  %REAL ;
<expression>          =  %CHAR ;
<expression>          =  %STRING ;
<expression>          =  <id>;
<expression>          =  <set constructor>;
<expression>          =  <list constructor>;
<expression>          =  <function call>;
<expression>          =  <field access>;
<expression>          =  "(" <expression> ")";
<field access>        =  <expression> "." <ID>;
```

Manifest constants are of four kinds: integral numbers for type *int*, real numbers for type *real*, character constants for type *char*, and string constants for type *string*. They are described in chapter 6.

Identifiers in expressions designate several kinds of values which are computed elsewhere, such as formal parameters of functions (see chapter 8), constants of enumerated types (see chapter 7), declared constants and values (see section 9.11 below and chapter 11), and other kinds described in this and other chapters. The association between their name, their type and their value (if it is constant) is recorded in the environment. When such an identifier appears in an expression, it is searched in the environment; if it cannot be found, then it is an "undeclared identifier" error, else its type and the means to compute its value are known. The precise definition of `<id>` appears in chapter 11.

Set constructors compute values of set types given the values of their elements, and the same holds for list constructors. They are both described in chapter 7.

Normal function calls are described in chapter 8. The actual parameters are computed, control is given to the function, and the value of the call is the value of the function body evaluated in an environment where the names of the formal parameters are dynamically associated to the values of the actual parameters. The type checking of such calls is also described in chapter 8.

Field accesses first involve the computation of a value of a record type it is an error ("not a record") if the expression does not belong to a record type. This type should be determinable statically. Then the field name is searched in the definition of that record type; if it cannot be found, it is an error ("unknown field"), else the type and the value of that field are retrieved, respectively from the record type definition (statically) and from the value of the record (dynamically).

Expressions embedded in parentheses are complicated expressions which, from a syntactical point of view, are of a lower priority than that of the construction of which they are an operand. Priority levels and associativity; rules are described in the last section of this chapter.

9.2 Operators

Operators are just function calls with special names and a special syntax which make them look like mathematical formulas.

Syntax:

```
<expression>         =  <expression> "+" <expression>;
<expression>         =  <expression> "-" <expression>;
<expression>         =  <expression> "*" <expression>;
<expression>         =  <expression> "/" <expression>;
<expression>         =  <expression> "%" <expression>;
<expression>         =  <expression> "=" <expression>;
<expression>         =  <expression> "!=" <expression>;
<expression>         =  <expression> "~" <expression>;
<expression>         =  <expression> "!~" <expression>;
<expression>         =  <expression> "<" <expression>;
<expression>         =  <expression> ">" <expression>;
<expression>         =  <expression> "<=" <expression>;
<expression>         =  <expression> ">=" <expression>;
<expression>         =  <expression> "&" <expression>;
<expression>         =  <expression> "|" <expression>;
<expression>         =  "-" <expression>;
<expression>         =  "+" <expression>;
```

```
<expression>          =  "!" <expression>;
```

The semantics and type checking of such operators is that of function calls, described in chapter 8. The significance of predefined operators is also presented in that chapter. Priority levels and associativity rules are described in the last section of this chapter.

9.3 Non-strict Boolean Operators

The boolean conjunction and disjunction operations also come in a non-strict version.

Syntax:

```
<expression>          =  <expression> and <expression>;
<expression>          =  <expression> or <expression>;
```

The semantics of these operators differ from that of function calls in that not all the operands are systematically evaluated; rather, just enough of them are evaluated, starting with the leftmost one, until the result is known. More precisely the semantics of `A` **and** `B` is that of **if** `A` **then** `B` **else** `false` and `A` **or** `B` is equivalent to **if** `A` **then** `true` **else** `B` (see section 9.6).

Regarding type checking, those operators have profile `(bool; bool): bool`, as their strict version.

9.4 The *let* Operator

The **let** operator introduces new names for local subexpressions.

Syntax:

```
<expression>          =  <let>;
<let>                 =  let <let clauses>
                         in <expression>;
<let clauses>         =  <let clauses> ","
                         <let clause>;
<let clauses>         =  <let clause>;
<let clause>          =  <ID> <type constraint> ":=" <expression>;
```

The expressions in the right hand sides of the let-clauses are evaluated—in an unspecified order—in the environment of the whole expression. Then a new environment is set up, in which the type and value of each of these expressions are associated to the corresponding identifiers, which must be different from one another. Then the expression in the right-hand side of the let-expression is evaluated in that new environment, where the identifiers of the let-clauses behave as declared values (see below). The type and value of that expression are then those of the whole expression.

If a type constraint is associated to the identifier of a let-clause, the type of the right-hand side expression is matched against that type constraint; it is an error if the match fails. This match may involve a dynamic range check. Otherwise the right-hand side expression must have a unique type, which will be that of the identifier.

9.5 Conditional Expressions

Conditional expressions allow the selection of one among several values according to the values of boolean conditions.

Syntax:

```
<expression>        =  <conditional>;
<conditional>       =  if <top expression> then
                          <expression>
                       <elsif part>
                       else
                          <expression>
                       end if ;
<elsif part>        =  <elsif part>
                       elsif <top expression> then
                          <expression>;
<elsif part>        =  ;
```

The value of the expression following the **if**, which must be of type *bool*, is evaluated. If that value is true, then the value of the whole conditional is that of the expression which follows the **then**. Otherwise, if an **elsif** ... **then** part is specified, the corresponding boolean condition is evaluated in the same way and, if true, the corresponding expression is evaluated and returned. This process is repeated in turn for each of the elsif-parts, until one condition evaluates to true. If no such condition evaluates to true, then the expression following the (mandatory) **else** is evaluated and returned.

As said before, the type of all the condition expressions must be *bool*. All the expressions following the **then**s and the one following the **else** must have the same type; that common type is that of the whole conditional expression. Briefly said, a conditional expression has profile

```
(bool; @T; bool; @T; ...; bool; @T; @T): @T
```

although this does not fit exactly in the type checking formalism of OLGA described in previous chapter (because of the variable number of arguments).

9.6 Selection Expressions

A *selection expression* allows the selection of a value among several ones, according to a combination of the following tests:

- *equality* of some scalar value with some constant;
- *structural matching* of some value against a *pattern* made of constructors, variables and sub-patterns;
- testing the *tag* of some value of a union type.

The former is the applicative counterpart of the classical selection operation which exists in different forms in a number of languages: Pascal (**case ... of**), C (**switch**), Ada (**case ... is**), etc. The second kind allows to match a value of a structured type (record, list or tree, see chapter 14) against a set of patterns; it is common in functional languages such as ML and Hope.

Syntax:

```
<expression>           =  <case>;
<case>                 =  case <top expression> is
                             <case clauses>
                          end case ;
<case clauses>         =  <case clauses>
                          <case clause>;
<case clauses>         =  <case clause>;
<case clause>          =  <pattern> ":" <expression> ";";
<pattern>              =  <named pattern>;
<pattern>              =  <unnamed pattern>;
<pattern>              =  <universal pattern>;
<named pattern>        =  <ID> ":=" <unnamed pattern>;
<unnamed pattern>      =  <id> "(" <pattern *> ")";
<unnamed pattern>      =  <list type name> "[" <pattern *> "]";
<unnamed pattern>      =  undef ;
<unnamed pattern>      =  undef ":" <type expr>;
<unnamed pattern>      =  "[" <case constant +> "]";
* it was necessary to expand <type constraint> here to solve (with
* a screwdriver!)  the shift/reduce conflict on ":"
<universal pattern>    =  <ID>;
<universal pattern>    =  <ID> ":" <type expr>;
<universal pattern>    =  other ;
<pattern *>            =  ;
<pattern *>            =  <pattern +>;
<pattern +>            =  <pattern +> "," <pattern>;
<pattern +>            =  <pattern>;
<case constant +>      =  <case constant +> "," <case constant>;
<case constant +>      =  <case constant>;
<case constant>        =  <constant expr>;
<case constant>        =  <constant expr> ".." <constant expr>;
```

The expressions in the right-hand sides of the case-clauses must all belong to the same type, which may be any type; that common type will be that of the whole case-expression.

Let us examine, case by case, which patterns match a given value.

If the value belongs to a scalar type `ST`, it is matched only by patterns surrounded by square brackets `[]` and listing constant expressions (or intervals thereof) of the same type `ST`. If the value is known to belong to a subrange type—e.g. because it is an attribute occurrence, or a let-identifier, declared to be of a subrange type—then `ST` is that subrange; otherwise `ST` is the host type of the type of the value. A case-constant which has the shape of a subrange stands for the list of constants which belong to that subrange, bounds included. A match with a given pattern succeeds if and only if the value of the expression to be matched belongs to the list of constants specified in the pattern.

If the value to be matched belongs to a record type, say

type RT = **record** F_1: T_1; F_2: T_2; ...; F_n: T_n **end record** ;

it can be matched only by a pattern of the form

RT(P_1, P_2, ..., P_n)

where the P_i's are subpatterns which can match values of type T_1, T_2 ..., T_n respectively. The match succeeds if and only if the value of each field F_i of the record recursively matches the pattern P_i. Any of the P_i's can be a "universal" pattern, either **other** or a pattern variable. In the latter case, the type constraint is optional; if it is given, then it may only be T_i; furthermore, the name of the variable will be visible in the RHS of the corresponding clause and will be known to belong to type T_i (this is true of any pattern variable, see below).

If the value to be matched belongs to a list type, say

type LT = **list of** T ;

then it can be matched by the following patterns:

- LT() or LT[]
  matches only if the value is the empty list of type LT.
- LT(P) or LT[P]
  matches only if the value is a list of type LT with one element of type T matching the pattern P.
- LT[P_1, P_2, ..., P_n] $n \geq 2$
  matches only if the value is a list of n elements, each of which matches the corresponding pattern P_i.
- LT(P_1, P_2)
  The patterns must be sufficiently precise to ensure that either P_1 is of type T and P_2 of type LT or *vice versa*; this can always be achieved by adding type constraints to pattern variables. The match succeeds only if the value is a non-empty list and:
  - in the former case, the head of the list matches P_1 and the tail matches P_2 or
  - in the latter case, the last element of the list matches P_2 and the sublist with all elements but the last matches P_1.

If the value to be matched belongs to a union type, say

type UT = **union** T_1; T_2; ... T_n **end union** ;

then it is matched only by patterns of the form

UT (P)

where P is a pattern which can match values of only **one** of the types T_i. If P is a pattern variable, then the type constraint **must** be explicitly given. The match succeeds only if the hidden tag of the union value indicates the value has the same type as P and this value matches P.

There are also tree patterns which can match values of tree types; this is discussed in chapter 14.

In addition an **undef** pattern is provided for every type but tree types (see section 9.9.2 for more details).

If any of the subpatterns is a pattern variable, it will be visible in the RHS of the corresponding clause with the appropriate type.

The "named pattern" construct allows both to match a (sub-)value with a (sub-)pattern and to refer to this value by a name in the RHS of the clause. This facility is akin to the **as** construct of ML and SSL [RT89].

Every selector expression is matched by a universal pattern, **other** or a pattern variable[1] (possibly with a type constraint).

If a given variable occurs more than once in a pattern, then the type determined for each of its occurrences must be the same this also means that a value matches such a pattern only if all the occurrences of that variable have the same actual value (see below).

The set of patterns must also verify two conditions:

- It must be *complete*: every possible value of the selector expression must be matched by at least one pattern; this can always be achieved by using a "default clause" triggered by a "universal pattern".

- It must be *non-ambiguous*: if a value is matched by several patterns, then one of them must be more specific than all the others.[2]

Furthermore these two conditions must be statically checkable by the OLGA compiler.

The dynamic semantics of a case-expression is as follows: the selector expression is evaluated. Then it is matched against all the patterns in parallel; if a variable occurs more than once in a pattern, then the corresponding values are tested for equality (using the = function) to determine whether the matching succeeds.[3] Then the most specific pattern among those which match is selected; note that the matching always succeeds because of the above completeness condition. Furthermore, that most specific pattern is always uniquely determined due to the above non-ambiguity condition. The variables in the pattern are bound to the corresponding sub-values in the actual value of the selector. A new environment is set up which records these bindings and the expression in the right-hand side of the selected clause is evaluated in that environment. The value of this expression becomes the value of the whole case-expression.

Chapter 14 gives more details about tree constructors and matching tree values. One of the attribute grammars in appendix A gives examples of the **case ... is** construct.

---

[1]This provides the same effect as a **let**, i.e. it allows to refer to the whole selector expression by a single identifier in the "default clause".

[2]A pattern A is more specific than a pattern B iff the set of values which match A is strictly included in the set of values which match B; note that this may be the effect of the repetition of some variable.

[3]Note that this test might not terminate if applied to values of recursive types (see section 8.3, note 10). It is thus recommended to avoid this possibility...

9.7 Range Tests

Range tests check whether a value of a scalar type belongs to a specified subrange.

Syntax:

```
<expression>        = <range test>;
<range test>        = test <expression> type <type id>;
<range test>        = test <expression>
                      range <expression> ".." <expression>;
```

The subrange type in the right-hand side is either explicit, i.e. defined by its bounds, or implicit, i.e. it must be the name of a subrange type declared elsewhere. In the former case, the bounds may be computed dynamically; however they must belong to the same host type. The expression in the left-hand side must belong to a scalar type which must be the host type of the subrange type. The type of the whole expression is *bool*.

The expression is evaluated, and its value is tested to determine whether it belongs to the specified subrange. If so, the value of the test is "true", else it is "false". If the subrange is evaluated dynamically and is such that its "lower bound" proves to be strictly greater than its "upper bound", the value of the test is false.

9.8 Error Messages

In a number of applications, e.g. compiler front-ends, it is sometimes necessary to tell the author of the source text about non-syntactical errors. The OLGA abstraction of these *error messages* is the **message** expression.

Syntax:

```
<expression>        = <message>;
<message>           = message <expression>
                         position <position>
                         severity <expression>
                         value <expression>;
<position>          = <symbol>;
<position>          = value <expression>;
```

The type and value of the **message** expression are those of the last expression. However evaluating such a construct also has the side-effect of communicating to the user a message whose text is the value of the first expression, attached to the position in the source text defined by the second parameter, with a severity level which is the value of the third expression. The actual mechanism used for the communication of the message is completely implementation-defined; see the corresponding sections in *the* FNC-2 *System Reference Manual.*

The position of that message is specified either by a value of type *source-index* (following the word **value**) or by a grammar symbol; this latter specification is only valid inside semantic rules where symbols are visible (see chapter 13) and **position** `symbol` is then equivalent to

position value **$scx**`(symbol)`. A **message** construct may appear outside a semantic rule (e.g. in a function body) provided a value of type *source-index* is available (e.g. passed as parameter).

The severity level must lie in an implementation-defined subrange of *int* with lower bound 0 (zero), which means that this construct may raise the `RANGE-ERROR` exception if the actual value does not lie in that range.

9.9 Construction of Circular Structures

9.9.1 The *letrec* Operator

The **letrec** operator is a special version of the **let** operator for constructing circular structures.

Syntax:

```
<expression>          =  <letrec>;
<letrec>              =  letrec <let clauses>
                         in <expression>;
```

The expressions in the right-hand side of the let-clauses are evaluated simultaneously in a new environment in which the identifiers in the left-hand side of the clauses are visible. This allows to construct mutually recursive values. Type inference is performed globally on the whole set of let-clauses. The "dependency graph" derived from the clauses may have any form, including cycles,[4] but it is required that every cycle in this graph includes a call to a construction function associated to a structured type (see chapter 7).

The same new environment is set up for the expression after the keyword **in**, in which the left-hand side variables are visible and refer to (parts of) the circular structure. The type and value of that expression are then those of the whole expression.

Important note: Present compilation techniques make the **letrec** construct a rather theoretical notion, because it can actually be compiled only in very limited cases which are, additionally, very hard to characterize. The OLGA compiler will put very stringent restrictions on the use of **lectrec**, so it is probably better to forget about it presently and wait till new research results allow effective use of it.

9.9.2 *undef* and *define*

Because of the above problems in using **letrec**, OLGA provides a more "imperative" notion for constructing circular structures, in which the physical replacement of a value in a "variable" by a new value is made explicit.

Syntax:

```
<expression>          =  <undef>;
<expression>          =  <define>;
<undef>               =  undef <type constraint>;
<define>              =  define <variable> ":=" <expression>;
<variable>            =  <expression>;
```

[4]since it is exactly the purpose of this construct...

In every type (except tree types), there is a special undefined value denoted by the **undef** keyword. This value can be manipulated (e.g. be assigned to an attribute, passed as parameter to a function or tested for equality) as any other value of the same type, except that you cannot access a field of an **undef**'ined record and that all the predefined functions (see section 8.3) raise the predefined exception `UNDEFINED` (see chapter 10) if any of its operands is the **undef** value of the appropriate type; this is however not the case for the (in)equality and (non)-identity tests, i.e. you may safely check at runtime whether a given value is **undef** by using any of these predicates (equality and identity are equivalent when either of their operands is **undef** of any type). **Undef** can also be used as a pattern in the **case ... is** construct (see section 9.6). The match succeeds if and only if an equality or identity test with **undef** would succeed.

undef itself has universal type `@T :` the type of each of its occurrences must be determinable from the context (see chapter 8) or be specified by an explicit type constraint.

The **define** "instruction" allows to physically replace the value of some member of an "assignable structure" or "variable" by a non-undefined value of the same type, provided the original value is undefined (which means than you can **define** a variable only once), otherwise the predefined exception `DEFINED` is raised.

Although a `<variable>` can syntactically be any expression, only the following expressions are valid:

- a record field the `<variable>` must identify that field;
- the head of a list the `<variable>` must be the list itself and must be non-empty, otherwise the `EMPTY-LIST` exception is raised instead of the possible `DEFINED`;
- a value of a union type the `<variable>` must be the value itself.

In the former case, the expression in the RHS must have the same type (or be constrained by the type checking process to the same type) as the record field; its value replaces the **undef** in that field.

In the second case the expression in the RHS must have the same type as the element type of the list; its value replace the **undef** in the head of the list.

In the latter case the expression in the RHS must be of (or be *uniquely* constrained to) one of the element types of the union type; the tag of the union value is then set to indicate that type and the value of the expression is assigned to the value of the union.

In all three cases the whole **define** expression is of type *void* whose only value is *null* (see section 7.3.8).

Example:

```
let l:= ...  {some list with undef as itshead}
in let dummy:= define l:= ...  {a new value for thehead}
   in l        {return the physically modified list}
```

If you want to replace the second element of the list you may write;

```
let l:= ...  {some list with undef as second element}
in let dummy:= define tail(l):= ...  {a new value for that elem}
   in l        {return the physically modified list}
```

The **undef/define** constructs can be used to build circular structures as follows: assume, without loss of generality, that the structure to be constructed is of the form

where S is any "big" structure and `X` an assignable substructure of `S` intended to contain a reference to `S`.

1. Build `X` as **undef**
2. Build the rest of `S` using that `X`
3. **define** `X` to be `S`.

Note that these three steps need not be contiguous in time, provided that you don't try to use the value of `X` for any useful purpose before **define**'ing it (you may however examine it in the meanwhile).

9.10 The *map list* Operator

The **map list** construct traverses a list value and computes a value using each element of the list in turn.

Syntax:

```
<expression>        =  <map list>;
<map list>          =  map list <expression> <map direction>
                          <map id> value <expression>
                       end map ;
<map direction>     =  left ;
<map direction>     =  right ;
<map id>            =  <id>;
<map id>            =  <operator>;
```

The **map list** construct allows to traverse the list either in the left-to-right (keyword **left**) or right-to-left (keyword **right**) direction and accumulate a value computed by the application of a specified function to that accumulated value and the "current" element of the list. The specified function, which may be an operator function (see chapter 8), must thus be such that the type of its first argument and the type of its result unify (see chapter 8); this common type must also unify with the type of the expression following the word **value**, which is the initial value of the accumulator; that common type will be that of the whole **map** expression. The type of the second parameter of the map-function must unify with that of the elements of the list, which itself is given by the expression following the **map list** keywords. In other words, the map-function must be of type `(@`T_1`; @`T_2`): @`T_1, the first expression must be of type list of @T_2 and the second expression must by of type @T_1; the type of the whole map expression is then @T_1.

The dynamic semantics of the **map** construct is as follows: first, the initial value of the accumulator (the expression following the word **value**) is computed; if the list is empty, then this value is the result of the whole construct. Otherwise, the list is traversed—starting with the leftmost one if the direction is **left**, or the rightmost one if the direction is **right**—and the following process is repeated on each element: the specified function is called with its first parameter bound to the value of the accumulator and the other parameter bound to the current element; the result of this call becomes the new value of the accumulator. When the last element has been processed, the value of the accumulator is the result of the whole **map** construct.

It is not possible to **map** the non-strict boolean operators **and** and **or**; this is anyway not necessary because, since it is not possible to terminate a traversal of a list except by raising an exception, the same result can be achieved using the strict operators `&` and `|`.

Example: we can write a polymorphic function `member`, testing whether a given element belongs to a given list, as follows:

```
function member (x: @T; l: list of @T): bool is
declare
   inline function equal-to-x (old: bool; y: @T): bool is
      if y = x {= must be polymorphic } then
         true
      else
         old
      end if
   end function ;
in
   map list l left equal-to-x value false end map
end function ;
```

A theoretically more efficient way to do this would be to raise an exception when the value is found, so that we don't traverse uselessly the sequel of the list:

```
function member (x: @T; l: list of @T): bool is
declare
   exception found ;
   inline function equal-to-x (dummy: bool; y: @T): bool is
      if y = x {= must be polymorphic } then
         raise found
      else
         false
      end if
   end function ;
in
   map list l left equal-to-x value false end map
   catch
      found: true ;
   end catch
end function ;
```

Of course, it is also possible to write `member` without a **map list**:

```
function member (x: @T; l: list of @T): bool is
```

```
    if empty(l) then
       false
    elsif x = head(l) {= must be polymorphic } then
       true
    else
       member(x, tail(l))
    end if
 end function ;
```

More generally, it is possible to never use the **map list** construct, i.e. it does not improve the power of the language, but we felt it enhances readability.

9.11 Constant Expressions, Declared Constants and Values

From the semantical point of view, a *constant expression* in OLGA is just any expression whose value is computable at compile time. Thus, regarding the syntax, it is just an expression.

Syntax:

```
<constant expr>       =  <expression>;
```

However, constant expressions rather than dynamically valued ones must be used in a number of contexts such as case-constants, subrange types bounds, etc. If so, it is always specified in the corresponding sections.

An expression is computable at compile-time, i.e. it is a constant expression, iff it is composed only of:

- manifest constants,
- constants of enumerated types,
- declared constants (see below), and
- application of predefined functions and operators (excluding those of the *string-functions* and *math-functions* modules) to constant expressions.

As opposed to that notion, *declared constants* and *values* in OLGA associate names to expressions in a block; in the former case, the expression must be a constant expression, whereas in the latter case, the expression is possibly not computable before run-time.

Syntax:

```
<const part>          =  const
                         <const decls>;
<const decls>         =  <const decls>
                         <const decl>;
<const decls>         =  <const decl>;
<const decl>          =  <ID> ":=" <constant expr> ";";
```

```
<value part>         =  value
                        <value decls>;
<value decls>        =  <value decls>
                        <value decl>;
<value decls>        =  <value decl>;
<value decl>         =  <ID> <type constraint> ":=" <expression> ";";
```

Constant and value declarations are to a block what a **let ... in** construct is to a single expression, that is a means to name common subexpressions. The main difference is that a declared constant or value is visible throughout the whole block (except where hidden by deeper declarations, see chapter 11); it hence may be used in other declarations, e.g. for types (constants only), and in the body of locally defined functions (constants and values).

The expression in the right-hand side of a constant declaration must be a constant expression. The expression in the right-hand side of a value declaration need not be computable at compile-time: for instance, if declared in a function body, it may refer to formal parameters of that function, or, if declared in a production (see chapter 13), it may refer to input attributes of that production. In that latter use, the declared value plays the role of a *local attribute*, i.e. an attribute attached to a production (operator) rather than to a non-terminal (phylum).

The type of a constant expression is always manifest hence a constant declaration requires no type constraint. As for a value declaration, if no type constraint is specified, the expression must be a top level expression with a single type, and that type will be that of the value; note that this type may be polymorphic in polymorphic functions. If a type constraint is specified, then the expression need not be a top level one, but the type constraint must unify with one and only one of the possible types; the type of the value and of the expression will be that specified in the constraint. If the type constraint specifies a subrange type, then a range test will be performed at run-time and a `RANGE-ERROR` exception may be raised.

The expression in the right-hand side of a value declaration will be evaluated at run-time, after the block where it appears is entered, and before any use of the value; however the exact moment for this computation is otherwise unspecified. See also chapters 11 and 13 for more details.

9.12 Priority Levels and Associativity Rules

This section lists all the OLGA operators and keywords used in the construction of expressions, together with their priority levels and associativity rules. They are ordered from higher priorities (designated by lower numbers) to lower priorities (designated by higher numbers).

Associativity rules specify the interpretation of expressions such as `A + B + C`:

- *left* means that the interpretation is `(A + B) + C`;
- *right* means that the interpretation is `A + (B + C)`;
- *no* means that such a writing is a syntactical error;
- — means that this concept is irrelevant; this is generally the case for completely bracketed expressions such as **if ... then ... else ... end if**.

Completely bracketed expressions can be considered as highest priority expressions.

Priority	Operator or main keyword	Associativity	Comments
0	`( )`	—	parentheses
	id`( )`	—	function call
	id`[ ]`	—	list constructor
	id`[| |]`	—	set constructor
	if	—	
	case ... is	—	
	case position	—	only in semantic rules
	case arity	—	only in semantic rules
	map	—	only in semantic rules
	map list	—	
	undef	—	
1	**with**	no	see chapter 14
2	`.`	left	field access
3	`-` (unary)	right	unary minus
	`+` (unary)	right	unary plus
	`!`	right	
4	`*`	left	
	`/`	left	
	`%`	left	
5	`+` (binary)	left	
	`-` (binary)	left	
6	`=`	no	
	`!=`	no	
	`~`	no	
	`!~`	no	
	`<`	no	
	`>`	no	
	`<=`	no	
	`>=`	no	
7	`&`	left	
	`|`	left	
8	**and**	right	
	or	right	
9	**test**	no	range test
10	**let**	right	
	letrec	right	
	message	right	
	define	right	
11	**catch**	no	see chapter 10

Chapter 10

Exception Handling

10.1 Introduction

Exceptions have become in the past few years a well-accepted means of handling all abnormal situations which may occur during a computation these abnormal situations may be due to arithmetic errors such as division by zero, hardware errors such as memory violation, and other causes leading to the termination of the whole program if they were not properly *handled*. Modern programming languages, e.g. Ada [ANS83], CLU [BRB$^+$81], ML [CHP85, HMM86], Le_Lisp [CDD$^+$86], provide constructions allowing the program itself, rather than the operating system, to catch those exceptions and to handle them more gracefully than by aborting the whole program. Incidentally, this idea is not new since we can trace it back to the early days of PL/1 in the late 60's. However it was cleanly formalized only a few years ago [Goo75].

The modern languages quoted above have extended this concept of abnormal situations to *programmer-defined* situations by allowing him to explicitly *raise* an exception when, say, the algorithm he is writing does not apply to the situation he has just detected. Giving the programmer the power to raise and handle his own exceptions in just the same way as run-time errors also improve the coherence of the language design. Furthermore, exceptions are becoming more and more widely used to implement *non-local exits* in a structured way, allowing to write thousands of lines of code without a mere GOTO.

Exceptions are even more appreciated in an applicative language, because in some situations one might like to abort a whole computation perhaps involving deeply nested recursions. ML and Le_Lisp provide useful constructs to deal with exceptions and non-local exits. It was thus very useful, or even necessary, to provide such constructs in Olga.

Olga exceptions may optionally return useful values, in the form of parameters, from a deeply nested expression. As a consequence, exceptions with parameters will be used for non-local exits, whereas parameterless ones will more likely be used to express true errors for which no significant value is available.

10.2 Exceptions Declaration and Predefined Exceptions

Syntax:

```
<exception part>      =  exception
```

```
                              <exception decls>;
<exception decls>      =   <exception decls>
                              <exception decl>;
<exception decls>      =   <exception decl>;
<exception decl>       =   <identifier +> <formal parameters> ";";
```

Exception names belong to the same name space as constants, values, etc. (see chapter 11). Exceptions with a non-empty formal parameter list will return values of the types specified in that list. Type constraints are **mandatory** for formal parameters of exceptions, and the type denoters appearing in the corresponding type expressions must be visible in the scope in which the exception declaration appears.

Example:

exception
`key-not-found ();`
`key-found (info:  info-type);`

All of OLGA predefined exceptions are parameterless ones, because they correspond to true error conditions. Here comes the list of their names, together with a short and informal description of the context(s) in which they may be raised; see the corresponding chapters for more information.

- `ARITY-ERROR`
  raised on a list node which expects a minimum number of offsprings—due to the shape of some semantic rule—when this minimum is not reached (section 13.4).
- `CONVERSION-ERROR`
  raised during the conversion of a string value into a value of a simple type, when the contents of the string is not a legal denotation of any value of that simple type (section 7.11).
- `DEFINED`
  raised by a **define** instruction (section 9.9.2) when the variable to be physically replaced is not **undef**.
- `EMPTY-LIST`
  raised when an empty list is passed as argument to the functions *head* and *tail* (section 8.3) and when attempting to **define** the head of an empty list.
- `RANGE-ERROR`
  raised whenever a range constraint is violated, for instance in the following contexts: assignment to an attribute, parameter passing, function result return, set construction, conversion between scalar types and from strings (chapters 7, 8 and 13).
- `TREE-ERROR`
  raised by list-extension tree construction functions when the initial and resulting lists are not headed by the same operator (chapter 14).

- `UNDEFINED`
  raised when trying to access a field of an **undef** record and by every predefined function—apart from the (in)equality and (non)-identity predicates—and the functions defined implicitly for each new enumerated type, when any of its operand is **undef**.

- `ZERO-DIVIDE`
  raised by division and remainder operations when second argument is zero (section 8.3).

10.3 Exceptions Handling

10.3.1 Exception Raising

Exceptions are raised either implicitly during the computation of some predefined operation or function (see the predefined exceptions above), or explicitly by the programmer.

Syntax:

```
<expression>          =  <exc raise>;
<exc raise>           =  raise <id> "(" <actual *> ")";
```

This construction raises an exception and optionally returns some values, whose types must be those of the formal parameters specified in the declaration of the exception; these values are then made available to the exception handler (see next subsection). From the point of view of type checking, this expression belongs to an unnamed type which is compatible with any other type, which means that it may appear where any other (typed) expression is expected without violating type checking rules. This also means that a **raise** expression is not a top expression, and that the type of an expression containing a **raise** construct must be determinable by other means.

Example:

```
case node is
   bin-tree (null:  void) :
      raise key-not-found () ;
   bin-tree (tree-node (the-key, the-info, left-s, right-s)) :
      if the-key = key then
         raise key-found (the-info)
      else ... end if ;
end case
```

10.3.2 Exception Catching

Exceptions are caught by *exception handlers* associated to expressions in which they can be raised. When an exception is raised, the computation of the current expression is aborted and control is returned to the (dynamically) nearest corresponding exception handler. The handler must supply a value of the same type as that of the expression which was aborted; that value will be that of the whole expression. If the exception has parameters, then the values which were given to them when it was raised are made available to the handler, which can use them to compute the value which will be substituted to that of the expression which was aborted. Several exceptions may be caught by the same handler and may provide different values.

Syntax:

```
<expression>           =  <exc catch>;
<exc catch>            =  <expression>
                          catch
                             <exception handler +>
                             <default handler>
                          end catch ;
<exception handler +> =  <exception handler +>
                          <exception handler>;
<exception handler +> =  <exception handler>;
<exception handler>   =  <id> ":" <expression> ";";
<default handler>     =  other ":" <expression> ";";
<default handler>     =  ;
```

Each exception handler specifies the name of the exception it catches, and an expression whose value will be substituted for that of the expression to which the handler is associated. If the exception has parameters, a new scope is opened for the expression which will be substituted, in which the names of the parameters, as specified in the declaration of the exception, are made visible, with a type as specified in the declaration. The expression which will be substituted must have the same type as that to which the handler is associated.

If a default handler is provided in the list, which is specified by putting the keyword **other** instead of an exception name, then it must be unique and the last one in the list. This default handler catches any exception which is not caught by another handler in the list; however in that case, if any values were returned when the exception was raised, they are ignored by the handler. The provided expression must follow the same rules as for normal handlers.

Note that the computation of the "handler" expression may itself give rise to an exception—the same as the one which is being caught, perhaps with other parameters, or a different one—; in that case, this new exception is not caught by the current handler but by the one which dynamically encloses it.

It is forbidden that an exception propagates without being caught out of the scope in which it is declared or visible. The OLGA compiler will (try to) enforce that rule, using the list of exceptions given in the header of each function.

As already stated in chapter 8, the header of each function should list exactly those exceptions which might be raised during the computation of its body and local declarations.

10.3.3 Examples

```
where var -> ID use
$type (var):=
   let id-info:= lookup ($id (ID), $symbol-table)
   in if id-info.kind = variable then
         id-info.the-type
```

```
    else
      message "Not a variable"
        position ID
        severity 2
        value no-type
    end if
  catch
    key-not-found :
      message "Undeclared identifier"
        position ID
        severity 2
        value no-type ;
  end catch ;
```

In the above example, borrowed from a Pascal compiler front-end, the **catch** exception handler applies to the **let** ... **in** expression. The *lookup* function searches the identifier in the symbol table; if it exists then the value of the whole expression is that of the conditional following the **let**—either the type of the variable if the identifier is actually a variable, or the default value *no-type* if not. If the lookup fails, the *key-not-found* parameterless exception is raised and caught by the handler, which outputs an error message because the identifier is undeclared, and supplies the same default value.

```
function lookup-in-tree (id: token; st: symbol-table)
           : info-type raise key-not-found, key-found is
  case st is
    symbol-table (null: void):
      raise key-not-found ();
    symbol-table (node: tree-node):
      if node.key = id then
        raise key-found (node.info)
      elsif id < st.key then
        lookup-in-tree (id, node.left-son)
      else
        lookup-in-tree (id, node.right-son)
      end if ;
  end case
end function ;

function lookup (id: token; st: symbol-table): info-type
                  raise key-not-found is
  lookup-in-tree (id, st)
  catch
    key-found: info ;
  end catch
end function ;
```

In this example the *symbol-table* data type is implemented as a binary tree. The *key-found* typed exception is used to terminate the recursive search of the tree and return the value found (although that was not strictly necessary). Notice that the *key-not-found* exception which could be raised in the *lookup-in-tree* function is not caught in the *lookup* function, so it propagates to the caller of the latter function. This accounts for the exception lists in the header of both functions.

Chapter 11

Structure, Visibility Rules and Modularity

11.1 Generalities

This chapter presents the structure of an OLGA specification and a number of related concepts: blocks, declarations, visibility rules, compilation units, modularity. Since these notions are very much interdependent, it was not easy to devise a consistent order to present them; this chapter will consequently look rather fuzzy, although it deals with structure!

This chapter deals only with modularity concepts which are relevant to any general-purpose applicative language. Modularity concepts dealing with attribute grammars are discussed in chapters 12 and 15.

OLGA is a block-structured language: some constructs open new scopes in which local objects can be declared. These blocks are strictly nested. The next section describes blocks, declarations and visibility rules.

Compilation units, i.e. the minimum "objects" which may appear alone in a file to be fed to the OLGA compiler, come in three kinds: definition of attributed abstract syntaxes, as described in chapter 12, *modules* which may define only constants, values, types, exceptions and functions, which are described in this chapter, and *attribute grammars*, presented in chapter 15.

Compilation units communicate with one another through *import* clauses and cross-modules visibility rules. An import clause allows to make visible at some point in some unit all or part of objects exported by another specified unit. Type checking is performed for imported objects. A library manager is associated to the FNC-2 system to keep track of the *dependencies* between units implied by import clauses and to keep such a library in a consistent state. These *modularity* features are presented in section 11.4.

11.2 Block Structure, Visibility Rules and Name Spaces

Blocks are high-level constructs of an OLGA specification in which local objects may be declared (implicitly or explicitly). Several levels of blocks are distinguished:

- Level 0 blocks open a new scope but only implicit declarations are performed. This is the case for e.g. the **let ... in** construct which opens a new scope for the right-hand

level	*construct*	*defining section*
0	**let ... in**	9.4
	letrec ... in	9.9.1
	case ... is	9.6 and 14.2
	case position ... is	13.6.1
	case arity ... is	13.6.2
	catch 10.3.2	
1	function chapter	8
	production	chapter 13
	AG phase	chapter 15
2	declaration module	this chapter
	definition module	this chapter
	attributed abstract syntaxes	chapter 12
	attribute grammars	chapter 15

side expression, in which the variables of the let-clauses are made visible.

- Level 1 blocks may contain local explicit declarations (see below) and import clauses in addition to implicit declarations, but none of these local objects may be exported; functions and productions are level 1 blocks.

- Level 2 blocks are compilation units: modules, attributed abstract syntaxes and attribute grammars. Each of these level 2 blocks has its own rules for local declarations and other modularity concepts, which are explained in the relevant parts of this manual.

However all of those blocks are properly nested and enforce the same *visibility rules.* The following table summarizes which Olga constructs are blocks.

An identifier declared at some point in a block is visible from that point (inclusive, since most declarations may be recursive) up to the end of the block, including in nested blocks except in those where it is redeclared; these are the visibility rules of Ada [ANS83]. Even in the blocks where an identifier is redeclared, the old declaration is still visible from the beginning of the block up to the redeclaration point (excluded). The redeclaration of an identifier hides outer declarations. Note that for functions those visibility rules apply to the combination of the function name and its profile, i.e. a function name may be overloaded. Also, several name spaces are used in Olga:

- one for names of compilation units: modules, attributed abstract syntaxes, and attribute grammars; this name space is *flat*, i.e. supports no block structure (but see section 11.4.2);

- one for attribute names, which is also flat; this is because attributes are in essence global objects; more details are given in chapters 12 and 13;

- one for type names (including type denoters), with block structure;

- one for function names, with block structure; this name space is special because of overloading;[1]

[1] This was one of the motivations for having a separate name space; the other one is that function names are always syntactically distinguishable from other names.

- and one for all other objects—constants, values and exceptions—, also with block structure.

There is a sixth name space for grammar symbols, to be presented in chapter 13.

The redeclaration of an identifier in a given name space does not hide the declarations of that identifier in other name spaces. However some declarations may effect several name spaces at a time: for instance the declaration of an enumerated type declares the type name in the type name space, its associated conversion functions in the function name space and its constants in the other "hold-all" name space. A given identifier cannot be declared twice in the same block and in the same name space (but see above for function names). There is no means to make an identifier visible before it is actually defined, except using the **and** connector to declare mutually recursive types or functions.

Only one block is associated to a function definition, which contains both the implicit declaration of formal parameters and other local explicit declarations; the function name and profile are declared in the enclosing block, but they are visible in the former, enclosed block (for recursivity), if not redeclared.

Predefined OLGA objects behave as if they were declared in a block enclosing each compilation unit; they are thus redeclarable at the outermost level in any unit. However this is bad programming practice and is not recommended.

No explicit declaration is allowed in level 0 blocks. In level 1 blocks, it is allowed to declare constants, values, non-opaque types, exceptions and functions. **Import** clauses may also appear in such blocks (see section 11.4).

Syntax:

```
<decl *>            =  ;
<decl *>            =  <decl +>;
<decl +>            =  <decl +>
                       <decl>;
<decl +>            =  <decl>;
<decl>              =  <const part>;
<decl>              =  <value part>;
<decl>              =  <type part>;
<decl>              =  <function def>;
<decl>              =  <exception part>;
<decl>              =  <import clause>;
```

11.3 Modules

OLGA modules allow to define a set of related constants, values, types, exceptions and functions, and to control the visibility of these objects outside the module. They are similar to compilation units found in other, more classical languages, e.g. global Modula-2 modules [Wir82] and Ada *packages* [ANS83].

A module is divided into two actual compilation units, a *declaration module* (DCM) which declares all the visible objects of the module and what is necessary to use them (e.g. the functions profiles), and a *definition module* (DFM) in which the actual definition (implementation) of these objects (and maybe other, non-visible objects) is given. The declaration module is the outside view of the module and represents the "contract" between the author of the module and its users, while the definition module contains the inner mechanics and should remain hidden to the user.[2] The author of the module may change the definition module without affecting the applications which use the module, provided of course that the declaration module remains unchanged.

Two features improve this clean separation between specification and implementation of a module:

- types and other objects in a declaration module may be declared as *opaque*, which means that their structure is not known to the user; only the functions provided in the declaration module may operate on an opaque type; functions are always opaque;
- a module may be *parameterized* ("generic" in the sense of Ada) by types and/or functions to adapt a general algorithm to a particular situation.

Both these features are exemplified in appendix A.

11.3.1 Declaration Modules

Syntax:

```
<compilation unit>    =  <DC module>
<DC module>           =  declaration module %ID <DCM parm *> is
                            <DCM decl +>
                         end module ";";
<DCM decl +>          =  <DCM decl +>
                         <DCM decl>;
<DCM decl +>          =  <DCM decl>;
<DCM decl>            =  <DCM const part>;
<DCM decl>            =  <DCM value part>;
<DCM decl>            =  <DCM type part>;
<DCM decl>            =  <function header> ";";
<DCM decl>            =  <exception part>;
<DCM decl>            =  <import clause>;
<DCM const part>      =  const
                         <DCM const decl +>;
<DCM const decl +>    =  <DCM const decl +>
                         <DCM const decl>;
<DCM const decl +>    =  <DCM const decl>;
<DCM const decl>      =  <const decl>;
<DCM const decl>      =  <ID> ":" <type id> ";";
<DCM value part>      =  value
                         <DCM value decl +>;
<DCM value decl +>    =  <DCM value decl +>
                         <DCM value decl>;
<DCM value decl +>    =  <DCM value decl>;
```

[2]It is possible, and even recommended, that a module author supplies the DFM only in compiled form.

```
<DCM value decl>       =  <value decl>;
<DCM value decl>       =  <ID> ":" <type id> ":=" <expression> ";";
<DCM value decl>       =  <ID> ":=" <expression> ";";
<DCM value decl>       =  <ID> ":" <type id> ";";
<DCM type part>        =  type
                             <DCM type decl +>;
<DCM type decl +>      =  <DCM type decl +>
                          <DCM type decl>;
<DCM type decl +>      =  <DCM type decl>;
<DCM type decl>        =  <type decl>;
<DCM type decl>        =  <ID> ";";
```

A declaration module specifies the visible part of the objects—constants, values, types, functions and exceptions—defined in the module; it also specifies the parameters (types and/or functions) of the module, if any (see section 11.3.3 for more information), and the list of objects imported from other modules necessary to establish those declarations.[3]

The visible part of a constant or value is its name, its type and maybe its value. If the value is not given, this declares an opaque constant or value the type must be given explicitly in this case. Note that, from a user's point of view, there is no difference between an opaque constant and an opaque value, since an opaque constant is not considered, outside the module, as a constant expression. However the difference is still valid in the associated DFM.

The visible part of a type is its name and maybe its definition (implementation). If the definition is given then all the operations applicable to that type (as defined in chapter 7, e.g. access to fields of a record) are available; in particular the type declaration also declares the associated conversion or construction function(s). Otherwise the type is opaque and no operation is implicitly defined for it, apart from (in-)equality and (non-)identity functions (`=`, `!=`, $\sim$, `!`$\sim$). The only functions defined on such an opaque type are those declared in the module. The only "operations" allowed in blocks which import this type are the declaration of attributes and formal parameters (of functions and exceptions) with this type, and passing actual parameters with this type. Furthermore, in the unification process used in type inference/checking and described in chapter 8, the name of an opaque type can only unify with itself or an unbound type variable, since its structure is unknown. Opaque types are very useful to implement *abstract data types*.

The visible part of a function is its complete header, composed of:

- its name (it may be an operator function);
- its profile, i.e. the list of types of its parameters and its result type; all type constraints are *mandatory*, but it is possible to declare a polymorphic function in a module by using explicit type expressions with explicit type denoters;
- and the list of exceptions it might raise.

The actual types and exceptions appearing in the profile of a function declaration must be either predefined ones, imported ones, or declared in the module (but see section 11.3.3

[3] If an imported object is only necessary for the *definition* of some object and not for its *declaration*, then the import clause need not appear in the DCM but only in the DFM.

for parameterized modules). The body of a function never appears in a DCM, but in the corresponding DFM, i.e. a function is always opaque. The **inline** specification is syntactically admitted but, for obvious reasons, will always be ignored.

The visible part of an exception declaration is the complete declaration, including its name and its parameter list. See above for possible parameter types.

All objects visible in a DCM, including imported objects, are visible from outside and may be imported in other blocks. This means that importations are transitive through DCMs.

Visibility rules in DCMs are the standard OLGA ones, in particular regarding the "declaration before use" rule. However opaque mutually recursive types or functions need not be introduced with the **and** connector since nothing in the DCM shows that they are actually mutually recursive. This does not apply to non-opaque mutually recursive types.

11.3.2 Definition Modules

Syntax:

```
<compilation unit>     =  <DF module>;
<DF module>            =  definition module %ID is
                             <decl *>
                          end module ";";
```

A definition module has the same name, cited in the header, as the corresponding DCM. A DFM must supply a definition for all objects declared in the DCM but not defined there, i.e. opaque constants, values and types, and functions. Objects declared and defined in the DCM are also visible in the DFM. Other local objects may be defined in the DFM but they will not be visible from the outside and cannot be imported.

A same enumerated type declaration may define both an opaque type and some opaque constants of the DCM.

Objects may be imported into the DFM from other modules. These objects are not visible outside the module.

The objects declared in the DCM for which the definition appears in the DFM must be defined using the usual OLGA declarations (those valid in level 1 blocks). In particular, each function definition must come with its complete header which must be the same, from a semantic point of view,[4] as the one appearing in the DCM.[5]

Visibility rules in a DFM are as usual, considering that non-opaque objects of the DCM are implicitly declared at the beginning of the DFM. Mutually recursive types and/or functions must be defined with the **and** connector.

A DFM cannot be compiled before the associated DCM has been compiled.

---

[4] The profile (number and types of parameters, and result type) and list of possible exceptions must be the same as in the DCM, but the syntactic appearance of these, in particular the names of the formal parameters, may be different. Also, consecutive parameters with the same type may appear in the same param-group or in different ones, regardless of how they are presented in the DCM. The names of the formal parameters visible in the body are those appearing in the DFM, not those of the DCM.

[5] The most simple method to achieve this is to use the text of the DCM as a template to construct the DFM; this is easy with any text editor.

11.3.3 Parametrized Modules

Syntax:

```
<DCM parm *>          =  ;
<DCM parm *>          =  "(" <DCM parm +> ")";
<DCM parm +>          =  <DCM parm +> ";"
                         <DCM parm>;
<DCM parm +>          =  <DCM parm>;
<DCM parm>            =  type <ID>;
<DCM parm>            =  function <function id>
                         "(" <DCM param group *> ")"
                         ":" <type id> <DCM func excp *>;
<DCM param group *>   =  ;
<DCM param group *>   =  <DCM param group +>;
<DCM param group +>   =  <DCM param group +> "," <DCM param group>;
<DCM param group +>   =  <DCM param group>;
<DCM param group>     =  <identifier +> ":" <type id>;
<DCM func excp *>     =  ;
<DCM func excp *>     =  raise <DCM func excp +>;
<DCM func excp +>     =  <DCM func excp +> "," <DCM excp id>;
<DCM func excp +>     =  <DCM excp id>;
<DCM excp id>         =  <ID>;
<DCM excp id>         =  <ID> "(" <DCM param group *> ")";
```

Modules may be parameterized by types and/or functions, so that the same module implementing a given concept, data structure or algorithm (e.g. symbol tables, as exemplified in appendix A) can be used in different situations (e.g. different types of keys and informations) without change.

A formal type parameter specifies only its name. No particular requirement on such a type is assumed, except for the availability of (in-)equality and (non-)identity functions, denoted as usual by the operators `=`, `!=`, $\sim$ and `!`$\sim$. If particular operations for this type are needed in the module, they must appear as function parameters of the module. In other words, a type parameter of a module has the same prerogatives as an opaque type. The formal type name is visible in the part of the parameter list following its appearance and in the DCM and DFM.

A formal function parameter specifies its name (it may be an operator function), its profile and the (formal) names and profiles of the exceptions it might raise. The types appearing in the profile must be either predefined ones[6] or formal type parameters appearing before the function in the parameter list. The names of the parameters in the profile of the function will not be used in the DCM or DFM, as are those appearing in the profile of a function declared in a DCM. The exceptions listed in the formal function header specify their (formal) names and profiles. The types appearing in those profiles are as for the function profile; however the names of the parameters themselves are significant, since they can be used in the DFM, namely in handlers for those exceptions.

[6]The rationale for this is the need to recognize boolean functions which drive the **if. . . then. . . else. . . end if** construct.

These formal types, functions and exceptions can be used in the DCM and DFM as opaque imported objects. In particular the DCM may declare values of formal types, functions taking as parameters or returning values of formal types and possibly raising formal exceptions, and exceptions returning values of formal types. Formal functions may be called in bodies of functions defined in the DFM, and formal exceptions may also be caught.

Visibility rules; in a parameterized module are as if the formal parameters were imported at the beginning of the DCM. However formal parameter names are NOT visible outside the module (see section 11.4.2 below for more information).

11.3.4 Modules dependency graph and values initialization

If some block in module A (either in DCM or in DFM) imports some object from module B, then A is said to *depend* on B. To prevent any problem with circular definitions,[7] i.e. to insure that mutually recursive definitions appear in a unique module, it is required that this "module dependency" relation contains no cycle. [8] The library manager and OLGA compiler of the FNC-2 system check that this always the case.

As said in section 9.11, declared values are computed at run-time when the block in which they are declared is entered. This has a clear meaning for functions, but it is not as clear for other kinds of blocks. The case of values declared in productions, in AG phases and in AGs proper will be examined in chapters 13, 15 and 15 respectively. Computation of values declared in non-parameterized DCMs or at the outer level of non-parameterized DFMs is discussed here, while the case of values declared in parameterized modules will be presented in section 11.4.2 below.

The computation of these values is performed in an *initialization part* associated with each module which needs it. These initialization parts are executed at the very beginning of each application generated by FNC-2 , in an order derived from a topological sort of the module dependency graph.[9] More precisely, to each module is associated a specific initialization procedure in the target language. When FNC-2 is directed to construct a complete application involving several modules and attributes evaluators, the list of involved modules is determined and then sorted according to the dependency relation, so that the first module in this order depends on no other. Then this order is used to add, at the beginning of the "main" program of the application, a call to the initialization procedures. More details are given in *the* FNC-2 *Reference Manual.*

11.4 Importation Clauses and Rules

Objects declared in DCMs can be imported and used in other level 1 and 2 blocks. The first subsection discusses importation from non-parameterized modules, in particular the notion of qualification and implicit importations. The second subsection discusses importation from parameterized modules, in particular *instantiation* of a module, i.e. the matching of actual and formal module parameters, and type checking of imported objects involving module parameters.

[7] such as object X of module A being defined in terms of object Y of module B and vice-versa.

[8] Note that a circularity in the module dependency graph does not imply the existence of a circular definition however the non-circularity of this graph is a sufficient condition to prevent those circular definitions, and is simpler to check.

[9] This is another reason to require it be non-circular.

11.4.1 Importation from non-parameterized modules

Syntax:

```
<import clause>       =  from <module id> import all ;
<import clause>       =  from <module id> import <import id +>;
<import clause>       =  import <module id>;
<module id>           =  %ID ;
<import id +>         =  <import id +> "," <import id>;
<import id +>         =  <import id>;
<import id>           =  <function id>;
* <id> designates a previously declared identifier
<id>                  =  <ID>;
<id>                  =  <ID> "#" <function id>;
* The <ID> above should be a <module id>, but not LALR(1)...
```

The first two forms of import clauses make visible, at the point where they appear, respectively all and part of the objects declared in the specified non-parameterized DCM. The names of these objects are thereafter visible in the same conditions as locally-declared names. The third form of import clause states that the objects declared in the specified DCM may be accessed thereafter using a *qualified* reference with the # connector.

Regarding visibility rules, an "unqualified" import clause, i.e. one of the first two forms, acts as the declaration of the specified names. Name conflicts (double declarations) and hiding of names declared in enclosing blocks occur as with other, "normal" declarations. With the first form of import clause, all the objects declared in the specified DCM, including objects imported into the DCM, are made visible. If some name designates several objects—e.g. a type and a function, or an overloaded function—, all these objects are imported together. With the second form of import clause, only a subset of the names exported by the specified DCM are made visible; it is not an error to (try to) import a name not declared in the DCM: the OLGA compiler will emit a warning, and the name will not be made visible.

If an imported and visible object references a non-imported object—e.g. a function is imported but one of the types appearing in its profile is not imported—, then that latter object is implicitly imported to make type checking still possible, but it will not be made visible nor accessible using a qualified reference. This applies in particular to enumerated types and their constants, i.e. it is possible to import an enumerated type but not some of its constants and vice-versa.

It is allowed to import and make visible more than once the same name from the same module into the same block. In particular, if DCM M_1 imports object A from DCM M_2, it is possible to import A into some block both from M_1 and M_2 (explicitly or implicitly). This is the only exception to the non-multiple declaration rule.

The third form of import clause imports all the objects from the specified DCM (including imported ones), but those objects are not made visible: they must be accessed using a qualified reference. This allows to avoid name conflicts. Thus qualified names may be used anywhere a visible name can be used. However if an operator function is thus imported, it can be used only with the "normal function" syntax, i.e. it is allowed (and actually mandatory) to write expressions of the form

```
M#+ (x, y)
```

but not

```
x M#+ y
```

Provided the relevant import clauses are present, it is possible to access the same object either with a non-qualified reference or with a qualified one. Also, if the same object is imported in qualified form from two different modules (explicitly or implicitly), any of the module names can be used to access it.

Importation clauses may appear as any other declaration in level 1 and level 2 blocks.

11.4.2 Importation from parameterized modules

Syntax:

```
<import clause>      =  from <module id> "(" <DCM actual parm +> ")"
                                 import all ;
<import clause>      =  from <module id> "(" <DCM actual parm +> ")"
                                 import <import id +>;
<import clause>      =  import <ID> "="
                                 <module id> "(" <DCM actual parm +> ")";
<DCM actual parm +>  =  <DCM actual parm +> "," <DCM actual parm>;
<DCM actual parm +>  =  <DCM actual parm>;
<DCM actual parm>    =  <id>;
<DCM actual parm>    =  <operator>;
```

The three clauses described above are similar in essence to those for non-parameterized modules but provide for passing actual types and functions parameters.

The matching of actual and formal parameters is as follows. First the actual and formal lists must have the same length. Then each actual parameter, in left-to-right order, is matched with its corresponding formal parameter. An actual parameter is a (qualified or visible) name which must be accessible at the point where the import clause appears.

If the formal parameter is a type, then the actual parameter must designate a type. The formal type name is then bound to that actual type in the sequel of the processing.

If the formal parameter is a function, then the actual parameter must designate a (possibly overloaded) function. The profiles of the actual and formal functions must be the same, taking into account that formal types in the profile of the formal function have been bound to actual types. This should bind the formal function to a unique actual one. The exception lists of the actual and formal functions are then examined together from left to right each formal exception is then bound to an actual one, provided their profiles match as explained for functions profiles. If any of these matches fails, the whole import clause fails.

If every match succeeds, a new non-parameterized module (DCM+DFM) is virtually constructed; this is called the *instantiation* of the module. The actual parameters are "imported and renamed" in this new virtual module. The objects declared in this new module are then imported into the importing block as described in previous section. However this explanation is only an approximation of reality because of the following reasons:

- Although importation is transitive through DCMs, formal parameters are not imported; rather the objects declared in the parameterized DCM and referring to formal parameters are "translated" to refer to actual parameters before being imported; since these actual parameters are accessible at the importation point, this causes no visibility problem. For instance, consider a DCM M_1 parameterized by type X, which declares a function A with result type X. If M_1 (Y) is imported into some block where Y designates a type accessible in the block, then A will be imported with result type Y, but name X will not exist at all.

- Considering actual parameters as being imported into the newly constructed virtual module which exports back other declared objects would violate the rule forbidding circularities in the module dependency graph.

- It is implementation-defined whether an actual copy of the parameterized module is created for each instantiation or whether the same "polymorphic" object code works for several different instantiations.

To avoid repeating the instantiation parameters at each qualified reference, and because the syntax accepts only single identifiers as module names in qualified references, the third form of import clause provides for associating a new local name with the instantiated module. This name must be used instead of the actual module name and parameters in qualified references. This name belongs to the "compilation units" name space, but is local to the block and follows usual visibility rules; this is the only case in which that name space is not considered as flat. Also, if the instantiation takes place in a DCM, this new module name is exported with the other identifiers and may be imported into another block and used to access the objects of the instantiated module.

If it is intended to use both qualified and non-qualified references to objects from a parameterized module, then the third import clause, specifying the local name of the instantiated module, must appear first, and the selected names for which a non-qualified reference is desired will be imported from that "local" module, using the second form of import clause of previous section.[10]

It is quite possible to instantiate in the same block a same parameterized module with different actual parameters. In this case the import clauses will generally use the third form because of name conflicts.

A parameterized DCM or DFM may itself instantiate another parameterized module. It may even pass some of its own formal parameters as actual parameters.

When a parameterized module declares a value to be computed at run-time, there will be one initialization; part for each instantiation, to be executed when the importing block is entered (whatever this means, see section 11.3.4). In fact, declared values of a parameterized module can be considered as local to the block(s) where this module is instantiated.

11.4.3 Importing an Asx into Olga

An Asx grammar can be imported[11] into an Olga specification (level 2 block).

[10] It is also allowed to use the **import all** clause, but since this makes all the objects visible, qualified references become useless...

[11] Though Asx and Olga are two distinct languages, conceptual similarities (and proper design and implementation) make it possible; this Asx-to-Olga importation is a recent development, fom Version 1.8.

An *import* clause can specify the importation of an Asx grammar (attributed abstract syntax, see chapter 12) into a declaration or/and a definition module.

Chapter 12

Attributed Abstract Syntaxes

With this chapter begins the description of OLGA features which make it an attribute grammar description language.

12.1 Generalities

With FNC-2 —and OLGA, as far as linguistic aspects are considered—, attribute evaluators are considered as tree-to-tree mappings, taking as input some decorated tree and returning zero, one[1] or several decorated trees. This gives the following advantages:

- this enhances modularity, since an AG writer may divide a large job into several smaller AGs which are "piped" into a sequence;
- this increases versatility since it is then possible to add a new evaluation pass—e.g. an optimization pass—between two existing ones, provided of course that the input and output tree structures agree;
- evaluators generated as tree-to-tree mappings are easier to integrate with modules generated by other tools, since trees are now widely used as intermediate data structures.

The decorated trees which are the input and outputs of an attribute evaluator are described by *attributed abstract syntaxes* (AASes). The latter are defined in separate compilation units which play the role of declaration modules w.r.t. a given AG, considered as a definition module defining an attribute evaluator.

The syntactic structure of the trees on which an attribute evaluator generated by FNC-2 operates is described by an *abstract syntax* rather than a concrete one (such as a BNF specification used by a parser generator (e.g. SYNTAX) or the FNC/ERN system). This has two main advantages:

- the abstract syntax is free from clumsy syntactic constraints due to the parsing method which is used by the parser generator, such as operators precedence and the "dangling else" problem in the LR parsing method, and expresses in a cleaner way the nesting of semantic constructs;

[1] This is the usual case.

- abstract trees are generally smaller than concrete ones because they don't contain keywords and other textual information.

The formalism used in OLGA to describe abstract trees is very close to Metal [KLM+83], the one used in the Mentor programming environment developed at INRIA. It provides for defining *phyla* and *operators*, which are respectively similar to non-terminals and productions of a usual context-free grammar. Operators are divided in two classes: fixed-arity, heterogeneous ones, which may have different phyla as operands, but have a fixed number of them, and variable-arity, homogeneous ones, the operands of which belong to the same phylum but in variable number for different instances (nodes). The latter are also called *list operators*, and their instances in a tree are *list nodes*. This is sufficiently versatile to describe easily most languages, while keeping consistency checks (e.g. in tree construction actions or tree transformation patterns) easy to perform.

The way in which these trees are produced is not specific to either OLGA or the FNC-2 system. An attribute evaluator described in OLGA and generated by FNC-2 takes as input such a tree, maybe partially decorated, and returns either the *same* tree decorated by different information or zero, one or more other decorated trees. However the FNC-2 system provides facilities for linking with two "tree constructors": SYNTAX, in which trees are constructed during the syntactic analysis of a source text, and Mentor, in which trees are constructed interactively in a syntax-directed fashion. See the relevant appendices of this volume for more information. The input trees of an evaluation phase may also be generated as an attribute value by a previous evaluation phase (see chapter 14), or be the result of a tree transformation pass.

AASes may share, by means of import clauses, some phyla, operators and attributes. This is especially useful to describe evaluation phases which do not construct new trees but merely add new information (attributes) to their input tree. In this case the input and output AASes share the whole underlying abstract syntax, and may even share some attributes.

12.2 Operators and Phyla

The abstract syntax of a language is made of *operators* and *phyla*.

The *operators* label the nodes of the abstract trees, and represent *features* of the language such as operators, conditional statements, etc. Operators come in two kinds: fixed arity and list operators. Operators with arity zero are the leaves of abstract trees and represent atoms of the language. Operators of fixed, non-null arity can have offsprings of different kinds, whereas all offsprings of list operators must be of the same kind. The meaning of "kind of offspring" is formalized by the concept of *phylum*.

The phyla are disjoint[2], non-empty sets of operators which represent *notions* of the language, e.g. variables, expressions, statements, declarations... To each offspring position of an operator is associated a phylum. This phylum contains precisely those operators that are allowed as root-operator of a subtree in this offspring position. Thus each occurrence in an abstract tree is associated with a phylum which indicates which operators are allowed at this location. However a given phylum may be associated with different offspring positions of different operators. In Pascal for instance, the *type* in a type declaration and the *type* in

[2]but see below the inclusion of a phylum into another one.

a variable declaration should probably be associated with the same phylum TYPE, because they correspond to the same notion of type constructors and have the same syntactic form.

12.3 Defining an Attributed Abstract Syntax

To define an abstract syntax one must describe operators, phyla and attributes. For operators, this involves defining their arity and the phyla associated with their sons. For phyla, this means defining which operators they include. For attributes, this means defining the phyla to which they are attached and their type. This section describes the Olga sublanguage[3] to be used for those definitions; this sublanguage is named Asx.

12.3.1 Lexical Structure

Operators and phyla are designated by Olga *identifiers*, as defined in chapter 6. Operators and phyla names can be written in any case, however we advise—and this will be so in all examples of this volume—to use lowercase names for operators and uppercase names for phyla. Phyla and operator names belong to separate name spaces—respectively that of types and that of functions—and thus do not normally conflict; more details are however given further in this and next chapters.

The lexical structure of attribute names is defined in section 6.8.

The keywords of Asx are the following:

all end from grammar import is root $scx

The other keywords of Olga, including **$arity** and **$position**, are *reserved words* and cannot be used as operators, phyla or attribute names.

The other lexical units of Olga abstract syntax sublanguage are the following operators and separators:

`+ * -> = ; , ( ) : # +=`

and comments as in Olga.

12.3.2 Syntactic Description

The definition of an abstract syntax begins by a header specifying its name (see next section). Then comes the definition of operators, phyla, and attributes, which can be interleaved in any order convenient for the reader (as can be done for rules of BNF definitions of concrete syntaxes).

<u>Syntax</u>:

```
<abstract syntax>     =  <header>
                            <import *>
                            <definition +>
                         end grammar ";";

<header>              =  grammar %ID is ;
```

[3]Viewing Asx as a sublanguage of Olga allows to automatically relate concepts which are the same in both languages. However, from an implementation and a definition point of view, they are separate languages. The importation of an Asx grammar into an Olga one has been made possible recently.

```
<import *>              =  <import *>
                           <import>;
<import *>              =  ;
<import>                =  import grammar %ID ";";
<import>                =  from %ID import all ";";
<import>                =  from %ID import <import ids +> ";";
<import>                =  import %ID ";";
<import ids +>          =  <import ids +> "," <import id>;
<import ids +>          =  <import id>;
<import id>             =  %ID ;

<definition +>          =  <definition +>
                           <definition>;
<definition +>          =  <definition>;

<definition>            =  <phylum def>;
<definition>            =  <operator def>;
<definition>            =  <attribute decl>;
<definition>            =  <root def>;

<root def>              =  root is %ID ";";

<attribute decl>        =  %ATTR "(" <phylum, +> ")" ":" <type id> ";";
<type id>               =  %ID "#" %ID ;
<type id>               =  %ID ;
<phylum, +>             =  <phylum, +> "," <phylum-name>;
<phylum, +>             =  <phylum-name>;
<phylum-name>           =  %ID ;

<phylum def>            =  %ID "=" <op-or-phy *> ";";
<phylum def>            =  %ID "+=" <op-or-phy *> ";";
<op-or-phy *>           =  <op-or-phy *> <op-or-phy>;
<op-or-phy *>           =  ;
<op-or-phy>             =  %ID ;

<operator def>          =  %ID "->" <phylum> "+" ";";
<operator def>          =  %ID "->" <phylum> "*" ";";
<operator def>          =  %ID "->" <phyla *> ";";
<phyla *>               =  ;
<phyla *>               =  <phyla +>;
<phyla +>               =  <phyla +> <phylum>;
<phyla +>               =  <phylum>;
<phylum>                =  %ID ;
```

12.3.3 Abstract Syntax

The meta-operators `+` and `*` are used to define respectively never-empty and possibly-empty list operators. As stated before, list operators are homogeneous, which accounts for the unique phylum identifier in the right-hand side of their declarations. All the offsprings of such operators, whatever their number, must belong to that unique phylum.

The other operator declarations define fixed-arity operators. The (possibly empty) list of phyla names in their right-hand side specifies both the number of offsprings and their associated phyla, in the same order.

Phyla declarations specify the list of operators which belong to it. A given operator should belong to exactly one phylum. However it is not forbidden that two operators have the same *profile*, i.e. their declarations have the same right-hand side.

It is also allowed to *include* one phylum into one (or more) other phyla; this is achieved by listing the name of the included phylum in the RHS of the definition of the including phylum. This has the effect of declaring an anonymous operator belonging to the including phylum with one offspring which is the included phylum. This operator will never appear in the trees described by the AAS but, apart from that, it has concrete existence; in particular, semantic rules must be attached to it in any AG based on the AAS (see chapter 13).

The declaration of any phylum `PH` involves the implicit declaration of the nullary operator `null-PH`. This operator is used in two contexts:

- when the tree is constructed during parsing—which means that a tree construction specification has been associated to the concrete syntax—, and when a severe, uncorrectable syntactic error occurs, a whole portion of text is skipped until the parser can recover; then this incorrect portion of text is assumed to be derivable from some non-terminal, which is associated to some phylum `PH`; the "tree" constructed for this portion of text is a leaf labelled by the operator `null-PH`;

- when the tree is constructed interactively by successive derivations, the operator `null-PH` corresponds to yet unexpanded portions of trees which must be labelled by an operator belonging to the phylum `PH`.

In this way, trees which are input to an attributes evaluator are always correct w.r.t. the abstract syntax, even in case of syntactic errors or incomplete trees. Note that it is always possible to define other nullary operators for any phylum, if this has a semantic meaning; it is also possible to explicitly construct trees with these `null-PH` operators when this proves useful: for instance the null statement of Pascal may correspond to the `null-STMT` operator in an abstract syntax where `STMT` is the phylum gathering all the statements.

See section 12.3.5.3 for the second form of phylum definition, using `+=`.

The *root phylum*; is specified in a root definition; there must be exactly one root definition in each AAS specification, even if it imports (see further) another AAS. Any operator labelling the root of a tree to be input to an attributes evaluator generated from an attribute grammar based on the currently defined AAS must belong to that phylum. It can not appear in the right-hand side of any operator declaration. It may not have any inherited, global or compound attribute in any attribute grammar based on that AAS (see chapter 13).

An OLGA abstract grammar should be *reduced*, in the same way as concrete grammars input to most parser generators such as SYNTAX. This means that, on one hand, all phyla should be reachable from the root phylum—i.e. there must exist at least one tree derivable from the root phylum in which a given phylum appears—and, on the other hand, they should all be productive—i.e. among all the trees derivable from a given phylum there should exist at least one in which all the leaves are labelled by zero-ary operators (excluding the implicitly defined `null-PH` operators). As a special case, the latter is not required for empty phyla, i.e. those in which definition appears no operator those empty phyla are assumed to be defined in a separate AAS which will import the current one (see below). However such an AAS cannot be directly used in an attribute grammar.

Apart from that, and the convention regarding the declaration of the root phylum, operators and phyla declarations can be interleaved in any order convenient for the reader. There is no need to define an abstract syntax symbol before it is used; this is anyway necessary since most of those definitions are inherently recursive. There is no provision for structuring the specification of an abstract syntax except by inserting meaningful comments at appropriate places.

Tree construction functions and tree matching patterns (see chapter 14 and A) for list nodes use names which are implicitly derived from the names of list operators by adding suffixes; the Asx compiler will check that these implicitly defined names do not conflict with other actual names. Such name conflicts are not true errors, but they will prevent the use of tree construction and tree matching operations, which can be rather annoying.

12.3.4 Attributes

An attribute is defined by its name, the list of phyla to which it is attached and its (Olga) type. The latter must be either a predefined Olga type (see section 7.3) or imported from a separate module. In that latter case, it must, may or must not be qualified according to the way the corresponding import clause(s) is(are) written (see chapter 11 and below). Note however that it is NOT possible to instantiate a parameterized module in an AAS; a separate module must be written to perform the instantiation, and the desired types will be imported from that module into the AAS.

A given attribute may appear in several attribute definitions, provided that its type (name) is the same in all of them. The corresponding lists of phyla are merged. It is not an error to cite the same phylum in more than one definition of the same attribute. This allows to add an imported attribute to newly declared phyla (see below).

Attributes definitions may be interleaved with phyla and operators definitions in any convenient order.

The attribute names **$arity**, **$position** and **$scx** are reserved and must not appear in an attribute definition. The latter has type *source-index* and is implicitly attached to every phylum; its value is set at tree construction time (see chapter 14 and appendices A and B). The former are "syntactic attributes" which are available on list nodes and sons of list nodes respectively, and may be used only in semantic rules (see chapter 13).

12.3.5 Modularity

All import clauses, if any, must appear at the beginning of an AAS specification.

12.3.5.1 Type names importation

The type names used in an AAS specification must be imported from other modules, since Asx does not provide for type definitions. The corresponding import clauses which may be used are those of OLGA and have the same semantics, except that importation is restricted to type names and that it is forbidden to instantiate parameterized modules.

Type names do not conflict with operators and phyla names in an AAS, because it is syntactically possible to discriminate them, but when an AAS is used as an output syntax of an AG, phyla names become type names and may thus cause conflicts.

12.3.5.2 Attributed abstract syntaxes importation

The clause **import grammar** `X ;`, where `X` designates a separate AAS, has the following semantics:

- All the import clauses of `X` are textually included at the importation point. For type names importation, this may cause name conflicts with other subsequent import clauses; however, as for ordinary OLGA, it is not an error to import more than once the same name from the same module. For grammar importation the currently described process is recursively repeated.[4]
- All the operators definitions of `X` are textually included.
- All the phyla definitions of `X` are textually included but with a semantics similar to the use of the `+=` operator (see below), so that new operators may be added to those phyla, either explicitly or implicitly in other **import grammar** clauses.
- All the attributes definitions of `X` are textually included.
- The root definition of `X` is NOT included.

12.3.5.3 Phyla augmentation

New operators may be added to existing phyla using the `+=` form of phylum definition. If the phylum name is already known, be it implicitly from an **import grammar** clause or explicitly from an `=` phylum definition, the operators are simply added to that phylum. Otherwise the `+=` definition reduces to an `=` definition.[5]

Note that augmenting an imported phylum leads to a *different* type as the original imported phylum, although they have the same name. This can cause name conflicts if a same subgrammar is imported into two AASes which will be used as output AASes of the same AG (see chapters 13 and 15): if the subgrammar is imported untouched into both AASes, then the common phyla will actually be the same, but if one AAS augments some of its phyla after importation, then the modified phyla will be different from the original ones, and it will be necessary to use qualified names to distinguish them (see chapter 15).

12.3.5.4 The resulting AAS

The resulting AAS is composed of all the implicit and explicit definitions. The reduceness test—with the same relaxed condition on empty phyla—and the test for implicit list operators names conflict are performed on this new AAS.

[4] Actually the tables generated from an AAS specification contain all the definitions in their expanded form, so this recursive process is only a “vue de l'esprit”.

[5] So, in case of doubt, using the `+=` form is harmless.

The name specified in the header identifies the current AAS. It belongs to the "compilation units" name space; and thus does not conflict with phyla, operators and types names. It can be used in other **import grammar** clauses and/or in attribute grammars to specify their input or output domains (see chapters 14 and 15).

12.3.5.5 Uses and examples

The main use of **import grammar** clauses is for defining the input and output AASes of an AG which merely adds new information to its input tree without producing a different tree. In this case the input and output AASes share the same underlying syntactic structure but have different attributes; this underlying syntax may[6] be written as a separate attributeless abstract syntax which will be imported in the input and output AASes, which will themselves be reduced to attribute definitions. It may even be the case that some attributes also be shared. This is exemplified in appendix A.

Import grammar clauses may also be used for other purposes, e.g. augmenting an existing grammar, sharing a subgrammar, etc. Note however that each subgrammar must be reduced, except maybe for empty phyla. Although the only real situation in which a grammar must actually be reduced is when it is used as an input or output AAS of some attribute grammar—FNC-2 will detect this and reject the grammar if necessary—, performing a reduceness test on each subgrammar allows to detect some errors more quickly.

[6] This is in fact mandatory.

Chapter 13

Attributes and Semantic Rules

13.1 Introduction

Attribute grammars are a very high-level, declarative method to specify any computation on trees. Since most trees are described by some grammar, this is achieved in a syntax-directed manner by attaching attributes to the nodes of the input tree and semantic rules to productions. These semantic rules express how to compute some attributes of a production in terms of the others.

An attribute grammar, as said in previous chapter, is seen as specifying a tree-to-tree mapping, i.e. a function[1] taking a tree as input and producing zero, one or more trees as output. These decorated trees are specified by AASes as described in previous chapter.

Attributes appearing in the input AAS are named *given attributes* because they are available when the computation specified by the AG begins. They are considered as constants in the semantic rules. Attributes appearing in output AASes are named *result attributes*; they must be defined either by semantic rules or in the "actions" constructing the output tree(s) to which they are attached (see chapter 14). Other attributes are "local" to the AG and are named *working attributes*.

Two kinds of AGs are distinguished:

- the most frequent case, and the one which is closest to the original theory, is when there is only one output AAS which is the same as the input one but with different attributes, and when no tree construction action appears in the AG then the tree itself is not changed, result attributes must be defined by semantic rules,[2] and both working and result attributes participate in the computation; this kind of AG will be named *side-effect AG*,[3] because it produces no new result but has the side-effect of modifying its parameter (the input tree);
- the other case is when there is zero, one or more output AASes, and when tree construction actions appear; the output trees and their corresponding result attributes must be

[1] An AG describe a function in the Olga sense and this function must be declared in a declaration module.

[2] except those which also appear in the input AAS, which are considered as constants and must not be redefined.

[3] This term is actually rather misleading, because such an AG is actually a function from the input AAS to the output AAS such that, on one hand, both AASes are constrained to be based on the same syntax and, on the other hand and consequently, the input and output trees are deeply shared. However this sharing is not contradictory with the notion of function.

constructed as described in chapter 14; only working attributes participate in the computation, and the input tree is destroyed thereafter; the result of the evaluation is then an anonymous record containing the values of the (input and working) attributes of the root of the input tree, including the output trees and maybe other attributes; this kind of AG will be named *functional AG* because the results are quite separated from the input parameter, which is itself destroyed; note that it is allowed that one of the output AASes be the same as the input one.

Previous chapter described AASes. The rest of this chapter will describe working attributes and their declarations, and semantic rules. Chapter 14 describes the construction and decoration of output trees, while chapter 15 discusses the structure of an AG and its relation with the modularity concepts of OLGA.

13.2 Attributes

13.2.1 Attribute Declarations

Attribute declarations introduce attributes which are local to a given AG, i.e. those which do not belong to its input and output AASes. A working attribute has a name, a kind (e.g. inherited or synthesized), a type, and is attached to a set of phyla of the input AAS. If the "automatic attribute declarations" option of the OLGA compiler is enabled (see next section), those declarations may be incomplete or even completely omitted.

Syntax:

```
<attribute part>      =  attribute
                         <attribute decls>;
<attribute decls>     =  <attribute decls>
                         <attribute decl>;
<attribute decls>     =  <attribute decl>;
<attribute decl>      =  <attribute kind> %ATTR "(" <phyla*> ")"
                               <type constraint> ";";
<attribute decl>      =  compound %ATTR and %ATTR "(" <phyla*> ")"
                               <type constraint> ";";
<attribute kind>      =  synthesized ;
<attribute kind>      =  inherited ;
<attribute kind>      =  global ;
<attribute kind>      =  ;
<phyla*>              =  ;
<phyla*>              =  <phyla+>;
<phyla+>              =  <phyla+> "," <phylum>;
<phyla+>              =  <phylum>;
<phylum>              =  %ID ;
```

As defined in the original theory of attribute grammars [Knu68], and also in OLGA, a synthesized attribute propagates information bottom-up from the leaves of a tree to its root; this means that in a given production, synthesized attributes of the right-hand side symbols are *input attributes* which must be computed elsewhere (in the offspring productions), whereas synthesized attributes of the left-hand side symbol are *output attributes* which must be computed in that production. Conversely, an inherited attribute propagates information top-down from the root of a tree to its leaves; this means that in a given production, inherited attributes of the left-hand side symbol are *input attributes* which must be computed elsewhere (in the parent production), whereas inherited attributes of the right-hand side symbols are *output attributes* which must be computed in that production. Be careful not to mistake given, working and result attributes on one hand, and input and output attributes (occurrences) on the other hand.

A global attribute behaves in every respect as an inherited attribute except when generating missing semantic rules (see below). The declaration of a compound attribute actually declares two attributes—there are two names in the decl—, the first one being a synthesised one and the second being an inherited one. Both these attributes behave in every respect as named (synthesized and inherited) attributes, except that they have the same type, are attached to the same phyla and have special rules for generating semantic rules (see section 13.8). Any or both attributes may be given[4] and/or result attributes however, since they must be attached to the same phyla, if the input and output AAS differ, and if one is a result attribute the other cannot be a given attribute and *vice versa*.

An attribute declared with no specified kind will have it determined by its first textual use (see next section).

The lexical form of an attribute name is described in chapter 6. Some of those names are reserved for predefined attributes (see section 13.3). Names of given and working attributes conflict, because they will be attached to the same abstract syntax. If an output AAS is based on a different abstract syntax as the input one—which means that explicit tree construction and decoration are used (see chapter 14)—, then given and working attribute names do not conflict with the names of result attributes of this output AAS. When the tree is not changed, which implies that all attributes (given, working and result) are attached to the same AS, then their names potentially conflict. In particular, if a same attribute appears in both the input and output AAS, this means that it is transmitted unchanged and that it cannot be redefined but only used; this of course applies only on the phyla to which this attribute is attached in both the input and output AASes.

After the attribute name comes the list of phyla to which this attribute is attached. Those phyla must belong to the input abstract syntax. If the compiler option is enabled, this list need not be complete and may even be empty.

If a type constraint is specified, the attribute will have that type. If the type constraint is a subrange type, then this has the additional effect to insert a range test in each semantic rule defining that attribute. If no type constraint is specified, and if the compiler option is enabled, the type of the attribute will be determined by type inference upon its first textual use (see next section).

A same attribute may appear in several attribute declarations, provided that these declarations are "compatible" regarding the type constraint—i.e. if more than one declaration specify a type constraint, then they must be the same type—and the kind—as for the type constraint.

When a phylum is included in another one in the sense of chapter 12, there need not be any

---

[4]It is nevertheless extremely rare to have compound attributes as given ones, because the only interest of compound attribute is for generating missing semantic rules, and it is forbidden to give semantic rules for given attributes.

relationship between the attributes attached to both these phyla, since they are considered as completely separate in the AG.

If the "automatic attributes declarations" option of the OLGA compiler is not enabled, all the attribute declarations must be present and complete, which means that they must specify:

- the kind of the attribute;
- the attribute name;
- the *complete* list of the phyla to which it is attached;
- the attribute type.

If that compiler option is enabled, any part of such a declaration except the attribute name may be omitted or incomplete, or the declaration itself may be completely omitted (but see the note at the end of this section). The default declaration rules described in next section will be used to synthesize part or all of attribute declarations.

If the input and output AAS are based on the same abstract syntax and the tree is unchanged, then result attributes must be defined by semantic rules. Accordingly they must have a kind. This kind can be automatically determined if the compiler option is enabled, else it must be specified using a reduced form of attribute declaration in which only the name and kind of the attribute is given. This is the only exception to the "complete declaration" rule.

13.2.2 Default attributes declarations

This section describes the processing performed by the OLGA compiler on each *attribute occurrence* if the "automatic attribute declarations" option is enabled. An attribute occurrence is the association of an attribute name and a symbol (operator or phylum) occurring in a production, and are more fully described below. The definition of *input* and *output* and *non-local* attribute occurrences also appears in that section. Automatic attribute declarations does not apply to non-local attribute occurrences.

Upon encountering an attribute occurrence, the following process occurs:

- The attribute name is searched to check whether it has already been declared or not. If it is declared as a given or result attribute, the rest of the process resorts to simple checks (because the complete declaration given in the AAS cannot be modified). If the attribute was not previously known, it is declared with no type and no kind and is attached to no phylum.
- The symbol specified in the attribute occurrence is then examined. If it is an operator, then the phylum it belongs to is retrieved, and the attribute is attached to that phylum if this was not already done. Otherwise, the symbol is a phylum and the attribute is attached to this phylum if this was not already done. If the attribute at hand is a component of a previously declared compound attribute, then the other component is also attached to the given phylum.

- If the attribute has no kind yet, it is given a kind depending on whether the attribute occurrence is the left-hand side of a semantic rule (output occurrence) or is in the right-hand side (input occurrence), and whether the specified symbol is the left-hand side of the production or is in the right-hand side. If the occurrence is an output one and the symbol is the left-hand side of the production, or if the occurrence is an input one and the symbol is in the right-hand side of the production, the attribute is declared as synthesized, otherwise it is declared as inherited. If the attribute at hand is a component of a previously declared compound attribute and is found to be synthesized, then the other component will be inherited and *vice versa*. Note that the automatic declaration process will *never* declare itself a compound attribute, because it has no mean to guess such a relationship between two separate attributes.

- If the attribute has no type yet, a fresh type variable is used instead and type inference is performed on the whole semantic rule as described in chapters 8 and below. The result of type inference must be so that the type variable be bound to one and only one *actual* (i.e. not polymorphic) type. This actual type then becomes the type of the attribute. If the attribute at hand is a component of a previously declared compound attribute, the other member will be given the same type.

The list of attribute declarations inferred by the OLGA compiler through this process is written down in the listing of the compilation, in the same format as expected on input this allows the grammar writer to "cut" them from the listing and "paste" them back into the source of his attribute grammar, provided of course that they please him.

Note that the "automatic attributes declarations" option is disabled by default. In that case the above process is not performed, rather each attribute occurrence is checked to see whether the attribute name has been declared, whether it is actually attached to the specified phylum, whether the kind of the occurrence—input or output one—determined by the combination of the kind of the attribute and the position of the symbol in the production corresponds to its position in the semantic rule, and the declared type is used in type checking the semantic rule.

Those default declaration rules were borrowed from the ancestor of FNC-2 , the FNC/ERN system [Jou84]. Experience gained using FNC/ERN proves that they are very valuable when developing new attribute grammars from scratch: the author writes the first version of his grammar without worrying about attribute declarations, compiles it with the "automatic declarations" option enabled, and pastes the generated declarations into his grammar. Subsequent compilations will be performed with that option disabled, in order to let the compiler make more extensive consistency checks.

13.2.3 Predefined attributes

A number of attributes, bearing lexical and/or syntactic information, are predefined for each abstract syntax. Those predefined attributes cannot be redefined, and their names are reserved words (keywords). Since they behave as given attributes, their value is available when the current evaluation pass starts also, they are implicitly attached to every AAS.

Every phylum of every abstract syntax has the attribute **`$scx`** of type `source-index`, whose value is set by the scanner and/or the parser/tree-constructor, and is the position in the source text of the first character of the terminal sentence derived from that node. Regarding trees constructed during attributes evaluation passes, this attribute behaves as a regular result one and needs to be defined on those new trees (see chapter 14); however it has a default definition.

The other two predefined attributes are special in that they have a purely syntactical meaning and are only defined on nodes of list productions (see below). Let `op -> PH*` or `op -> PH+` be such a production. Then the operator `op` has the attribute **`$arity`** of type `int`, whose value on each of its instances is the number of offsprings of that instance. The phylum `PH` has the attribute **`$position`** of type `int` its value is 1 for the leftmost son, 2 for the second leftmost son, etc., up to **`$arity`**`(op)` for the rightmost son. See below for more details. Note however that **`$arity`** is *not* attached to the phylum to which `op` belongs, and that **`$position`** is *not* attached to other occurrences of `PH` or its members, except of course in other list productions. Regarding trees constructed during attributes evaluation passes, these attributes do not need to be defined on those new trees: their value is set implicitly by the tree construction operations. In fact those "attributes" have the shape of attributes but represent a completely different notion; maybe a different syntax would make the things clearer.

13.3 Generalities about Semantic Rules

Semantic rules are attached to each production of an abstract syntax and specify the values which will be assigned to the *output attribute occurrences* of that production, i.e. the synthesized attributes of the left-hand side symbol and the inherited (and alike) attributes of the right-hand side symbols. Those values are defined by an expression, perhaps involving as atomic operands some *input attributes occurrences* of the production, i.e. the inherited attributes of the left-hand side symbol, the synthesized attributes of the right-hand side symbols, and the given attributes of all symbols. The designation of some attribute attached to some symbol of a production is called an *attribute occurrence*; productions and attribute occurrences are defined in section 13.4. This section also describes how some non-local attribute occurrences may be referred in a production and be considered as input attributes occurrences.

Some attribute grammar systems allow atomic operands in the right-hand side of semantic rules to be any attribute occurrence, i.e. output ones as well as input ones. Restricting them to be only input attribute occurrences makes the attribute grammar in *normal form* [Boc76]. We enforce this restriction because, on one hand, it makes the generation of evaluators much simpler and, on the other hand, it is not a strong restriction in OLGA due to the "declared values" facility.

Semantic rules for fixed-arity productions are defined in section 13.5. This is a direct adaptation of the classical theory of attribute grammars [Knu68] which was based on concrete grammars, where productions always have a fixed number of symbols in the right-hand side. However, variable-arity (list) productions need some special constructs because this number of symbols is not statically known. Semantic rules for list productions are defined in section 13.6. Other expressions which may appear in semantic rules are presented in section 13.7.

OLGA and its compiler have a mechanism which can automatically generate some semantic rules when they have not been specified by the grammar writer. This mechanism is described in section 13.8.

13.4 Productions and Attribute Occurrences

Productions in OLGA are just the abstract syntax counterpart of concrete grammars productions, i.e. the specification of an operator and of its offsprings. Note that such a specification is not a new one, but just a reminding of what has been specified in the definition of the input AAS (see chapter 12). The left-hand side operator of a production and its right-hand side phyla are collectively designated as the *symbols* of that production.

Attribute occurrences allow to designate some attribute attached to some symbol of a production. This involves the specification of the attribute name and the specification of the (variable) symbol(s). Note that this notion of attribute occurrence applies only to given and working attributes, and to result attributes when the output tree is the same as the input one. Chapter 14 deals with other cases.

With a production comes the list of its associated semantic rules. A production is also a level 1 block in which local declarations may appear.

Syntax:

```
<production group>    =  where <production> use
                            <sem rules>
                         end where ";";
<production group>    =  where <production>
                         declare
                            <decl +>
                         use
                            <sem rules>
                         end where ";";
<production>          =  <ID> "->" <phylum> "+";
<production>          =  <ID> "->" <phylum> "*";
<production>          =  <ID> "->" <phyla *>;
<phyla *>             =  ;
<phyla *>             =  <phyla +>;
<phyla +>             =  <phyla +> <phylum>;
<phyla +>             =  <phylum>;
<sem rules>           =  <sem rules>
                         <sem rule>;
<sem rules>           =  ;
<sem rule>            =  <attrib occ> ":=" <expression> ";";
<expression>          =  <attrib occ>;
<attrib occ>          =  <attrib name>;
<attrib occ>          =  <attrib name> "(" <symbol> ")";
<attrib occ>          =  <attrib name> "[" <up symbol +> "]";
<attrib name>         =  %ATTR ;
<attrib name>         =  $scx ;
<symbol>              =  <ID>;
<symbol>              =  <ID> "." <qualifier>;
<up symbol +>         =  <up symbol +> "," <up symbol>;
<up symbol +>         =  <up symbol>;
<up symbol>           =  <ID>;
```

A production group specifies a production and its associated semantic rules. The production must be the copy of an (explicit or implicit) operator definition of the underlying abstract syntax (see chapter 12). In addition, for each pair of phyla $\mathtt{PH}_1$ and $\mathtt{PH}_2$ such that $\mathtt{PH}_1$ is included in $\mathtt{PH}_2$, there exists in the AG the production

$\mathtt{PH}_2$ `->` $\mathtt{PH}_1$

which represents the corresponding anonymous unary operator mentioned in chapter 12. Such a production has exactly the same prerogatives as any other production, and semantic rules must be associated with it to define the synthesized attributes of $\mathtt{PH}_2$ and the inherited attributes of $\mathtt{PH}_1$.

An attribute occurrence designates some attribute of some symbol. The attribute is designated by its name; this name is either a user-defined one, a predefined one (**$scx** or others), or is not declared at all. In that latter case it is automatically declared if the corresponding compiler option has been enabled (see section 13.2.2). The symbol is designated by one of three means: if it is omitted, then the symbol in the left-hand side of the production (the operator) is assumed. Otherwise it must be designated by its name and an optional qualifier. If the symbol appears only once in the production, then its name suffices to designate it unambiguously; otherwise the qualifier is (generally) mandatory. Qualifiers are different for fixed-arity and list productions, and are described in the corresponding sections. An attribute occurrence specifying several symbol names between brackets is a non-local reference, which is described in section 13.5. If an operator and a phylum have the same name, and if they happen to appear in the same production, then qualifiers must be used to distinguish both the occurrence and the sort (operator or phylum) of the symbol.

The designated attribute must be attached to the designated symbol, that is, to the designated phylum or, if the symbol is an operator, to the phylum to which this operator belongs. If this is not so, and if the "automatic attribute declarations" compiler option is enabled, then this attachment is automatically realized (see section 13.2.2); otherwise it is an error.

Regarding type checking, the type of an attribute occurrence is that of the designated attribute. If the attribute was declared without type constraint, or if the attribute was not declared at all, then this type is determined by type inference (see chapter 8); however that type may not be polymorphic, and must be unique (that is, the type determined by type inference must be the same for all occurrences of that attribute in the grammar). See section 13.2.2 for more details.

Value declarations in a production whose definition involves input attribute occurrences are just like semantic rules; these values play the role of "local attributes", i.e. attributes attached to a production rather than to a phylum. More details appear in section 13.7.2 below.

13.5 Semantic Rules for Fixed-Arity Productions

Semantic rules for fixed-arity productions just specify the assignment of a value to each output attribute occurrence of the production; that value is determined by the evaluation of an expression, perhaps involving some input attribute occurrences.

Syntax:

```
<qualifier>          =  <INT>;
```

In fixed-arity productions, qualifiers are used to distinguish among several occurrences of the same phylum in the right-hand side of the production; those qualifiers are variable names.

A non-local reference has the form described in previous section and specifies an attribute name and a list of symbols (phyla and operators). It is a (dynamic) reference to the corresponding attribute of the first node, in the path from the node corresponding to the left-hand side operator of the production (excluded) to the root of the tree, which belongs to the list of symbols. It is an error if the OLGA compiler cannot determine statically, by analyzing the abstract syntax, whether that target node will always exist in some tree. The designated attribute must then be a given or inherited (or alike) attribute of that node.[5] Note that the automatic attribute declaration mechanism never applies to non-local attribute occurrences, even if the compiler option is enabled. Such a non-local attribute occurrence is considered in the current production as an input attribute occurrence. If an operator and a phylum have the same name, and if this name appears in some non-local reference, then it is assumed to refer to the phylum rather than to the operator.

The left-hand side of a semantic rule must be an output attribute occurrence of the production. The right-hand side of a semantic rule is an expression. Input attribute occurrences, including non-local and given attribute occurrences, may be used as operands in this expression. The type of the attribute occurrence being defined and the type of the expression must unify as defined in chapter 8. Note that the expression needs not be a top-level one if the attribute has a declared type the set of types of the expression will be restricted to that of the attribute, if possible. This means that the assignment operator `:=` can be considered, for all issues regarding type checking, as having profile `(@T, @T): void`, and must be a top-level "expression".

The dynamic semantics of such a semantic rule is that, on each node of the input tree which is an instance of the production, the value of the expression will be computed and assigned to the *attribute instance* corresponding to the attribute occurrence being defined. If the attribute is declared to belong to a subrange type, then the value to be assigned is checked to see whether it fits in that subrange. However—and *this* is precisely what makes the declarative character and the expressive power of attribute grammars—, the order in which the tree is traversed and the different attribute instances are computed is not specified; it is only known that an expression involving attribute occurrences—be it in a semantic rule or in a value declaration—will be evaluated only after all of those occurrences have been computed. The evaluator generator integrated in the OLGA compiler will determine, through a global analysis of the *dependencies* implied by the semantic rules, a suitable order for this traversal and these evaluations. See *the Theory of* FNC-2 for more details.

If an exception propagates up to the semantic rule level without being caught, then the whole attributes evaluation phase is aborted. This is because every attribute instance *must* receive a value. What happens next depends on the implementation language of the evaluator (see the corresponding chapters in *the* FNC-2 *Reference Manual*).

[5]This specification implies that all the phyla and operators listed in a non-local reference must have that same attribute. Experience will determine if this condition is too restrictive.

13.6 Semantic Rules for List Productions

The specification of attribute occurrences and semantic rules for list productions is somewhat more complicated because, since the number of offsprings of the production is not statically known, it is not possible to refer to each symbol and each attribute occurrence separately.

Syntax:

```
<attrib name>         =  $arity ;
<attrib name>         =  $position ;
<qualifier>           =  left ;
<qualifier>           =  right ;
<qualifier>           =  first ;
<qualifier>           =  last ;
<expression>          =  <case position>;
<case position>       =  case position is
                            <first son clause>
                            <last son clause>
                            <other son clause>
                         end case ;
<case position>       =  case position is
                            <last son clause>
                            <first son clause>
                            <other son clause>
                         end case ;
<case position>       =  case position is
                            <first son clause>
                            <other son clause>
                         end case ;
<case position>       =  case position is
                            <last son clause>
                            <other son clause>
                         end case ;
<first son clause>    =  first ":" <expression> ";";
<last son clause>     =  last ":" <expression> ";";
<other son clause>    =  other ":" <expression> ";";
<expression>          =  <case arity>;
<case arity>          =  case arity is
                            <fixed arity part>
                            <other arity clause>
                         end case ;
<fixed arity part>    =  <fixed arity part>
                         <fixed arity clause>;
<fixed arity part>    =  <fixed arity clause>;
<fixed arity clause>  =  <INT> ":" <expression> ";";
<other arity clause>  =  other ":" <expression> ";";
```

As for fixed-arity productions, input attribute occurrences are synthesized attributes of right-hand side symbols, inherited attributes of the left-hand side symbol and non-local and given attribute occurrences, whereas output attribute occurrences are inherited attributes of right-hand side symbols and synthesized attributes of the left-hand side symbol. Semantic rules for those two kinds of output occurrences will be described separately.

13.6.1 Inherited Attributes of Right-hand Side Symbols

The basic rationale for the following definitions is the desire to control the dependencies implied by the semantic rules so as to make the mapping to "standard" attribute grammars easy while not restricting the expressive power too much. The result is that in the semantic rule defining an inherited attribute of an offspring, only the left-hand side operator and the left and/or right brothers of that offspring, if any, are accessible. It was also required that the OLGA compiler performed as many consistency checks as possible, and in particular checked that, whatever the actual arity of an instance of a list production, each attribute instance has a unique definition.

Let a list production be of the form `op -> PH*` or `op -> PH+`. Then the offspring of that production under consideration is designated, in an attribute occurrence, by `PH`, while its left brother (if any) is designated by `PH.`**left** and its right brother (if any) is designated by `PH.`**right**.[6] An inherited attribute of such an offspring may be defined by several kinds of semantic rules:

- if this definition does not depend on the position of the offspring in the right-hand side of the production, and if it does not involve any attribute of other offsprings, then it will look like a semantic rule of a fixed-arity production, with `PH` designating the offspring under consideration;
- if the definition depends on the position of the offspring and/or it involves some attribute of its brothers, then the above defined visibility rules must be enforced using the **case position** construct.

This latter construct allows to specify and check separately the definition of an inherited attribute of an offspring according to its position: leftmost one, rightmost one or other. That definition for the leftmost son is the expression following the word **first**; the definition for the rightmost son is the expression following the word **last**; the definition for other sons is the expression following the word **other**. The **other** clause and at least one of the **first** and **last** clauses are mandatory. In a **first** clause, only attributes of the current son `PH`, its right brother `PH.`**right**, the father `op` and any non-local symbol may be referenced. In a **last** clause, only attributes of the current son `PH`, its left brother `PH.`**left**, the father `op` and any non-local symbol may be referenced. In an **other** clause, the symbols which may be referenced depend on which of the **first** or **last** clauses or both are present: in the first case, only attributes of the current son `PH`, its left brother `PH.`**left**, the father `op` and any non-local symbol may be referenced this is understandable since the **other** clause must also be valid for the rightmost son. In the second case, only attributes of the current son `PH`, its right brother `PH.`**right**, the father `op` and any non-local symbol may be referenced. In the latter case, attributes of the current son `PH`, its left brother `PH.`**left**, its right brother `PH.`**right**, the father `op` and any

[6]The lack of suitable qualifier means that it is impossible to have a production of the form `name -> name*` or `name -> name+`, where `name` designates both a list operator and a phylum.

non-local symbol may be referenced. If, in a **first** or **last** clause, the (right or left) brother is referenced, this implies that there should exist at least *two* sons; since this condition can only be checked at run-time, it is implemented by raising the predefined exception `ARITY-ERROR` when that non-existent brother is referenced.

If a **case position** construct specifies both a **first** and a **last** clause, and if a particular instance of that production has only one son, then the definition which is used to define the corresponding inherited attribute of that son is the first one in textual order, namely that of the **first** clause if the construct is **case position is first**: ...; **last**: ...; **other**: ...; **end case**, and that of the **last** clause in the other case. This means that the relative order of the **first** and **last** clauses is significant.

As for every other selection construct, the expressions in the right-hand side of the clauses must have a common type which will be that of the whole expression.

The above specification is valid when the **case position** construct is the main operator of the semantic rule. It is however possible to embed this construct more deeply into an expression; the semantics is then that of a factorization. More precisely, an expression of the form:

```
func (arg1, arg2, ...,
   case position is
      spec1 :  expr1;
      spec2 :  expr2;
   end case,
   ..., argn)
```

is equivalent to the expression

```
case position is
   spec1 :  func (arg1, arg2, ..., expr1, ..., argn);
   spec2 :  func (arg1, arg2, ..., expr2, ..., argn);
end case
```

and an expression of the form

```
func (arg1, ...,
   case position is
      spec1 :  expr1-1;
      spec2 :  expr1-2;
   end case,
   ...,
   case position is
      spec1:  expr2-1;
      spec2:  expr2-2;
   end case,
   ..., argn)
```

is equivalent to the expression

```
case position is
   spec1 :  func (arg1, ..., expr1-1, ..., expr2-1, ..., argn);
   spec2 :  func (arg1, ..., expr1-2, ..., expr2-2, ..., argn);
end case
```

When an expression has as components a **case position** construct specifying a **first** clause and no **last** clause, and another **case position** construct specifying a **last** clause and no **first** clause, the **other** clause in each of these constructs is split to provide the missing **last** and **first** clauses respectively, so that the above distributivity rules may apply; this then implies that there exist at least two sons.

Due to the significance of the order of the **first** and **last** clauses pointed out above, the OLGA compiler checks that this order is used uniformly in a same semantic rule when it applies the distributivity rules. If this is not the case, an error message is emitted and the order which is used is that of the first encountered construct in textual order.

Nested **case position** constructs have strictly no meaning.

The above distributivity rules have been exemplified using function calls, but they apply to every other expression constructor: operators, conditionals, selection operators, etc.

Right-hand side symbols of list productions have a syntactic attribute **$position**, of type *int*. Its value is the rank of the corresponding son in the list production: 1 for the leftmost son, 2 for the second leftmost son, etc., up to **$arity** (father) (see below) for the rightmost son.[7]

13.6.2 Synthesized Attributes of the Left-hand Side Symbol

Semantic rules defining the synthesized attributes of the left-hand side symbol of a list production look like "normal" ones, in which, apart from inherited attributes of the left-hand side symbol and non-local references, only the synthesized attributes of the leftmost and/or rightmost son are visible those definitions may depend on the actual arity (number of sons) of the production, and the **case arity** construct may be used to describe and check those cases.

More precisely, such a semantic rule may only refer to the inherited attributes of the left-hand side symbol `op`, the synthesized attributes of the leftmost and/or rightmost son(s) `PH.`**first** and `PH.`**last**, and non-local attributes. If it does not depend on the actual arity of the production, it is still necessary to (dynamically) check that the referenced sons are actually present, which is achieved using the predefined `ARITY-ERROR` exception. Note that this dynamic check is only necessary for perhaps-empty list productions, and not for never-empty ones. Also note that the leftmost and the rightmost sons may be the *same* node when the actual arity is one.

The left-hand side symbol of a list production has a predefined attribute named **$arity**, of type *int*, whose value is the number of its offsprings. It could thus seem that the **case arity** construct, to be used when the definition depends on the arity of the production, is subsumed by the standard **case ... is** selection expression based on the value of that attribute,

[7]This attribute can replace the older attributes **$is-first** (≡ **$position** = 1) and **$is-last** (≡ **$position** = **$arity** (father)).

however the former construct is preferable because it provides for more extensive consistency checks. Indeed, each fixed-arity clause specifies the definition to be used when the actual arity corresponds to the constant in the left-hand side of the clause, as a standard **case ... is** construct would do; but in the clause corresponding to arity 0, no son may be referenced, and this is checked by the OLGA compiler. In the clause corresponding to arity 1, the unique son must be referenced as `PH`. In clauses corresponding to other arities, and in the **other** clause which is mandatory, the leftmost and the rightmost sons `PH.`**first** and `PH.`**last** may be referenced (those sons may actually be the same one if the **other** clause is used for arity 1); integer qualifiers may also be used in clauses for fixed arities (i.e. not in the **other** clause); for instance

```
case arity is
   ...
   3 :...  $a(PH.1 {same as PH.first }) ...  $b(PH.2)
      ...  $c(PH.3 {same as PH.last }) ...
   ...
end case
```

Although it is syntactically correct to have fixed-arity clauses for arities greater than 1 —and these clauses will actually be taken into account—, the only special cases are for arities 0 and 1. Also note that the clause for arity 0 is *mandatory* for perhaps-empty productions and *useless* for never-empty ones.

As for every other selection construct, the expressions in the right-hand side of the clauses must have a common type which will be that of the whole expression. Distributivity rules similar to those of the **case position** construct apply. emantic rule

13.7 Other Expressions Allowed in Semantic Rules

13.7.1 The *map* Operator

The **map** operator can be used in list productions to compute some value by accessing synthesized attributes of all sons in a left-to-right or right-to-left traversal of the list of those sons.

Syntax (the `<map direction>` and `<map id>` non-terminals are defined in section 9.10):

```
<expression>        =  <map>;
<map>               =  map <map direction> <map id>
                          <opt-other>
                          value <expression>
                          <opt-other>
                       end map ;
<opt-other>         =  ;
<opt-other>         =  other <actual +>;
```

The **map** construct allows to traverse the list of sons either in the left-to-right (keyword **left**) or right-to-left (keyword **right**) direction and accumulate a value computed by the application of a specified function to that accumulated value and some other arguments. Assume there are i - 1 **actuals** after the first **other** keyword ($i = 1$ if the whole clause is omitted) and j actuals after the second **other** keyword ($j = 0$ if the clause is omitted). Then the specified map-function, which may be an operator function (see chapter 8), must have $i + j$ arguments such that the type of the i-th argument unifies with the type of the result; this common type must also unify with that of the expression following the **value** keyword, which is the initial value of the accumulator; that common type will be that of the whole **map** expression. The other expressions following the **other** keywords must match the types of the other parameters of the function, from the first to the (i - 1)-th and from the (j + 1)-th to the (i + j)-th, respectively. Those expressions may involve, in addition to inherited and given attributes of the left-hand side symbol and non-local references, synthesized and given attributes of the "current" son `PH`, its left brother `PH.`**first** and its right brother `PH.`**last**; in those latter cases, those references should be embedded in **case position** constructs[8] or an `ARITY-ERROR` exception might be raised.

The dynamic semantics of the **map** construct is as follows: first, the initial value of the accumulator (the expression following the word **value**) is computed; if the arity of the production is zero, then this value is the result of the whole construct. Otherwise, the list of sons is traversed—starting with the leftmost one if the direction is **left**, or the rightmost one if the direction is **right**—and the following process is repeated on each son: the expressions following the words **other** are evaluated in an unspecified order in the context of the current son; then the specified function is called with its i-th parameter bound to the value of the accumulator and the other parameters bound to those other values; the result of this call becomes the new value of the accumulator. When the last son has been processed, the value of the accumulator is the result of the whole **map** construct.

A **map** construct can be used in list productions anywhere any other kind of expression is allowed; it is simply a special manner to compute a value in terms of the attributes of the production.

Map constructs do not distribute. Two **map** constructs in two different branches of an expression are perfectly valid. It is even possible to have nested **map** constructs, although this could lead to tremendous computation times...

Map constructs may appear in semantic rules of list productions, of course, and also in production-wise value definitions (local attributes); however they are forbidden in production-wise local functions.

It is not possible to **map** the non-strict boolean operators **and** and **or**; this is anyway not necessary because, since all the synthesized attributes of right-hand side symbols which participate in a **map** construct are evaluated before the construct itself is evaluated, the same result can be achieved using the strict operators `&` and `|`.

Examples:

The first example shows how to compute the arity of a list production:

```
where op -> PH* use
   $computed-arity (op) :=
      map left + value 0 other 1
      end map ;
```

[8] This means that **case position** constructs distribute and "bubble" up only to the semantic rule level or to the enclosing **map** construct.

Note that if we had a function `plus1` with profile `(int): int` which adds one to its argument, we could also write:

```
where op -> PH* use
   $computed-arity (op) :=
      map left plus1 value 0
      end map ;
```

The second example shows how to compute on the father the list of values of an attribute of the sons; we shall assume that the type `list-of-string` has been declared as **list of** `string`, that the phylum to which the operator `op` belongs has a synthesized attribute `$ls` of type `list-of-string` and that the phylum `PH` has a synthesized attribute `$s` of type `string`:

```
where op -> PH* use
   $ls (op) :=
      map left +
         value list-of-string ()
         other list-of-string ($s (PH))
      end map ;
```

Note that type inference here needs to solve both an overloading problem and a polymorphism problem for function `+`. Other examples appear in appendix A.

13.7.2 More on local attributes

In a production considered as a level 1 block, local declared values whose definitions involve input attribute occurrences play the role of "local attributes", i.e. attributes attached to an operator (a production) rather than to a phylum. They allow to compute and store subexpressions occurring in more than one semantic rule, so that efficiency is increased. This also alleviates the constraint that an OLGA AG be in normal form.

Attribute occurrences which may occur in the definition of a local attribute are the following:

- given and inherited attributes of the LHS operator,
- non-local references to attributes of nodes upward (`[ <up symbol +> ]`) in the tree, and
- given and synthesized attributes of the RHS symbols if the production is a fixed-arity one, or
- if the production is a list one, given and synthesized attributes of the first and last son, embedded in a **case arity** construct if necessary, and **map** constructs.

Other local attributes may also be used, provided they are defined previously.

A local attribute is considered as a full rank attribute by the FNC-2 system regarding the moment when it is computed and the way it is stored. It also participates in attribute dependencies.

13.8 Generation of Missing Semantic Rules

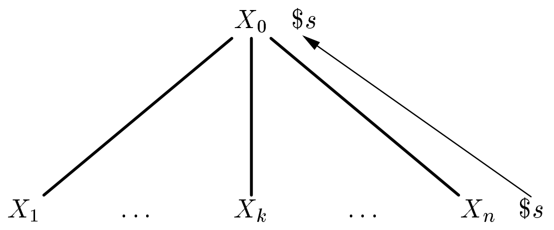
In a given production there should exist one and only one semantic rule defining each output attribute occurrence. However, experience shows that in most attribute grammars a large proportion of those semantic rules are just *copy rules* (also named *identities* or *transfer rules*), i.e. of the form

```
$a1 (sym1) := $a2 (sym2)
```

Also, a very large majority of those identities are such that `$a1` and `$a2` are identical, and it is very easy to devise mechanical rules to generate most of them. Doing so results in much more compact attribute grammars, and in substantial savings on the effort needed to develop them. The default rules presented hereafter have been introduced originally by Bernard Lorho in his systems DELTA [Lor77] and TABACT, and slightly improved in FNC/ERN, the ancestor of FNC-2 [Jou84]. Their important utility has been proved in a number of large applications.

In this section, the word "father" will designate the left-hand side symbol of a production, and the word "son" any of its right-hand side symbols. After processing all the semantic rules attached to a production,[9] the OLGA compiler will determine whether all the output attribute occurrences have been defined; if not so, it will try to generate some identities to replace those missing semantic rules, using the default rules presented hereafter.

If not explicitly defined, a (normal) synthesized attribute of the father will receive the value of the same attribute—the one with the same name—of the rightmost son if it exists otherwise this is an error.



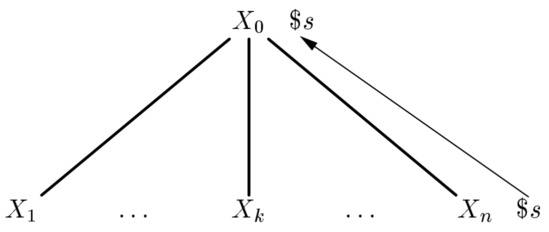
or **error**

Note that when the production is a perhaps-empty list production, the generated rule has the form:

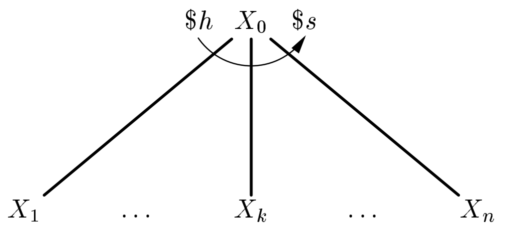
```
where op -> PH* use
   $s (op) :=  case arity is
                  0      :  raise ARITY-ERROR;
                  other  :  $s (PH.last);
               end case ;
```

The synthesized part `$s` of a compound attribute `$s` and `$h` of the father will receive the value of the same attribute of the rightmost son if it exists; otherwise it will receive the value of the inherited part of that compound attribute of the father.

[9] See chapter 15 for modularity considerations.



or



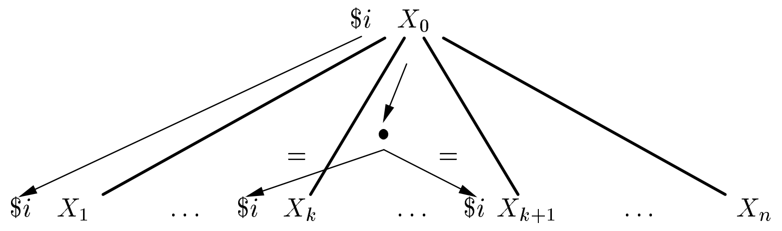
Note that when the production is a perhaps-empty list production, the generated rule has the form:

```
where op -> PH* use
   $s (op) :=  case arity is
                  0      :  $h (op);
                  other  :  $s (PH.last);
               end case ;
```

if `$s` and `$h` are attached to the phylum `PH`, otherwise it is:

```
   $s (op) := $h (op)
```

A (normal) inherited attribute of a son will be defined according to the position of that son in the right-hand side of the production. If that son is the leftmost one, then it will receive the value of the same attribute of the father, if it exists otherwise it is an error. If that son is not the leftmost one, then it will receive the value of the same attribute of its left brother, if it exists; otherwise it is an error. In the latter case, this is not achieved by creating a copy rule, because this would violate the normal form of the attribute grammar. Rather, an (anonymous) value is created, with the value which would have been assigned to that attribute of the left brother; then, two copy rules are created to assign that value to those attributes of the son being considered and its left brother.



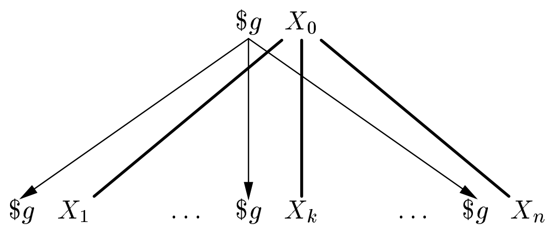
or **error**

• new anonymous value

The situation is quite different for list productions since the same semantic rule specifies the value of the attribute occurrence of each son. If such a d efinition is missing, and if that attribute is not attached to the father, then it is an error; otherwise the following definition is generated:

```
where op -> PH* use
   $i (PH) := $i (op);
```

A global attribute of a son, whatever its position, will receive the value of the same attribute of the father, if it exists; otherwise it is an error.

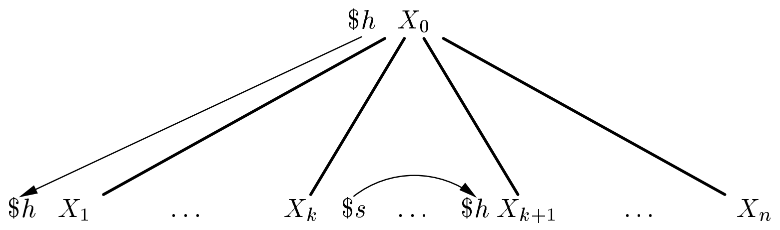


or **error**

For list productions, the following rule is generated, if possible:

```
where op -> PH* use
   $g (PH) := $g (op);
```

The inherited part `$h` of a compound attribute `$s` and`$h` of a son will be defined according to the position of that son in the right-hand side of the production. If that son is the leftmost one, then it will receive the value of the inherited part of the same attribute of the father, if it exists otherwise it is an error. If that son is not the leftmost one, then it will receive the value of the synthesized part of the same attribute of its left brother, if it exists; otherwise it is an error.



or **error**

For list productions, this amounts to creating the following definition, if possible:

```
where op -> PH* use
   $h (PH) :=  case position is
                  first   :  $h (op);
                  other   :  $s (PH.left);
               end case ;
```

The purpose of a compound attribute is thus to walk left-to-right and top-down around the tree and be updated e.g. at each node of a list production; it is for instance the case of a "symbol table" attribute in a declaration list. The use of a global attribute is, as its name means, to be globally available in a whole subtree; it is for instance the case of a "symbol table" attribute in the statement part of a program or procedure. Note however that in the latter case it is preferable to use non-local references. See appendix A for examples.

Chapter 14

Tree Construction Operations

14.1 Tree Construction

14.1.1 Generalities on tree types and values

OLGA provides a mechanism allowing to construct, as working attributes computed on some tree described by some abstract syntax, other trees described by other syntaxes. These new trees may also be decorated during their construction with some attribute values. They may then be fed to subsequent evaluation passes. All those decorated trees are described by AASes. This feature is very helpful for instance to write compilers in which the program is successively represented by several different intermediate languages each phase thus produces as an attribute value the tree which is input to the next phase. This allows to express, at least at the linguistic level, the concepts embodied in *attribute coupled grammars* (ACGs) [GG84, GGV86]; implementing ACGs is another story...

Syntactically specifying an AG to be functional (see chapter 15) states that *output* AASes describe trees which will be constructed as attributes of trees described by the *input* AAS. In most practical examples there will be only one output AAS. This chapter deals only with functional AGs, in which one or more new tree(s), possibly described by the same AAS as the input one, must be constructed and decorated. Side-effect AGs, which do not construct any new tree but merely add different attributes to the input tree, were described in previous chapter.

Specifying an output AAS declares its phyla as (abstract) tree types and its operators as tree construction functions. This means that the names of these phyla and operators must not conflict with the names of other types and functions. The declaration point of those names is the beginning of the outermost block of the attribute grammar.

More precisely, a phylum `PH` of an output syntax corresponds to a tree type named `PH`. A tree type is a completely opaque type, such that nothing is known about its implementation; the only possible operations available for a tree type are the declaration of (synthesized) attributes and values of this type, passing values of tree types as parameters, tree construction operations corresponding to the operators belonging to the corresponding phylum, destructuring through pattern matching (see section 9.6 and section 14.2 below), and testing two tree values for equality and identity (discussed in section 14.2 below). In particular, it is NOT allowed to use tree types as components of structured types, and it is NOT allowed to pass tree types and tree construction functions as module parameters[1].

(Parts of) output trees are constructed as working attributes on the input tree. The OLGA compiler checks that the root phylum of the input AAS has exactly one tree-typed synthesized working attribute for each output AAS, and only these attributes. The type of these attributes must be the root phyla of the output AASes. The result of the evaluation is then the set of those trees, while the input tree is destroyed.

Trees can be constructed and "assigned" to declared values in some productions, but these values must not be used to define any inherited attribute. The only other situation in which tree construction functions may appear outside semantic rules is for defining other, more complicated tree construction functions.

A same (input) phylum may have any number of tree-typed synthesized attributes belonging to the same output AAS.

The OLGA compiler checks that each constructed subtree is well-formed w.r.t. the corresponding output AAS. However it does not check that all of those constructed subtrees actually appear in the final tree(s).

IMPORTANT NOTE: the desire to make *descriptional composition* of ACGs [Gie86] possible with FNC-2 leads to impose several restrictions on the use of tree values:

- tree-valued attributes must be synthesized only;[2]
- user-defined functions involving tree types and tree construction functions must appear in the compilation unit (attribute grammar) in which they are used, so that their body is known to the compiler;[3] furthermore these functions must not be (directly or indirectly) recursive.

14.1.2 Tree construction functions

A fixed arity operator `op` belonging to phylum `PH` and declared as

```
op -> PH1 PH2 ...  PHn
```

corresponds to a tree construction function named `op` with profile

function `op (PH1; PH2; ...; PHn):  PH`

A list operator `op` belonging to phylum `PH` and declared as

```
op -> PH1* or op -> PH1+
```

corresponds to several tree construction functions defined as follows:

function `op ():  PH`
function `op (PH1):  PH`

[1]This means that, in OLGA, the syntactic and semantic domains, i.e. that of trees and that of tree attributes, are completely separated. We admit that this is a questionable choice.

[2]It has been shown in [Gie86] that composing two SNC ACGs is guaranteed to lead to an SNC AG only if tree-valued attributes are purely synthesized.

[3]so that they may be expanded in-line if necessary; it is not yet known, but very probable, that this sort of macro-expansion of tree construction functions is necessary for descriptional composition.

are the "initialization" functions which construct a list with zero and one offspring respectively. If `op` is a never-empty list operator then the first function is accessible only in the initialization part of a **map** construct associated to a never-empty list production (see section 13.7.1), so that it remains impossible to construct an empty list headed by `op`, except temporarily during that **map**.

function `op-post (PH; PH1):  PH` **raise** `TREE-ERROR`
function `op-pre (PH1; PH): PH` **raise** `TREE-ERROR`

allow to add an offspring (the `PH1` parameter) to an already existing list node headed by the operator `op` (the `PH` parameter), respectively at the end and at the beginning of that list. If the `PH` parameter is not a list node headed by `op`, the parameterless exception `TREE-ERROR` is raised.

function `op-merge (`**list of** `PH1;` **list of** `PH1):  PH`
raise `ARITY-ERROR`

allows to merge the offsprings of two list nodes into a single node headed by `op` whose offsprings are those of the first parameter followed by those of the second. What is denoted by **list of** `PH1` in the "profile" above is actually a very special, anonymous type—this type is a type expression in sense of chapter 7—, such that the only way to acquire a value of that type (a *forest* of trees) is to destructure a list node whose offsprings are of type `PH1` (see section 14.2 below).

function `op-all (`**list of** `PH1):  PH` **raise** `ARITY-ERROR`

creates a list node headed by `op` whose offsprings are those of another destructured list node, provided that the phylum of these sons is actually PH1. This allows to rename an existing list node.

In the two latter functions, if respectively both and the unique parameter are empty lists, and if `op` is a never-empty list operator, the `ARITY-ERROR` predefined exception is raised.

These new functions explain the test performed on each AAS (see chapter 12) to check that these implicit names do not conflict with explicit ones.

As an exception to the normal type checking rules, if a phylum $\mathtt{PH}_1$ is included in a phylum $\mathtt{PH}_2$ then for any tree construction function with a parameter of type $\mathtt{PH}_2$ the actual parameter may be of type $\mathtt{PH}_1$ (in addition to $\mathtt{PH}_2$ of course). This is only an application of overloading.

14.2 Tree Matching

14.2.1 Tree Patterns

A fixed-arity node is matched by a pattern quite similar to that matching a record (see section 9.6) with the same semantics.

The only patterns which could match a list node headed by operator `op` belonging to phylum `PH` and whose offsprings belong to phylum $\mathtt{PH}_1$ are as follows:

- `op()`
  matches only an empty list; does not apply if `op` is a never-empty list operator.

- `op(P)`
  where `P` is a subpattern which can match a tree belonging to phylum $\mathtt{PH}_1$, matches if the `op` node is a one-element list whose element matches `P`.

- $\mathtt{op\text{-}post(P_1,\ P_2)}$
  where $\mathtt{P}_1$ is a subpattern which can match a tree headed by operator `op` and $\mathtt{P}_2$ is a subpattern which can match a tree belonging to phylum $\mathtt{PH}_1$, matches if the `op` node is not empty and its last (rightmost) son matches $\mathtt{P}_2$ and the list of all sons but the last, when considered as an `op`-node, matches $\mathtt{P}_1$.

- $\mathtt{op\text{-}pre\ (P_2,\ P_1)}$
  same as `op-post` with $\mathtt{P}_2$ matching the first (leftmost) son and $\mathtt{P}_1$ the list of all sons but the first.

- `op-all (X)`
  where `X` **must** be an **untyped variable**—this is because no pattern can match a forest except the universal pattern—matches in any case; `X` will be bound to a forest containing all the offsprings of the matched node, whatever their number.

`op-merge (X, Y)` is not a possible pattern because there would be no way to draw the limit between the sublists `X` and `Y`.

Finer patterns may be constructed by combining these constructors; for instance, in the pattern `op-pre (X, op-pre (Y, Z))`, `X` would correspond to the first offspring, `Y` to the second and `Z` to a list node containing all the other offsprings, provided there are at least two of them (three if `op` is a never-empty list operator).

If a phylum $\mathtt{PH}_1$ is included in a phylum $\mathtt{PH}_2$, and if a variable appearing in some pattern is determined to be of type $\mathtt{PH}_2$, then (by exception to normal overloading rules) it will match any tree of type $\mathtt{PH}_1$ or $\mathtt{PH}_2$ and will be visible in the RHS of the clause as having type $\mathtt{PH}_2$; to have it match only trees of type $\mathtt{PH}_1$ and be visible with type $\mathtt{PH}_1$, use an explicit type constraint.

Unlike pattern variables in non-tree patterns, which are visible only in the "hold-all" name space described in chapter 11, the pattern variables in tree patterns are visible in this "hold-all" name-space—to participate in tree construction functions, which is their primary goal. Care must hence be taken to choose these pattern variables names so that they don't conflict with the names of other grammar symbols of interest in the production.

When some pattern variable appears more than once in some pattern, equality of the corresponding subtrees must be tested. Two subtrees are equal (in the sense of operator `=`) if and only if they have the same syntactic structure and are decorated by the same (in the sense of usual equality) attributes, not taking into account the **$scx** attributes. Two subtrees issued from different tree nodes are never identical.

Note that tree construction functions and tree pattern matching are defined in OLGA as in the tree construction actions associated to a concrete grammar: see Part V of this complete volume, *Generating Abstract Trees with SYNTAX*.

If some variable corresponding to some output tree node is used more than once in the final value of the right-hand side expression of a match clause, the corresponding tree is *duplicated* for each of these instances, so that the output tree actually remains a tree and does not become a graph. More generally, if any subtree is used more than once to construct a larger tree, it is duplicated. However, and although this is of no concern for OLGA, the attributes attached to the duplicated nodes (see next section) are not copied but shared as much as possible.

See section A.6 of appendix A of this manual for examples of tree construction and destructuring.

14.3 Tree Decoration

If some output AAS specifies that some phyla have attributes—this will be the usual case—then these attributes must be computed and attached to the output tree during its construction.

The construction of a node of an output tree and the assignment of values to its attached attributes must be performed together in a single **with** expression.

Syntax:

```
<expression>           =  <decoration expr>;
<decoration expr>      =  <expression>
                             with <decoration clauses>
                             end with ;
<decoration clauses>   =  <decoration clauses> ","
                          <decoration clause>;
<decoration clauses>   =  <decoration clause>;
<decoration clause>    =  <attrib name> ":=" <expression>;
```

The expression before the keyword **with** must express the construction of some new tree node as defined in section 14.1. The corresponding tree type is the type of the whole expression. There must exist in such an expression one and only one decoration clause for each of the attributes attached to the corresponding phylum, except maybe for the **$scx**; attribute which by default receives the value of the **$scx** attribute of the symbol in the left-hand side of the current input production (however a different value may be specified), and for the "syntactic" attributes **$arity** and **$position** which must not be given a value (their value is determined by the shape of the tree node to which they are attached). A decoration clause is similar in any respect to a semantic rule for a synthesized attribute, except that the left-hand side attribute occurrence must not specify any grammar symbol (the newly constructed tree node is assumed). In the RHS of the decoration clause, the other constructs dealing with attributes—attribute occurrences, **case arity** constructs, **map** constructs, etc.—refer to input trees and attributes, and can also refer to attributes of already constructed output tree nodes, provided that the latter are denoted by an identifier this can be achieved by using a **case ... is** construct, possibly reduced to a single universal pattern.

When an already constructed output tree node is destructured by pattern matching, it and its offsprings come with their attached attributes; however it is possible to change some or all of these attributes using a new decoration expression.

Every newly constructed tree node, be it in a semantic rule, in a value definition, or in a user-defined function, MUST be embedded in a decoration expression if the corresponding phylum has attributes. As said before, this decoration expression must supply a value for every such attribute.

See section A.6 of appendix A for examples of tree construction and decoration.

Chapter 15

Structure of Attribute Grammars

15.1 Attribute Grammars and Phases

Attribute grammars are the backbone of an OLGA specification. In addition to other kinds of declarations—an AG is a level 2 block—, they allow to declare (working) attributes and to define them using productions and semantic rules.

Syntax:

```
<compilation unit>     =  <attribute grammar>
<attribute grammar>    =  <AG header>
                             <AG decl *>
                             <semantics>
                          end grammar ;
<AG header>            =  attribute grammar <ID> <AG profile> is ;
<AG profile>           =  "(" <AG param> ")" ":" "(" <AG param attr *> ")" ;
<AG param>             =  <AG param> ";" <AG param element> ;
<AG param>             =  <AG param element> ;
<AG param element>     =  <attrib name> ":" <type id> ;
<AG param element>     =  <ID> ":" <type id> ;
<AG param element>     =  <ID> "in" <AAS name> ;
<AG param element>     =  <ID> "out" <AAS name> ;
<AG param attr *>      =  ;
<AG param attr *>      =  <AG param attr +> ;
<AG param attr +>      =  <AG param attr +> ";" <AG param attr element> ;
<AG param attr +>      =  <AG param attr element> ;
<AG param attr element>=  <attrib name> ":" <type id> ;
<AAS name>             =  <ID>;
<AG decl *>            =  ;
<AG decl *>            =  <AG decl +>;
<AG decl +>            =  <AG decl +>
                          <AG decl>;
<AG decl +>            =  <AG decl>;
<AG decl>              =  <decl>;
<AG decl>              =  <attribute part>;
<semantics>            =  <production groups>;
```

```
<semantics>           =  <attribute phases>;
<production groups>   =  <production groups>
                         <production group>;
<production groups>   =  ;
<attribute phases>    =  <attribute phases>
                         <attribute phase>;
<attribute phases>    =  <attribute phase>;
<attribute phase>     =  phase %ID is
                            <decl *>
                            <production groups>
                         end phase ";";
```

An AG definition specifies whether it is a side-effect AG (with the "on" keyword) or a functional one (without the "on" keyword), the AAS names of its input, a collection of declarations, including working attributes declarations, and a complete set of productions and semantic rules. An AG is the basic unit of attribute evaluation, i.e. the generation of missing attribute declarations and semantic rules (see chapter 13) applies to a whole AG and each AG defines a complete evaluator[1].

The header of an AG specifies its name. The AG definition define this function which must be declare in the declaration module. The AAS name attached at "in" keyword is the input AAS: the productions of the AG must correspond to those of this input AAS, and the working attributes will be attached to its phyla. In a side-effect AG, the AAS name attached at "on" keyword is the unique output one, and the OLGA compiler will check that both AASes are based on the same attributeless syntax. The first identifier of the input or ouptut AAS clause is the root phylum of its. The result part of AG header specifies the list of result attributes which are to be automatically declared as working attributes of the input root phylum.

Since an AAS is a special kind of module, the specification of an AAS name as an output AAS can be viewed as a special kind of import clause which, as said in chapter 14, defines tree types and tree construction functions. However, as opposed to module import clauses (see chapter 11), it is not possible to selectively import some names and not some others from an AAS: it must come as a whole. This can cause name conflicts for phyla if two differents phyla with the same name[2] exist in two different AASes which are output AASes of a same AG. In this case, and contrary to what happens for normal modules, this is **not** an error to do so; a mere warning will be emitted, listing the conflicting phyla names. This has however an important consequence: conflicting phyla names cannot be used as such and they **must** be qualified (using the # connector of chapter 11) by the name of the AAS, for instance to declare tree-valued attributes; in other words they "disappear" from both "import clauses". Such conflicts do not occur for operator names: they are viewed as (possibly overloaded) functions.

Also note that an AAS does *not* "export" the types of the attributes it declares. This is the only situation in which importation is not transitive. As a consequence, if any interesting usage of the input attributes, or the attributes attached to output trees, is planned—which

[1] However the FNC-2 system can be directed to generate a single evaluator for a sequence of several "piped" AGs.

[2] They can be completely different phyla which, by chance (?), happen to have the same name, or may stem from the importation of a subgrammar followed by augmentation, as mentioned in chapter 12.

is the case in every attribute grammar[3]—, their types *must* be imported separately from the module(s) in which they are declared, by using explicit import clauses (see chapter 11). Doing so allows to solve possible name conflicts, which would not be possible if importations of DCM were transitive through AASes.

An attribute grammar may contain no declaration at all if all the attributes declarations are to be automatically generated and if they all have predefined types; however this is not very likely to occur, and is bad programming practice anyway. Otherwise, constants, values, types, exceptions, and functions may be declared and/or imported from external modules, and working attributes may be declared as explained in chapter 13. An AG exports no object: its "profile" is given in terms of externally defined AASes.

The sequel of an AG is the specification of the productions and associated semantic rules. Each production of the base abstract syntax may be omitted or may appear once or several times, in any convenient order. The semantic rule defining each attribute occurrence may be specified or omitted; in the latter case it will be generated automatically if possible. If it is given however, it may appear only once, in a production group headed by the corresponding production (see chapter 13). Semantic rules in a given production group may be given in any convenient order. If a production is not listed, or is listed with no associated semantic rule, then all of those rules are assumed to be generated automatically.

If it is needed to further structure an attribute grammar, it may be divided into several *phases*. A phase has a name, which is completely ignored and appears only for documentation purposes, and consists of an optional list of declarations and in a list of production groups. A phase is a level 1 block where declarations of constants, values, types, exceptions and functions are allowed and where other such objects may be imported. These objects may then be used in the semantic rules of the phase. No local object is visible out of a phase. A phase is not an evaluation unit, which means that it may specify only a subset of the productions and semantic rules. The generation of missing attribute declarations—if an attribute is encountered for the first time in a phase, it will not be declared at the phase level but rather at the enclosing AG level—and semantic rules, and the associated "well-defined" check are performed at the AG level. Note that if an AG is structured into phases, no production group may appear out of a phase.

The initialization; of values declared in the AG proper and in all of its phases will be performed at the beginning of the evaluation, along with all parameterized module instantiations, if any.

15.2 Result of the Evaluation

Two kinds of AGs must be distinguished, as stated in section 13.1. In any case the input to an attribute evaluator is a decorated tree described by the input AAS.

If the AG is a side-effect one, which implies that it has one output AAS based on the same syntax as the input one, and that no explicit tree construction operation appears in the AG, the result of the evaluation is the input tree decorated with result attributes according to the output AAS. Those result attributes must be defined by semantic rules, as working attributes. Their direction (kind in the sense of chapter 13) can be automatically inferred from the semantic rule or specified explicitly using incomplete declarations. Given attributes

[3]The only exception is when an input attribute is passed unchanged in the output tree(s), but this would be quite unusual.

also attached to the output AAS cannot be changed during the evaluation and must not be defined by semantic rules. Other given and working attributes are destroyed.

If the AG is a functional one, the result of the evaluation is an anonymous record containing the values of the given and working attributes of the root of the tree[4]. The input tree itself is destroyed, together with its other given and working attributes. How this record is accessed and exploited is out of the scope of the OLGA language: see *the* FNC-2 *Reference Manual.* If the functional AG has one or more output AASes, explicit tree construction operations must appear in the AG the constructed trees are available as working attributes of the root of the input tree, and thus appear in the resulting record. Note that one of the output AASes may be the same as the input one.

It is not allowed to mix both forms of AGs, i.e. it is not possible to have one output AAS based on the same syntax as the input one whose result attributes would be defined by semantic rules, and other output AASes whose trees are constructed and decorated using tree construction functions. If it is desired to have, among others, an output AAS based on the same syntax as the input one, then its trees must also be constructed and decorated using tree construction functions.

It is possible to "pipe" two attribute evaluators defined by two AGs so that the input tree of the second one is (one of) the output tree(s) of the first one, provided of course that the AAS describing this common tree is the same for both AGs. This is even the most powerful way to structure an application based on AGs. However the way this "connection" is achieved is out of the scope of the OLGA language; see *the* FNC-2 *Reference Manual.* It is also possible to direct the FNC-2 system to generate a single evaluator for such a sequence of AGs, provided that the same input tree, possibly with different attributes, is used in the whole sequence; this allows to gain efficiency since intermediate result attributes become working and may be stored more efficiently than at tree nodes. It is also planned in the future to implement *descriptional composition* of ACGs, which would lift this restriction on the shape of the tree; it would also gain more efficiency since intermediate trees would not need be constructed at all.

[4]The names of the fields of this record are derived from the names of the corresponding attributes of the root.

Part III

Fnc-2 Reference Manual

Chapter 16

Outline of the Fnc-2 System

This chapter will be a general presentation of Fnc-2 . In the present release, it gives only pictures showing the organization of the system.

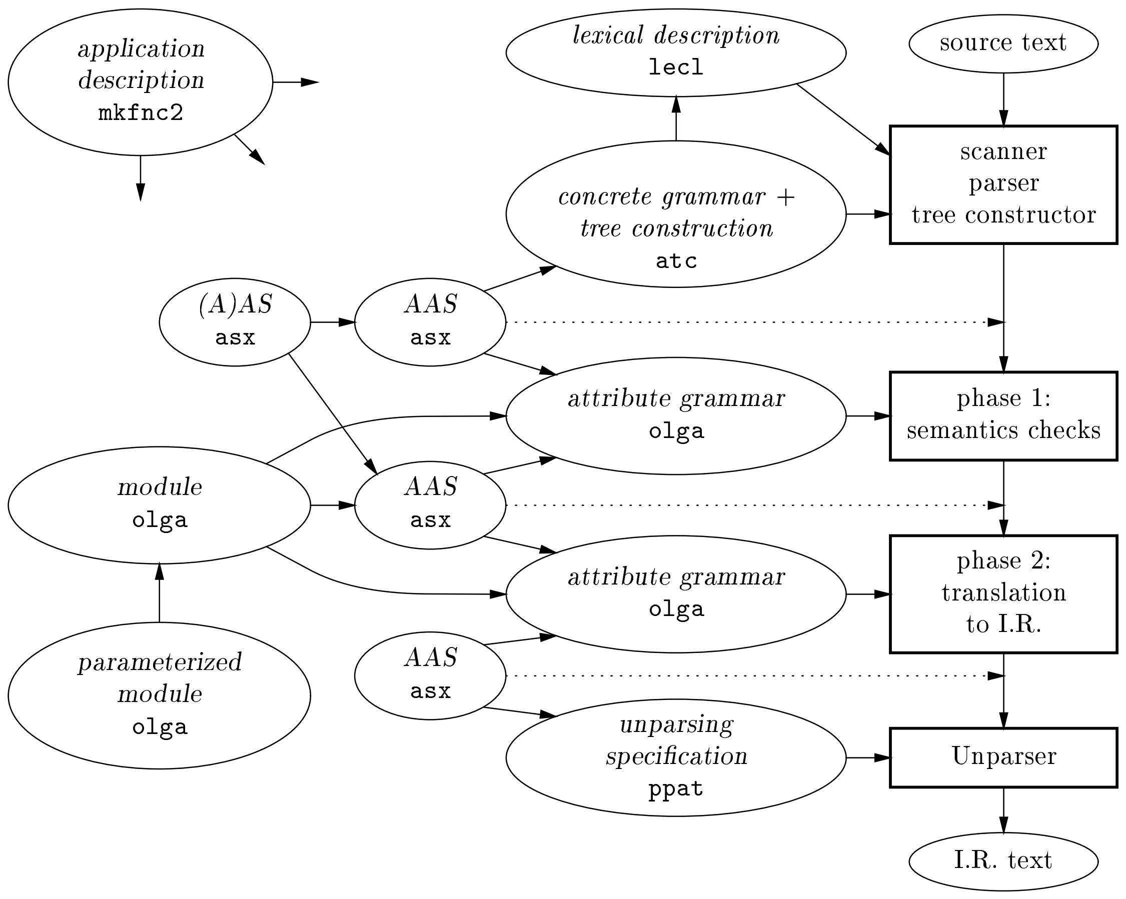


Figure 16.1: An FNC-2 application

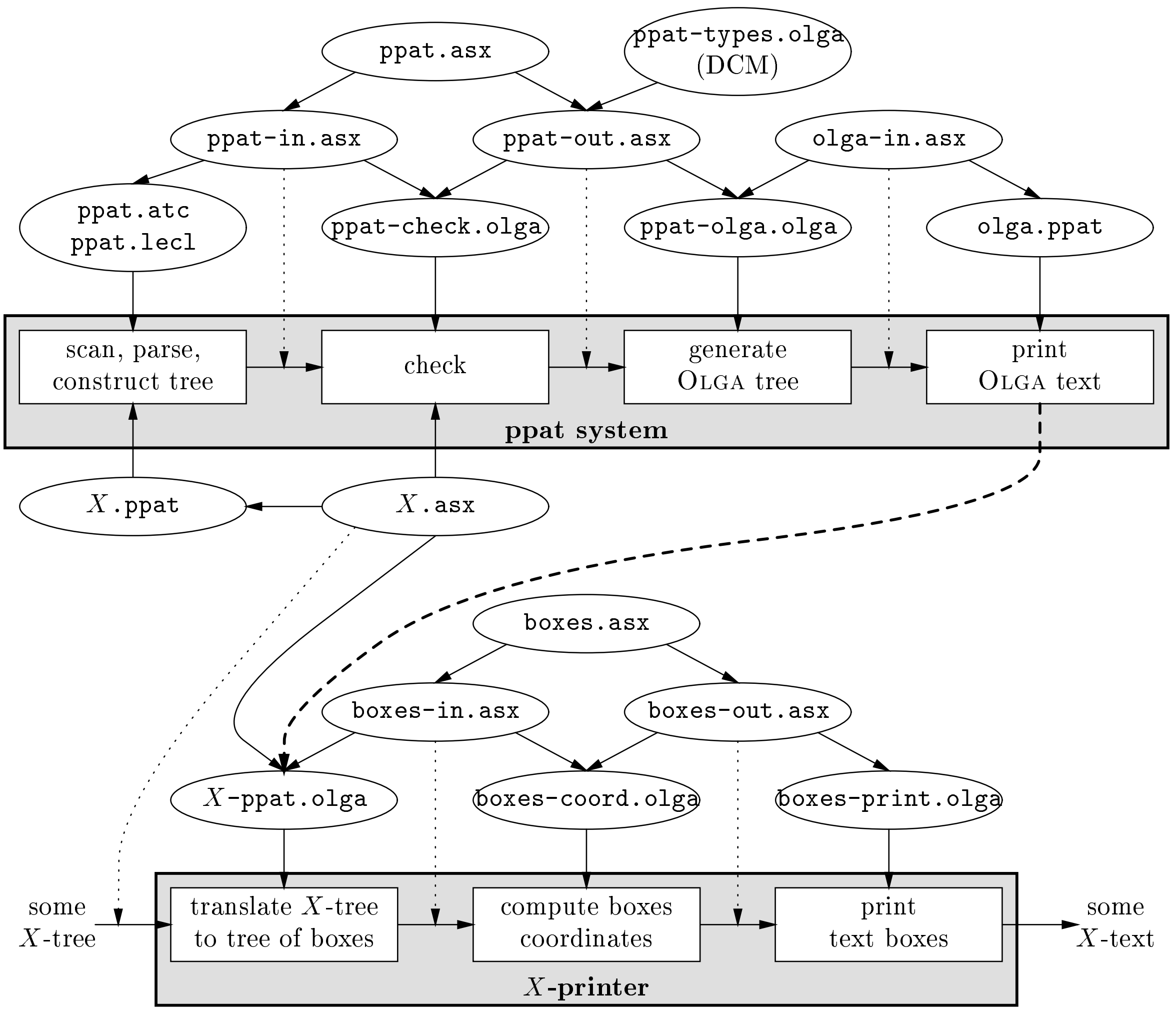


Figure 16.2: A PPAT appication

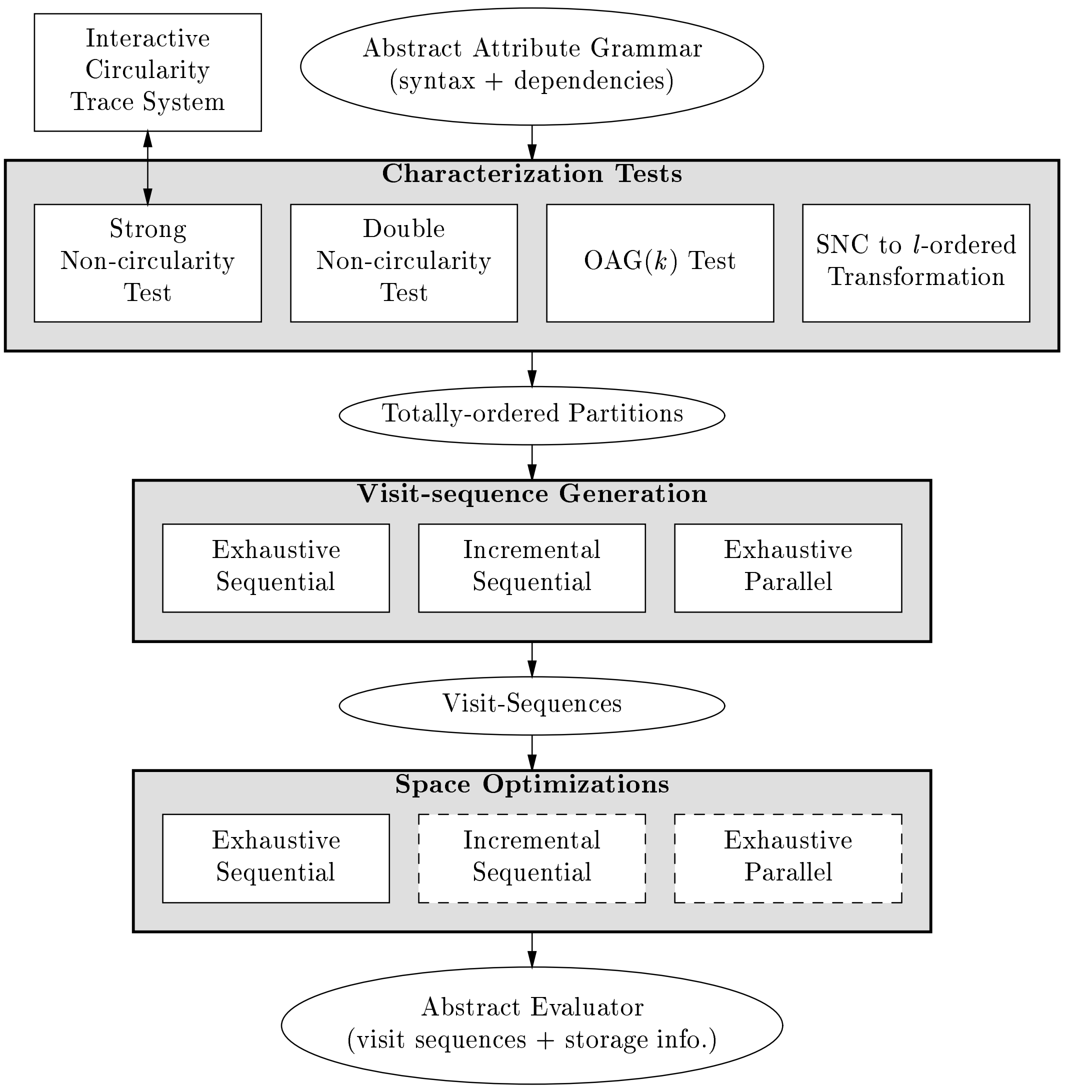


Figure 16.3: The evaluator generator

Chapter 17

The Attributed Abstract Syntaxes Compiler ASX

This chapter presents how to operate ASX, the component of the FNC-2 system whose role is to read, check and compile into an internal form descriptions of attributed abstract syntaxes (AASes) as described in chapter 12 of *the* OLGA *Reference Manual*. The generated internal forms will then be used by ASX and OLGA to process other descriptions.

17.1 Operation

When you type the following command:

```
tutorial% asx simproc-base.asx
```

ASX analyses the specified source file and, with default options and provided that no errors are detected, echoes the following on the terminal:

```
Release 1.8 of to tomorrow
simproc-base.asx:
   Scanner & Parser & Abstract Tree
   Symbol Table:
   Property Test
   Writing Symbol Tables
   Listing Output
```

The first line identifies the version you are using. Then, for each source file—ASX can compile several files in a single run—, its name is echoed, and the work is divided into several phases:

- *Scanner & Parser & Abstract Tree*
  performs lexical and syntactic analysis of the source text lexical and syntactic errors are detected and give rise to messages, but analysis proceeds.
- *Symbol Table*
  checks that every phylum and every operator used in the AAS is actually defined.

Double declarations are also detected in this phase. It also processes import clauses: the **import grammar** `my-grammar;` clause causes the tables generated by a previous run of Asx on AAS named `my-grammar` to be read in; type names import clauses cause Asx to read the corresponding module tables. All these tables must pre-exist, otherwise a fatal error is signaled and the processing of the current text is immediately terminated. NOTE: since Olga does not run yet, processing of type names is presently purely lexical. Thus, the **from** `my-module` **import all** ; form of import clause is forbidden, and other imported type names (either qualified or not) are assumed to exist without being checked. Only predefined type names are actually known by Asx. It is an error if any phylum used in an operator definition is not defined. However it is only a warning when an operator cited in the definition of a phylum is not defined: the offending name is then simply deleted from the phylum definition. The same occurs when a phylum cited in an attribute definition is not defined. If any error, in the above sense, is detected in this phase or in the previous one, processing of the current text is terminated.

- *Property Test*
  checks that the syntactic part of the AAS is reduced in the sense of chapter 12 of *the* Olga *Reference Manual.* If not so, processing is terminated (see also section 17.2.3 below).

- *Tables Output*
  generates the internal form of the AAS as an unreadable set of tables, gathered in file `my-grammar.ast` if `my-grammar` is the name of the AAS defined in the header. It needs not be the same as the name of the input file, however it is recommended to do so. The tables file is produced in the working directory.

- *Listing Output*
  produces a listing of the compilation in file `my-grammar.asx.l` if `my-grammar` is the name of the AAS defined in the header, whatever the name of the input file is. This file is produced in the working directory. Its contents is detailed below.

17.2 Contents of the Listing

17.2.1 Header

The header of the listing specifies:

- the base-name of the source file;
- the name (Asx), purpose and version number of the processor;
- the processing options used in this run, stripped from their leading "–";
- the date and time of the compilation.

17.2.2 Source text and error messages

Then comes the source text. Each line is numbered in the left margin. Error messages are embedded in this listing, together with a marker indicating the exact place where the error occurred. The messages themselves should be self-explanatory. This listing is followed by a summary of the number of warnings and errors reported. This listing is followed by a summary of the number of reported warnings and errors.

17.2.3 Reduceness Information

If the abstract syntax proves to be not reduced, this is indicated by two kinds of messages:

- `Phylum XXX is unreachable from the root phylum`
  states that it is not possible to derive phylum `XXX` from the root phylum, i.e. it is not possible to find a correct tree with root the root phylum containing an instance of `XXX`.
- `Phylum YYY cannot derive a terminal tree`
  states that it is not possible to find a correct tree with root (an operator of) `YYY` whose leaves are all labeled with nullary operators different from the implicit `null-XXX` ones.

The latter error message is turned into a mere warning if YYY is an empty phylum. This is interpreted as the fact that the grammar is incomplete and will be completed by importing it into other AASes which will define YYY. Note that such an incomplete grammar cannot be used directly as an input or output AAS of any AG: the OLGA compiler will detect this and reject the specification.

17.2.4 Description of the whole AAS

Then comes a cross-reference listing of the whole grammar: phyla, operators, attributes.

Phyla are listed in alphabetic order. Each comes with its internal code (of no practical use for the casual user), the number of the line where its definition appears, if relevant,[1] its name, the list of operators it contains, and the list of operators for which the phylum is used in the profile. For the root phylum, this list is replaced by a statement that this is the root phylum.

Operators are also listed in alphabetic order. Each comes with its internal code, the number of the line where its definition appears, if relevant, its name, and the name of the phylum its belongs to.

Attributes are also listed in alphabetic order. Each comes with the number of the line where its definition appears, if relevant, its name, the names of the phyla it is attached to, and its type (type name and name of the module from which it is imported, if relevant).

17.3 Command Line and Options

The basic syntax to invoke ASX is as follows:

`asx [` *options...* `] [` *input_files...* `]`

[1] i.e. if the phylum is actually defined or extended in the current file, and not merely imported from another AAS.

If no *input_files* argument is present, the standard input is read. Otherwise, each *input_file* is analyzed.

Options may appear in any order as long as they appear before the *input_files*. Only the last occurrence of a given option or its contrary is taken into account. Available options are as follows:

- **–v**, **–verbose**
  Animate the user's screen by displaying information about what happens. (Default)
- **–nv**, **–noverbose**
  Execute silently.
- **–sc**, **–source**
  Produce a short listing, including the items described in sections 17.2.1 to 17.2.3 above. (Default)
- **–nsc**, **–nosource**
  Do not produce any listing; implies **–nls**.
- **–ls**, **–list**
  Produce a long listing, including the items described in sections 17.2.1 to 17.2.4 above; implies **–sc**. (Default)
- **–nls**, **–nolist**
  Do not produce in the listing, if any, the items described in section 17.2.4 above.
- **–null**
  Enable implicit nullary operator generation (default; see section 17.2.3).
- **–nnull**
  Suppress implicit nullary operator generation (see 17.2.3).

Default options are such that the command

```
asx
```

is equivalent to

```
asx -v -sc -ls
```

17.4 Utility Programs

17.4.1 asx_to_olga

The command

```
tutorial% asx_to_olga abstract-grammar.asx
```

converts the attributeless abstract syntax `abstract-grammar`, which must exist and be readable, and produces a skeleton OLGA AG on the standard output. The generated skeleton contains a production group for each operator of the grammar, with one (empty) semantic rule for each result attribute. Constructs for dealing with list productions are inserted where possible, but it must be taken into account that, at this stage, result attributes have no kind yet.

17.4.2 asx_to_ppat

The command

```
tutorial% asx_to_ppat abstract-grammar.asx
```

converts the attributeless abstract syntax `abstract-grammar`, which must exist and be readable, and produces a skeleton PPAT grammar on the standard output. The generated skeleton contains a production group for each operator of the grammar. Constructs for dealing with list productions are inserted where possible.

17.4.3 atcmkfnc2

The command

```
tutorial% atcmkfnc2 concrete-grammar.atc
```

creates from the concrete syntax `concrete-grammar`, which must exist and be readable, a skeleton for automatic *makefile* generation in the `concrete-grammar.atc.f2aux` file. Usually, it is no concern of the user because *makefile* generation is done in a transparent manner by XFNC2. The generated skeleton contains ATC (see chapter 21) and ASX processor calls of the relevant (concrete and abstract) grammars. By putting such skeletons together, the resulting specification is used by a a makefile generator tool of FNC-2 to create corresponding makefiles.

Chapter 18

The Olga Compiler and evalgen

This chapter presents how to operate Fnc-2 , the component of the Fnc-2 system whose role is to read, check and compile into an internal form descriptions of Olga compilation units (declaration modules, definition modules and attribute grammars) as described in *the* Olga *Reference Manual*. Fnc-2 also translates these compilation units into one of the target implementation languages.[1]

IMPORTANT NOTE: this manual is only tentative, so it is NOT guaranteed that the executable program you have at hand behaves exactly as described below. Furthermore, the current release does not fully implement all the features of Olga described in Part II of this volume. Most of the discrepancies between the manual and the implementation can be inferred from the example files in the distribution.

18.1 Olga and evalgen makes Fnc-2

Fnc-2 is divided, for practical purposes, into two parts, Olga and evalgen. They can be run either integrally or separately. If you call the program fnc2, it performs both functions. But, if it is called with the option **-agwrite**, there is no evalgen pass (that is, no attribute evaluation happens) rather an intermediate file is created (called `<grammar-name>.agst`) instead. This file contains the attribute grammar in internal form proper for further evalgen processing. The Olga part performs classical compiler front-end functionalities; evalgen creates an executable evaluator for the given grammar. Moreover, if evalgen is called with the **-circ** or **-visu** options, visualization of the grammar becomes possible (see chapter 3).

The use of Fnc-2 will be illustrated by the generation of the *simproc* compiler whose specification is presented in Appendix A. This example is integrated, that is, `fnc2` performs both Olga and evalgen funtionalities.

18.2 Operation

When you type the following command:

```
tutorial% fnc2 simproc-check.olga
```

[1] Currently, C is the only possible target language.

FNC-2 analyses the specified source file (an attribute grammar here, see below for modules) and, with default options and provided that no errors are detected, echoes the following on the terminal:

```
Release 1.8 of tomorrow
simproc-check.olga:
   Scanner & Parser & Abstract Tree:  0.18
      Olga :
         Type checking :****rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
         Attribute Dependencies Construction :  rrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrr
            Default:.....................................................
..............
         cpu :  0.68 AG + 0.48 RWD = 1.17
         Test for terminal recursif functions :  0.02
            Evaluator generator :
               Class
               Dag (Bottom up)
               Strongly non circular test ci++ci++ci+++++ci+++++++i+++++ci++++++
++i+++++ci+++++++++++i+++++i+++++ci++
               Dag (Top down)
               Doubly non circular test ci++++ci+++++++++++i+++++++++++ci+++++++
+++++++++ci++++++++++++++ci++++++++++ci++++ci++++
               OAG(0) test +++++++++++++++++++++++++++++++++++++
               Visit Sequences (abstract) sssssssssssssssssssssssssssssssssssss
               Memory Optimisation :
                  Temporary Attributes ttttttttttt
                  Variable Attributes vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                  Stack Attributes ppppppppppppppppppppppppppppppppppppppppppppp
ppppppppppppppppppppppppppppp
                  Regrouping Variables and Stack
                     Construction of incompatibility graph :....................
.................
                     Regrouping :gcccccccccccccccgcccgccccgc
                  Elimination of copy rules -----------------||
            Evaluator generator :  0.38
      Translator of Visit Sequences :sssssssssssssssssssssssssssssssssssssssssss
sss :  0.12
      Translator of semantic rules (Bootstrapped) :rrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrr :  0.37
      Listing Output (Olga)
         Listing Output (Evaluator generator)
      Close (Olga)
         Close (Evaluator generator)
End of simproc-check :  2.33
```

The first line identifies the version you are using. Then, for each source file—FNC-2 can compile several files in a single run—, its name is echoed, and the work is divided into several phases:

- *Scanner & Parser & Abstract Tree*
  performs lexical and syntactic analysis of the source text lexical and syntactic errors are detected and give rise to messages, but analysis proceeds.

- *Olga*
  The OLGA compiler now takes control.

- *Type checking*
  Performs type checking on the whole attribute grammar. Each `r` corresponds to the Type checking of a single production.

- *Attribute Dependencies Construction*
  Processes attributes declarations and semantic rules and construct an internal form to be input to the evaluator generator.

- *Default*
  If specified by the **-dd** command line option (see below), missing attribute declarations are automatically generated (if possible). Missing semantic rules are also generated in this phase

If any error occurs in these phases, the "generation" phases are aborted, and the remaining "analysis" phases do only as much as reasonable. The next phases are as follows:

- *Evaluator generator:*
  The evaluator generator now takes control.

- *Class*
  Construct the syntactic equivalence classes which will be used to traverse the grammar in the following phases (see Part III of this volume, FNC-2 *Reference Manual*).

- *Dag (Bottom up)*
  Constructs, from the above equivalence classes, a bottom-up processing order of the grammar.

- *Strongly non circular test*
  Tests whether the attribute grammar is strongly non-circular and constructs partial evaluation orders. The characters after the title display the progress of the algorithm: `c` means a new class is being processed, `i` means a new iteration is starting and `+` means the processing of a single production.

- *Dag (Top down)*
  Constructs, from the above equivalence classes, a top-down processing order of the grammar.

- *Doubly non circular test*
  Tests whether the attribute grammar is doubly non-circular and constructs more refined

partial evaluation orders. The characters after the title display the progress of the algorithm: `c` means a new class is being processed, `i` means a new iteration is starting and `+` means the processing of a single production.

- *OAG(0) test*
  Tests whether the attribute grammar is OAG(0) and, if so, constructs total evaluation orders. Each `+` means the processing of a single production.

If any of the above phases fails (which is not the case here), comes then:

- *Transformation*
  Transforms the SNC or DNC AG into an equivalent l–ordered one and constructs total evaluation orders. The characters after the title display the progress of the algorithm: `c` means a new class is being processed, `i` means a new iteration is starting, `o` means a new order is being computed and `+` means the processing of a single production.

- *Visit Sequences (abstract)*
  Produces the (abstract form of the) visit sequences.

- *Memory Optimisation*
  The storage optimization phase now takes control.

- *Temporary Attributes*
  Temporary attributes are being identified. Each `t` corresponds to one such attribute.

- *Variable Attributes*
  Stores attribute occurrences into global variables (or at least attempts to). Each `v` corresponds to one such occurrence.

- *Stack Attributes*
  Stores remaining temporary attribute occurrences into global stacks. Each `p` corresponds to one such occurrence.

- *Regrouping Variables and Stack—Construction of incompatibility graph and Regrouping—*

  Groups several global variables together and several global stacks together (or at least attempts to). An incompatibility graph is constructed therefore: each `g` marks the beginning of a new pass and each `c` corresponds to the coloring of an attribute. This is a simple case in which a single pass is sufficient.

- *Elimination of copy rules*
  After storage means have been regrouped, some copy rules may be eliminated. Each `-` corresponds to the elimination of a copy rule between two attributes in the same global variable, and each `|` corresponds to the elimination of a copy rule between two attributes in the same global stack.

- *Translator*
  The translator to C now takes control and proceeds in two phases: first the semantic rules are translated, each `r` marking one rule, then the visit-sequences are translated.

- *Listing Output*
  produces a listing of the compilation in file `my-grammar.olga.l` if `my-grammar` is the name of the AG as specified in the header, whatever the name of the input file is. This file is produced in the working directory. Its contents is detailed below. This is done in two phases, one by the OLGA compiler and one by the evaluator generator.

- *Close*
  Cleanup.

For a (declaration or definition) module, the processing phases are the same except that, of course, the evaluator generator is bypassed. The translator also emits a slightly different message which, in particular, states whether a declaration module or a definition module is being compiled.

18.3 Contents of the Listing

18.3.1 Header

The header of the listing specifies:

- the base-name of the source file;
- the name (FNC-2), purpose and version number of the processor;
- the processing options used in this run, stripped from their leading "–";
- the date and time of the compilation.

18.3.2 Source text and error messages

Then comes the source text. Each line is numbered in the left margin. Error messages are embedded in this listing, together with a marker indicating the exact place where the error occurred. The messages themselves should be self-explanatory. Automatically generated semantic rules also appear embedded in the listing. This listing is followed by a summary of the number of reported warnings and errors.

18.3.3 Attribute declarations

After the source text comes the complete list of attribute declarations, in OLGA syntax, as inferred by the OLGA compiler. This list is identical to the one in the source text if automatic declarations have not been enabled, or, otherwise, contains the full declarations: name, kind, type and the list of phyla for each attribute. This piece of text can be cut off from the listing and pasted back into the source AG.

18.3.4 Equivalence classes

Then comes the listing of the syntactic equivalence classes.

18.3.5 Results of the SNC test

The results of the SNC test are then displayed. For each synthesized attribute occurrence is given the list of inherited attributes on which it may depend.

18.3.6 Results of the DNC test

The results of the DNC test are then displayed. For each inherited attribute occurrence is given the list of synthesized attributes on which it may depend.

18.3.7 Results of the OAG test or of the l–ordered transformation

If the AG is OAG, each phylum is given with the unique total order for evaluating its attributes. If the AG is not OAG, it has been transformed into an l–ordered one (see also [Par88]), and the same information appears but with possibly several total evaluation orders per phylum.

18.3.8 Storage means

If the storage optimization phase has been run, each evaluation order is given again with an identification of the storage means (global variable, global stack or tree node) assigned to each attribute occurrence.

18.3.9 Statistics

Lastly, a brief summary of the most important figures describing the AG is given.

18.4 Command Line and Options

The basic syntax to invoke Fnc-2 is as follows:

```
fnc2 [ options... ] [ input_files... ]
```

If no `input_files` argument is present, the standard input is read. Otherwise, each `input_file` is analyzed.

If the **-agwrite** option is given, the EVALGEN part is not executed and the grammar is written into an internal form.

EVALGEN can be called separately; it reads a `grammar_name.agst` file created by Olga.

```
evalgen [ options... ] [ grammar_name... ]
```

Options may appear in any order as long as they appear before the `input_files`. Only the last occurrence of a given option or its contrary is taken into account. Most options are common for Olga and EVALGEN, except for –**agwrite**, –**lisp** and –**visu**, –**circ**, respectively. Available options are as follows:

- –**v**, –**verbose**
  Animate the user's screen by displaying information about what happens. (Default)

- –**nv**, –**noverbose**
  Execute silently.

- –**sc**, –**source**
  Produce a short listing, including the items described in sections 18.3.1, 18.3.2 and 18.3.9 above. (Default)

- –**nsc**, –**nosource**
  Do not produce any listing; implies –**nls**.

- –**ls**, –**list**
  Produce a long listing, including the items described in sections 18.3.1 to 18.3.9 above; implies –**sc**. (Default)

- –**nls**, –**nolist**
  Do not produce in the listing, if any, the items described in sections 18.3.2 to 18.3.8 above.

- **-type**
  Perform type checking of the OLGA text. (Default)

- **-ntype** Do not perform type checking of the OLGA text.

The following options apply only if an attribute grammar is being compiled.

- **-dd**
  Enable automatic attribute declarations generation.

- **-ndd**
  Prevent automatic attribute declarations generation. (Default)

- **-dnc**
  Perform the DNC test after the SNC test, which is always run. (Default)

- **-ndnc**
  Do not perform the DNC test after the SNC test. The construction is aborted after the SNC test, unless an explicit **-visit** option directs FNC-2 to construct an unoptimized evaluator directly from the partial orders constructed during the SNC test.

- **-oag**
  If the AG passes the DNC test, perform the OAG(0) test. Implies **-dnc**. (Default)

- **-oagk** `k`
  If the AG passes the DNC test, perform the OAG(k) test, where `k` is a positive integer. Implies **-dnc**.

- **-noag**
  Do not perform any OAG test after the DNC test. The construction is aborted after the DNC test, unless an explicit **-ordre** option subsumes it (see below) or unless an explicit **-visit** option directs FNC-2 to construct an unoptimized evaluator directly from the partial orders constructed during the SNC and DNC tests.

- **-ordre**
  If the AG fails the DNC test or the OAG test, transform it into an l–ordered AG. Implies **-dnc**. (Default)

- **-nordre**
  If the AG fails the DNC test or the OAG test, do not transform it into an l–ordered AG, but rather abort the construction.

- **-optim**
  If total evaluation orders have been constructed by the previous phases, optimize storage usage. (Default)

- **-noptim**
  Do not optimize storage usage in any case.

- **-visit**
  Force the generation of an evaluator from whatever orders are available.

- **-nvisit**
  Prevent the generation of an evaluator in any case, but rather abort the construction.

- **-incr**
  An incremental evaluator is built.

- **-agwrite**
  (Can not be used with EVALGEN.) The attribute grammar is written into an intermediate file (called `<grammar-name>.agst`). This enables a separate EVALGEN run (and possible visualization by XVISU, see **-circ** and **-visu**).

- **-lisp**
  (Can not be used with EVALGEN.) A LISP evaluator is generated (instead of a C one).

- **-visu**
  (For EVALGEN; can not be used with OLGA.) Allows visualizing dependency graphs of a whole grammar (by calling XVISU, see chapter 3).

- **-circ**
  (For EVALGEN; can not be used with OLGA.) Allows visualizing phyla and attributes of a non–SNC grammar (by calling XVISU, see chapter 3). If the grammar is SNC and the **-circ** option is specified, *evalgen* will execute normally but the use of *xvisu* is meaningless and the dependency graph will not be displayed.

Default options are such that the command

```
fnc2
```

is equivalent to

```
fnc2 --v --sc --ls -type -ndd -dnc -oag -ordre -optim
```

i.e. it tries to construct the most optimized evaluator.

18.5 Generated Files

The compilation of an OLGA compilation unit uses and produces several files.

- The compilation of the declaration module (DCM) `my-module` whose source is in file `anything.olga` produces the following files: `my-module.olga.l` is the compilation listing, `my-module_b.h` is a C header file containing the declaration of the (C implementation of the) non-opaque objects in `my-module`, `my-module_b.c` is a C source file containing the definition of the (C implementation of the) non-opaque objects in `my-module`, and `my-module.dcst` is a data file containing a machine-readable version of the OLGA declarations in `my-module`.
- The compilation of the definition module (DFM) `my-module` whose source is in file `anything.olga` produces the following files: `my-module_df.olga.l` is the compilation listing, `my-module__df.h` is a C header file containing the declaration of the (C implementation of the) opaque objects in `my-module` (including functions), `my-module__df.c` is a C source file containing the definition of the (C implementation of the) opaque objects in `my-module`, and `my-module.dfst` is a data file containing a machine-readable version of the OLGA declarations in `my-module`.
- The compilation of the attribute grammar `my-AG` whose source is in file `anything.olga` produces the following files: `my-AG.olga.l` is the compilation listing, `my-AG_auto.c` is a C source file containing the code of the evaluator (visit sequences only), `my-AG_regles.h` is a C header file containing the code of the semantic rules[2], `my-AG_ga.h` is a (small) C header file containing the declaration of the C function implementing the evaluator, and `my-AG.dcst` is a data file containing a machine-readable version of the OLGA declarations at the outermost level in `my-AG`. If the **–agwrite** option is in effect, a `my-AG.agst` file is created containing the grammar in internal form (for EVALGEN).

The following files are used:

- `x.ast` for each reference to AAS `x`;
- `y.dcst` and `y.dfst`[3] for each reference to module `y`;
- `Olga-predef.dcst` for the predefined OLGA objects.

18.6 How to Obtain an Executable Program

In addition to the `.asx` and `.olga` files, whose compilation produces C header files and C source files as described above, you have to write by hand (short) C source files before

[2] One may wonder why it is a header (`.h`) file rather than a source (`.c`) file, since its only use of the functions it contains is to be called by the evaluator. The reason is that, in its current state, FNC-2 is unable to prevent name conflicts in the generated C code. Physically including `my-AG_regles.h` in `my-AG_auto.c` solves these conflicts. It is planned to add to FNC-2 a postprocessor which will help solving the conflicts (either by producing "renaming files" or by editing the files themselves), in which case `my-AG_regles.h` will become `my-AG_regles.c` and will be separately compilable.

[3] This is a violation of the principle of separing the declaration and definition modules. This will disappear in future releases.

obtaining a complete application. Their purpose is to make everything fit together and call the generated evaluators in the right order.

The root file must contain the usual `main` function which will perform the following tasks:

- decode command-line arguments;
- invoke the initialization part of the various modules, in the right order;[4](see section 11.3.4)
- initialize and invoke the first phase which constructs the initial tree (valuable information can also be found in chapter 23 of Part V of this volume, *Generating Abstract Trees with the SYNTAX System*);
- call the various evaluators in the right order, passing them the previous output tree;
- process the final tree(s).

There is no unique way to construct this file: each application is a special case. Valuable information can be gained by browsing through the example source files coming with the FNC-2 distribution. Of particular interest are `simproc_main.c`, `simproc_smp.c` and the `makefile`'s.

In the chapter 2 the use of the XFNC2 interface is described: it is the recommended way of creating, developing and maintaining FNC-2 applications.

[4]Presently this order must be guessed by hand. In future releases the postprocessor described above will generate these calls automatically.

Chapter 19

FNC-2 Pretty-Printers

19.1 Generalities

Pretty-printers are tools which take as input a text in some (computer) language—it may be a program or a specification such as those to be input to FNC-2—and output the *same* text—provided of course that it is syntactically correct—but with a possibly different layout: lines may be broken at different places, the indentation[1] of the lines may be different, etc. Pretty-printers are useful tools because of the following reasons:

- the resulting layout is chosen so that the text is easier to read in particular, the indentation usually reflects the nesting level of each construct;
- in a project involving several persons, they allow to enforce a uniform style;
- since they perform a complete lexical and syntactic analysis of the source text, they may be used to cheaply detect a number of errors before submitting the text to a sometimes slow compiler.

FNC-2 comes with five such pretty-printers, one for OLGA one for ASX, one for ATC(and **bnf**, one for MKFNC2 and one for PPAT. They have been generated by the system Paradis [DB85] which comes with SYNTAX.

19.2 Running a Pretty-printer

To run one of the FNC-2 pretty-printers, e.g. the OLGA one, on some OLGA text to be found in file `my-text.olga`, you just type:

```
tutorial% ppolga my-text.olga
```

The computer echoes the name of the input file, and then parses the text. If lexical and/or syntactic errors are detected, they are flagged on the terminal, as with the FNC-2 compilers, by displaying a message specifying the name of the input file, the line and column numbers, and the kind of error. Then the text is pretty-printed, which is made visible on the terminal by animating the line:

[1] i.e. the amount of white space before the first non-blank character in the line.

Pretty-printer char count: `NNNN`

When pretty-printing is done, the `NNNN` above indicates the number of characters in the output file.

If no error was detected, the pretty-printed text replaces the original text in the input file. Otherwise, it is left in a temporary file, which is indicated by the message:

ppolga: the paragraphed version of `text.olga` is in `/tmp/ppolgaNNNNNN`

where `NNNNNN` varies with each input file.

`ppolga` can pretty-print several files in a single run; just specify the names of all the files you want on the command line, as in e.g.:

tutorial% `ppolga *.olga`

However, to avoid spurious error messages, you must ensure that each file actually contains an OLGA text.

`ppasx` performs the same job, but for `asx` texts.

The command

```
tutorial% ppbnfatc grammar.bnf
```

pretty-prints a (concrete) grammar in *bnf* form (see chapter 21). The same command can be used for an already transformed grammar being in ATC form (see again chapter 21), then the file name extension should be `.atc`.

19.3 Modifying the Default Behavior

The behavior of `ppolga` and `ppasx` can be modified by specifying `options`, which can be found on the command line and/or in files.

19.3.1 Command syntax

The basic syntax to invoke `ppolga` and `ppasx` is as follows:

```
ppolga [ options... ] [ input_files... ]
ppasx [ options... ] [ input_files... ]
```

If no `input_files` argument is present, the standard input is read and the pretty-printed program is written on the standard output. Otherwise, each `input_file` is analyzed and, under default options, the corresponding pretty printed program is written on the same file if no error occurs.

19.3.2 Options

Options may appear in any order as long as they appear before the `input_files`. Only the last occurrence of a given option is taken into account.

- **–v**, **–verbose**
  Animate the user's screen by displaying cryptic information. (Default)

- –**nv**, –**noverbose**
  Execute silently.

- –**of** `output_file`, –**output_file** `output_file`
  Redirect program output. The default behavior is to overwrite each `input_file`, or, if none is given, to write on the standard output. Note that this option should usually be regarded as a debugging option, as only the last pretty printed program will be found in `output_file`.

- –**nof**, –**nooutput_file**
  Redirect program output to the standard output. This is the default option when input is done from the standard input.

- –**d** `path`, –**directory**`path`
  Immediately analyze the options file `path/`**.ppf2rc**, if that file exists (the directory itself must exist). See next section for further details.

- –**kwlc**, –**key_words_in_lowew_case**
  Pretty print keywords in lower case letters; excludes –**kwuc**and –**kwci**. (Default)

- –**kwuc**, –**key_words_in_upper_case**
  Pretty print keywords in upper case letters; excludes –**kwlc** and –**kwci**. This option should not be used if the resulting program is intended to be compiled, as OLGA and ASX keywords must be lower case to be recognized.

- –**kwci**, –**key_words_capitalized_initial**
  Pretty print keywords with each leading letter capitalized and the others written in lower case; excludes –**kwuc** and –**kwlc**. This option should not be used if the resulting program is intended to be compiled, as keywords must be lower case to be recognized.

- –**kwdk**, –**key_words_in_dark**
  Attempt to make keywords look darker than rest of text when displayed on a printer, by overstriking each letter twice (using backspaces). This option should not be used if the resulting program is intended to be compiled, as keywords will no longer be recognized.

- –**nkwdk**, –**nokey_words_in_dark**
  Do not artificially embolden keywords. (Default)

- –**ll** nnn, –**line_length** nnn
  Define the maximum length of an output line to be nnn. (Default: nnn = 122)

- –**mm** nnn, –**max_margin** nnn
  Specify that no output line should begin after column nnn. That value must be less than the line length. (Default: nnn = 60)

- –**sm**, –**shift_margin**
  Specify that when an output line should begin after the column defined by the –**mm** option, it is to be indented to the left, in order to preserve the structure of the program (more on this later). (Default)

- **–nsm**, **–noshift_margin**
  Specify that when an output line should begin after the column defined by the **–mm** option, it is to be output beginning in that column.

- **–tabs**
  Replace sequences of spaces by tabulation characters wherever possible outside string and comments. (Default)

- **–notabs**
  Do not produce tabulation characters.

- **–every** nnn
  Define the number of columns between two tabulation positions to be nnn. (Default: nnn = 8)

Default options are such that the command

```
ppolga
```

(for instance) is equivalent to

```
ppolga --v --kwlc --nkwdk --ll 122 --mm 60 --sm --tabs --every 8
   --d ${HOME} --d .
```

19.3.3 Options files

You may set up your own default options by creating a file named `.ppf2rc` in your home directory, and including whatever options you like. You may also add further default options related to the FNC-2 source files residing in the current directory by creating a `.ppf2rc` file there also. Both your home and the current directory are always searched, in that order, for options files. This search occurs before any command line option analysis, and thus cannot be skipped.

You may also specify other directories to be searched by using the **–directory** option, in options files as well as on the command line.

The options must be separated by white space or comments; a comment begins with a sharp character (TT#), either at the beginning of a line or preceded by a tabulation character, and ends with the line.

These options files are used by all the FNC-2 pretty-printers: `ppolga`, `ppasx` and `ppatc` if you use it (see Part VI).

The meaning of a **–directory** option is to immediately replace this option by the contents of the specified options file, if it exists; thus, recursion between options files leads to a loop, which is exited (gracefully) only when no more file can be opened by the pretty-printer.

19.4 Problems with Pretty-Printing

19.4.1 Line splitting

As all indenters, formatters, pretty-printers and other beautifiers, `ppolga` and `ppasx` have to meet the problem of physically bounded lines. The currently implemented way of resolving that stupid limitation is to put on a fresh line, indented by five spaces w.r.t. the current line, the first lexical unit which cannot be output on the current line. This is not deemed to be satisfactory, and the implementers of Paradis are currently studying a new strategy. The new strategy will be implemented in the FNC-2 pretty-printers as soon as it will be available.

19.4.2 Indentation restriction

In order to minimize the effect of line splitting the FNC-2 pretty-printers arbitrarily refuse to indent lines past a given column (defined by the −**max_margin** option). Their behavior on a line which should begin far to the right of the page is governed by the −**shift_margin** option: when this option is set, such a line is shifted to the left, and the program text keeps its structure; when the −**noshift_margin** option is in effect, the line is simply output beginning at the `max_margin` column. As a picture is always better than a long digression, here comes an example...

Suppose you have the following program fragment[2] deeply nested in your text, with the maximum margin indicated by the vertical bar in the comment:

```
...:= (  if $id(ID) = error-token then
            false
         elsif ...  then
            ...
         else
            true
         end if) & ($ok2(NUMBER) and if $value(NUMBER) = 0 then
                                        message "Null size"
                                           position NUMBER
                                           severity 2
                                           value false
                                     else
                                        true
                                     end if);
{max_margin -------->| }
```

On this example, not shifting the margin might produce something like this:[3]

```
...:= (  if $id(ID) = error-token then
               false
            elsif ...  then
               ...
            else
               true
            end if) & ($ok2(NUMBER) and if $value(NUMBER) =
               0 then
                     message "Null size"
```

[2]this is excerpted with some modifications from the example in appendix A.

[3]the awkward indentation of the line `0` **then** is due to line splitting described above.

```
                        position NUMBER
                        severity 2
                        value false
                     else
                        true
                     end if);
```

Shifting the margin would produce this kind of result:

```
...:= (  if $id(ID) = error-token then
            false
         elsif ...  then
            ...
         else
            true
         end if) & ($ok2(NUMBER) and if $value(NUMBER) =
                  0 then
            message "Null size"
               position NUMBER
               severity 2
               value false
                           else
                              true
                           end if);
```

The whole second **message** construct is shifted to the left[4] because of margin shift. The following lines are output with the right indentation: the **else** and the **end** are aligned with the corresponding **if**.

Choosing what to do here is mainly a matter of taste. Note that if you give to **-max_margin** the same value as the one you give to **-line_length**, only the process of line splitting as described earlier will be in effect.

19.4.3 Placement of comments

A comment—in the sense of chapter 6 of *the* OLGA *Reference Manual*—is attached by the scanner to the token which follows it; if this token is a generic terminal, then the comment is output by the pretty-printer before the token. But this token may be a non-generic terminal (e.g. a keyword or a separator) which does not appear in the syntax tree. In this case, the comment is attached by the parser, either as a pre-comment or as a post-comment, to the "closest" node of the syntax tree—either a non-terminal or a generic terminal—and is output just before or just after the construct corresponding to that node. During pretty-printing, comments may thus shift over non-generic terminals.

Also note that each comment is submitted to some cosmetic processing before being attached to some node. More precisely:

[4]The new margin is determined nearly arbitrarily.

- a comment containing only horizontal white space and/or newlines is completely erased, since the role of a pretty-printer is precisely to add horizontal white space and newlines to get a pleasant layout of the text;

- other, more meaningful comments have their leading and trailing white space trimmed, and also leading and trailing sequences of newlines are reduced to one newline[5]; however vertical spacing inside the comment is unchanged.

Non-empty comments containing no vertical spacing are output with one leading and one trailing (horizontal) space. Other non-empty comments are output on a fresh line, indented to the margin of the following construct.

It is guaranteed that pretty-printing a pretty-printed text—with the same options regarding line length, maximum margin and margin shift—will produce the same result as the original, i.e. `ppolga` and `ppasx` are idempotent.

Placement of comments is in any case a difficult problem, for which there is no "best" solution but only less bad solutions.

[5]This is because we believe that the fact that a comment begins and/or ends on a fresh line rather than on the same line as the text which precedes or follows is significant and must be maintained in the pretty-printed text. However the number of empty lines between two pieces of text must be controlled by the pretty-printer.

Part IV

PPAT: Pretty Printer for Attributed Trees

Chapter 20

PPAT Reference Manual

Introduction: a Pretty-Printer Language for Attributed Trees

The PPAT language provides an FNC-2 user a tool enabling concrete representations of attributed trees manipulated in FNC-2 . PPAT design is – deliberately – quite close to the OLGA language, both syntactically and semantically. In PPAT, we wanted to keep the expression power of OLGA so that *decompilation* of complex trees be possible with maximal flexibility. The applied formalism is inspired by that of PPML (Pretty–Printer Meta–Language, [Bor88]), a language to decompile (non-attributed) abstract trees.

20.1 PPAT Language Principles

A PPAT program specifies how an attributed abstract tree of a given AAS will be decompiled. Chapter 17 describes the definition of the AAS of an abstract tree by means of ASX (ASX specification). The basic principle is schematized in Fig. 20.1.

In the specification of an attributed grammar in OLGA, for each tree operator a set of semantic rules is defined; similarly, in a PPAT decompilation specification, for each tree operator an *editing format* is defined. Like in OLGA, a tree node is symbolized by the operator of the node.

To describe an *editing format*, a box language is used, similar to that of PPML. A box can be of three types:

- *empty*, a box whose height and width are zero,
- *simple*, a type to represent an individual object,
- *composite*, a type corresponding to a list of boxes positioned relative to each other by means of a *separator*.

PPAT makes possible to decompile attributed trees (unlike PPML). To do that, the user must provide a corresponding editing format to each complex type to be displayed. This format is described the same way as the one used for tree operators.

Consequently, a PPAT program is a set of rules:

$$\begin{array}{lcl} < \mathit{operator} > & \leadsto & < \mathit{box} > \\ < \mathit{type} > & \leadsto & < \mathit{box} > \end{array}$$

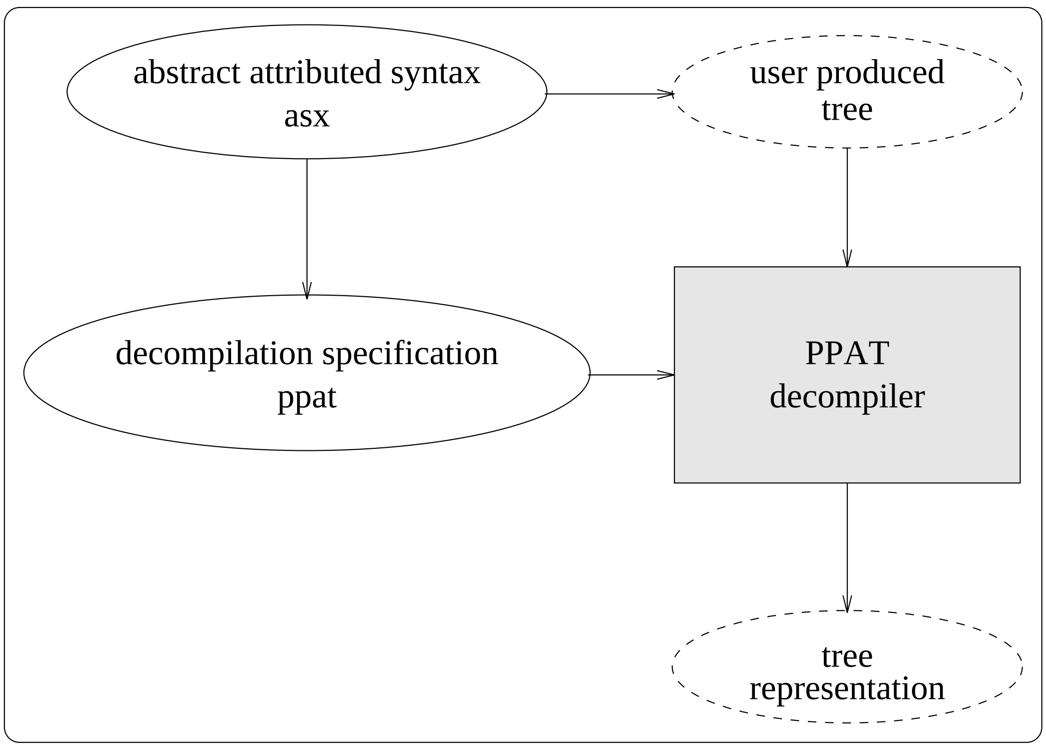


Figure 20.1: Ppat general application schema

This basic principle is extended by the next facilities:

- control structures, similar to those of Olga, allowing to create formats governed by values of attributes, expressions, the type of a tree etc.
- Decompilation-specific information can be transmitted in the tree by means of **context**s: they behave like inherited attributes.
- The form of Ppat expressions is similar to that of Olga expressions. In particular, functions defined by the user in external modules can be called.

20.2 Lexical definition

The lexical definition of Ppat is based on that of the Olga language. In particular, `%ID` and `%ATT` are the same. Separators and operators are as follows:

`!  != !   % & ( ) * + , - -> ./ :   := ; < <= = > >= [ ] |   "`

Reserved words are:

$arity $position $scx and case context else elsif end first global if insert last left match or then type value where wheretype

20.3 Syntactic and semantic description of Ppat

Now we describe the syntactic definition of a Ppat program as it can be seen in the *atc* specification of Ppat (see in appendix E) and, at the same time, a detailed explanation of notions is given.

20.3.1 General program structure

The concrete grammar giving the general structure of a PPAT program is:

Syntax:

```
<ppat prog>          =  <header>
                        <decls>
                        <rules>
                        end ppat ";" ;
<header>             =  ppat <ID> "(" <ID> ")" ";" ;
```

That is, a PPAT program consists of a header, declarations and rules. The header has two identifiers:

- The first one is the *internal name* of the PPAT module, that is, its name with respect to FNC-2 .
- The second is the (internal) name of the module containing the definition of the ASX module to be decompiled.

The declarations are defined by:

Syntax:

```
<decls>              =  <decls>
                        <decls sep> ;
<decls>              =  <decls>
                        <decls cxt> ;
<decls>              =  <decls>
                        <import clause> ;
<decls>              =  ;
```

That is, a PPAT program has a (possibly empty) list of declarations where the user can define separators, contexts and module importations.

20.3.2 Importing a module

Importing a module is done like in OLGA, that is:

Syntax:

```
<import clause>      =  from <ID> import all ";" ;
```

By this clause the user makes accessible, in a PPAT module, all the type and function objects defined in the module of the internal name corresponding to the identifier.

20.3.3 Declaration of separators

In section 20.1 it is stated that composite boxes are defined by a separator. These separators are similar to constants and they must be defined before use. Their definition corresponds to:

Syntax:

```
<decls sep>          =  separator
                        <decl sep +> ;
<decl sep +>         =  <decl sep +>
                        <sep def> ;
<decl sep +>         =  <sep def> ;
<sep def>            =  "<" <SEP ID> ">" "=" <separator> ";" ;
<separator>          =  "<" <ID> <sep param *> ">" ;
<sep param *>        =  <sep param +> ;
<sep param *>        =  ;
<sep param +>        =  <sep param +> "," <sep param> ;
<sep param +>        =  <sep param> ;
<sep param>          =  <INT> ;
```

The declaration of a separator corresponds to the assignment of a separator description to an identifier: <SEP ID>. The description of a separator itself consists of two parts:

1. the type of the separator : h , v , hv or hov
2. the parameter list of the separator (integers).

The description of four types of separators follows together with their parameters.

20.3.3.1 Horizontal separator <h dx>

Two boxes B1 and B2 composed by the separator <h dx> are placed so that the first line of B2 is aligned with the last line of B1 by a horizontal space of dx characters (see Fig. 20.2). The advantage of this setup is that the produced text is analyzable since the text is read from left to right and from top to bottom; it wouldn't be the case if we aligned the top of the two boxes.

20.3.3.2 Vertical separator <v di , dh>

Two boxes B1 and B2 composed by the separator <v di , dh> are placed vertically separated by a vertical space of dh lines and with an indentation of di characters relative to the enclosing box (see Fig. 20.3).

20.3.3.3 Horizontal and vertical separator <hv dx , di , dh>

Two boxes B1 and B2 composed by the separator <v di , dh> are placed in such a way that:

- if B2 is aligned with the last line of B1 by a horizontal space of dx characters and fits into the display frame then they are displayed so;

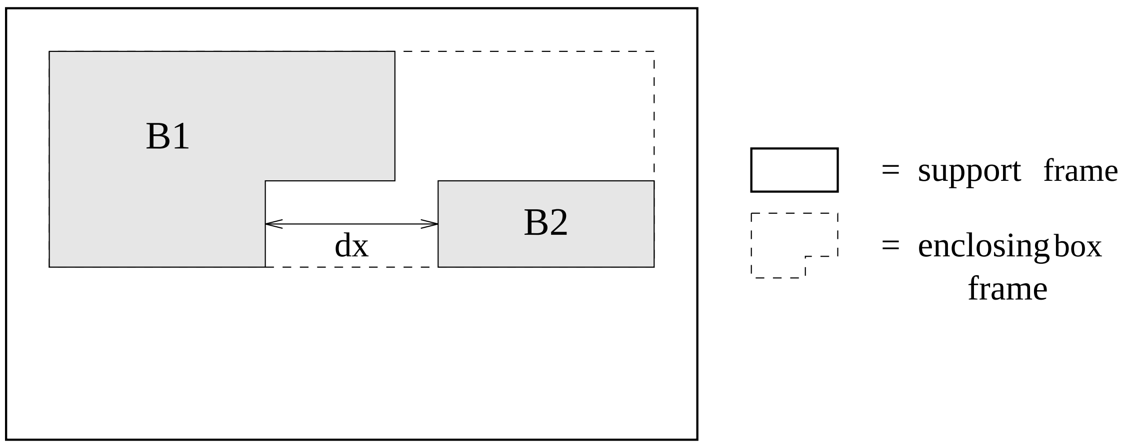


Figure 20.2: Composition of two boxes by `<h dx>` separator

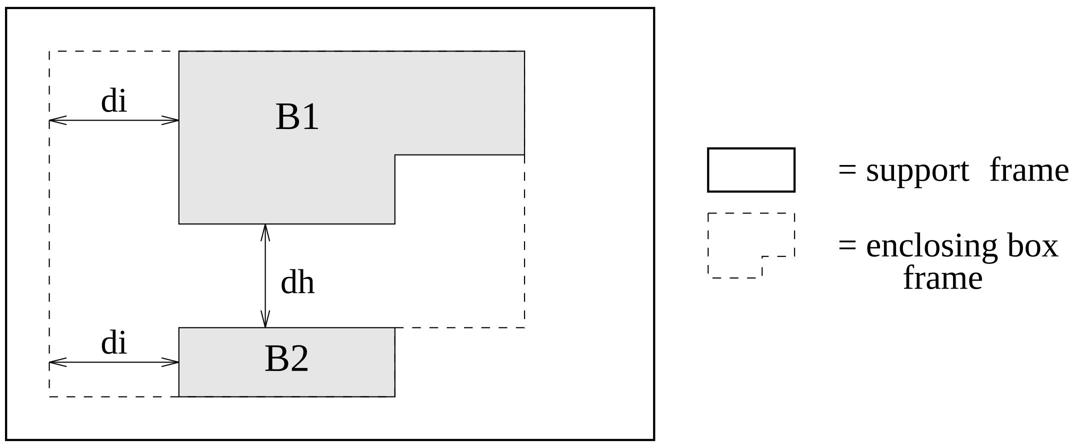


Figure 20.3: Composition of two boxes by `<v di , dh>` separator

- if not, then B2 is placed `dh` lines under B1 by an indentation of `di` characters relative to the enclosing box.

Fig. 20.4 shows the result of composing three boxes by the `<hv dx , di , dh>` separator illustrating both ways of placement.

20.3.3.4 Horizontal or vertical separator `<hov dx , di , dh>`

A list of boxes composed by the separator `<hov dx , di , dh>` are placed:

- like in the case of the `<h dx>` separator if this setup fits the display frame;
- like in the case of the `hv` separator if horizontal disposition does not fit the display frame.

The difference between this and the `hv` separator is that here the decision of placement is global with respect to the list of boxes; but at the `hv` separator this decision is taken for each box. Fig. 20.5 shows the two possible results when composing three boxes. The last two types

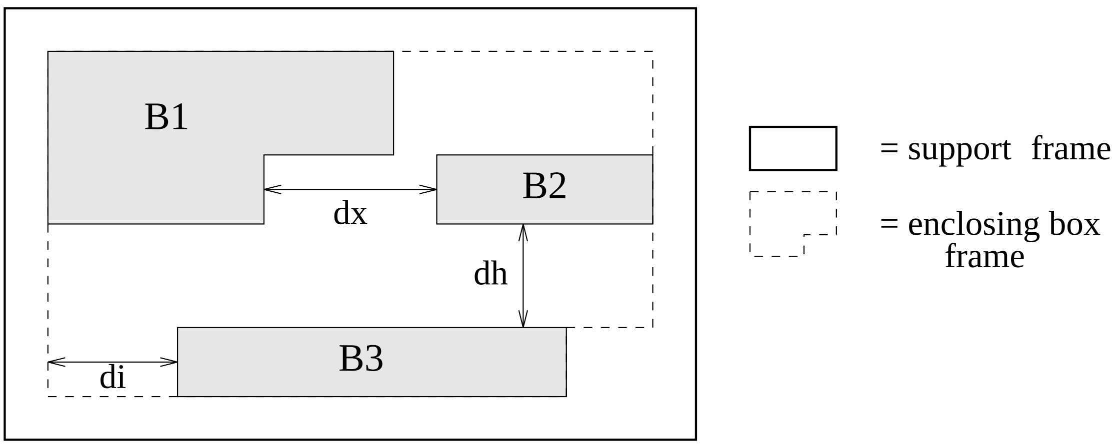


Figure 20.4: Composition of three boxes by the `<hv dx,di,dh>` separator

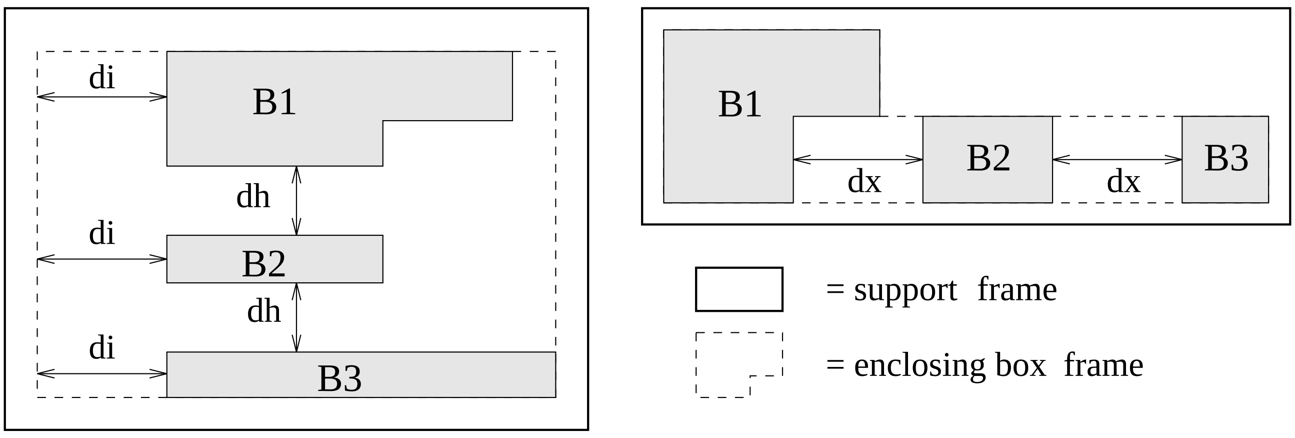


Figure 20.5: The two possible ways when composing three boxes by the `<hov dx , di , dh>` separator

of separators are conditional ones, that is, the produced result depends on the width of the display frame; however, the first two ones are *rigid* so capable of causing *overflow*.

20.3.4 Context declaration

As mentioned in section 20.1, it is possible to transmit decompilation–related information by means of **context**s. These **context**s are considered as inherited attributes in the OLGA sense. Context declaration is described by:

Syntax:

```
<decls cxt>          =  context
                        <decl cxt +> ;
<decl cxt +>         =  <decl cxt +>
                        <cxt def> ;
<decl cxt +>         =  <cxt def> ;
<cxt def>            =  <attrib name> "(" <phyla, *> ")" ":" <ID> ";" ;
```

```
<phyla, *>            =  <phyla, +> ;
<phyla, *>            =  ;
<phyla, +>            =  <phyla, +> "," <phylum> ;
<phyla, +>            =  <phylum> ;
```

These declarations are similar to attribute declarations in OLGA: the **attribute** keyword of OLGA is substituted by the **context** keyword. A context is defined by giving its name, the list of phyla it is defined over and its type. Unlike in OLGA, the context class is not specified because all contexts are, by definition, inherited.

A context can be of predefined or of complex type. In the latter case the corresponding type must be defined in a imported declaration module as we have seen in subsection 20.3.2.

20.3.5 Formatting rules on operators

Having declared all the necessary objects in the definition of a PPAT module, we go on by studying the principal part of a PPAT specification, that is, the formatting rules on operators. Such a rule is defined by:

Syntax:

```
<rule op *>           =  <rule op *>
                         <rule op> ;
<rule op *>           =  ;
<rule op>             =  where <production> pprint
                            <top box>
                         end where ";" ;
<production>          =  <ID> "->" <ID> "+" ;
<production>          =  <ID> "->" <ID> "*" ;
<production>          =  <ID> "->" <phyla *> ;
<phyla *>             =  <phyla +> ;
<phyla *>             =  ;
<phyla +>             =  <phyla +> <ID> ;
<phyla +>             =  <ID> ;
<top box>             =  <box> ;
<top box>             =  ;
<box>                 =  <string box> ;
<box>                 =  <attrib occ> ;
<box>                 =  <rec call> ;
<box>                 =  <conditional> ;
<box>                 =  <compound> ;
<box>                 =  <case match> ;
<box>                 =  <pplist> ;
```

The description of the operator, as mentioned in section 20.1, is identical with the description of operators in OLGA. As far as the editing format is concerned, the description of a box can belong to one of the next formats:

- composite box
- character chain (string) box
- recursive call
- attribute occurrence
- operator list handling
- conditional box
- box controlled by expression pattern matching

Now we describe all these different formats.

20.3.5.1 Composite box

The composite box represents, as mentioned in section 20.1, the effect of a separator on a list of sub–boxes. This format is described by:

Syntax:

```
<compound>          =  "[" <SEP ID> <box +> "]" ;
<box +>             =  <box +> <local sep> <box> ;
<box +>             =  <box> ;
<local sep>         =  "<" <SEP ID> ">" ;
<local sep>         =  ;
```

Each sub–box may have an arbitrary format from the choice mentioned beforehand. In order to diminish the number of brackets, local separators can be used. They mean a shorthand convenience only, as follows:

$$\begin{array}{rl} & [\texttt{sep}\ b_1\ \ldots\ b_i < \texttt{seploc} > b_{i+1} \ldots\ b_n] \\ \Leftrightarrow & [\texttt{sep}\ b_1\ \ldots\ [< \texttt{seploc} > b_i\ b_{i+1}]\ \ldots\ b_n] \end{array}$$

20.3.5.2 Character chain (string) box

This box is described trivially by:

Syntax:

```
<string box>        =  <STRING> ;
```

it represents a box of height of one unit and its width is equal to the number of characters in the chain.

20.3.5.3 Recursive call

The recursive call represents a reference to a variable (typed by phylum) appearing in the definition of an operator. The corresponding box is the box resulting from the decompilation of the tree associated to the variable mentioned. This format is described by:

Syntax:

```
<rec call>          =  <symbol> <context list> ;
<symbol>            =  <ID> ;
<symbol>            =  <ID> "." <qualifier> ;
<context list>      =  "(" <context *> ")" ;
<context list>      =  ;
<context *>         =  <context +> ;
<context +>         =  <context +> "," <context>;
<context +>         =  <context> ;
<context>           =  <attrib name> ":=" <expr>;
```

In order to allow the decompilation of a sub–tree, depending on the informations contained in the current node, the value of the contexts (by which this sub-tree must be decompiled) can be specified. In fact, it is obligatory to give a value to all the contexts attached to the variable (typed by phylum) representing the subtree: remember that contexts are, indeed, inherited attributes. The value of a context is described by means of a PPAT expression: we will study these expressions in subsection 20.3.7.

It is not obligatory to present all the variables (typed by phyla) of an operator definition in its associated editing format: this allows for ignoring certain subtrees (the fact of not mentioning a variable results in the non-decompilation of its corresponding subtree).

20.3.5.4 Attribute occurrence

We claimed that PPAT's advantage over PPML is based on the fact that PPAT decompiles attributed abstract trees. This form allows to visualize the value of tree attributes the following way:

Syntax:

```
<attrib occ>        =  <attrib name> ;
<attrib occ>        =  <attrib name> "(" <expr> ")" ;
<attrib occ>        =  <attrib name> "(" <expr> "." <qualifier> ")" ;
<expr>              =  <ID> ;
<qualifier>         =  left ;
<qualifier>         =  right ;
<qualifier>         =  first ;
<qualifier>         =  last ;
```

As in OLGA, the attributes capable of appearing in the edition format associated to an operator are local with respect to the production defining this operator, that is, either the attributes attached to the phylum where the operator belongs or the attributes attached to the offspring phylum of this operator.

An attribute occurrence is represented, like in OLGA, by the name of the attribute followed by the name of the variable concerned (typed by phylum), in parentheses. If the symbol is omitted, the attribute is considered as being an attribute of the phylum where the operator belongs to.

It is possible to mention all the attributes of a phylum, be they those of the input tree or even the contexts defined by this phylum.

An attribute is decompiled according to the rules associated to its type inside of the specification.

20.3.5.5 Operator list handling

We have seen before that the operators of an ASX specification may have fixed arity or they can even be list operators. Just like in OLGA, there is a special treatment of list operators. It is described by:

Syntax:

```
<pplist>          =  pplist <direction>
                        global <SEP ID>
                        <insert>
                        subbox <box>
                     end pplist ;
<direction>       =  left ;
<direction>       =  right ;
<insert>          =  insert <STRING> ;
<insert>          =  ;
```

The box corresponding to such a format is, in fact, a composite box; its separator is declared after the **global** keyword and its list of sub–boxes are constructed the following way: to each element of the list being processed, a box is associated, defined by:

- if the current element is not the last processed then it is a composite box with `<h 0>` separator. The first sub–box is the box associated to the format given after the **subbox** keyword and the second one is the character chain box associated to the chain following the **insert** keyword (eventually, the empty chain if the **insert** keyword is omitted).
- if the current element is the last processed then it is the box associated to the format given after the **subbox** keyword.

The order in which the processed list is visited is given by the two keywords **left** or **right**. So, if the next format is defined:

```
where op -> PH * pprint
        pplist left
           global <h1>
           insert ","
```

```
        subbox PH
      end pplist
end where ;
```

If

$$\texttt{sep} - \texttt{nul} =< \texttt{h0} >;$$

then during the decompilation of the tree the box associated to a node of arity n with a root `op` can be conceived as:

$$[< \texttt{h1} > \texttt{PH.1} < \texttt{sep} - \texttt{nul} > "," \texttt{PH.2} < \texttt{sep} - \texttt{nul} > "," \ldots \texttt{PH.n}]$$

If **left** was replaced by **right** in the format, the result would be:

$$[< \texttt{h1} > \texttt{PH.n} < \texttt{sep} - \texttt{nul} > "," \texttt{PH.n} - 1 < \texttt{sep} - \texttt{nul} > "," \ldots \texttt{PH.1}]$$

20.3.5.6 Conditional box

Until now, we only presented rigid editing formats. But it is necessary to introduce control structures so that the way the tree is decompiled be controllable. The conditional box is the simplest form of decompilation control, defined by:

Syntax:

```
<conditional>         =  if <expr> then
                              <box>
                          <elsif part>
                          else
                              <box>
                          end if ;
<elsif part>          =  <elsif part>
                          elsif <expr> then
                              <box> ;
<elsif part>          =  ;
```

This structure is, of course, quite close to those of OLGA conditional expressions. Condition expressions are PPAT boolean expressions. They can be constructed using all internally visible elements of a rule, in particular the local attribute occurrences of this production, the contexts and the constants.

20.3.5.7 Box controlled by expression pattern matching

The second form of control is that corresponds to *pattern matching* of OLGA on expressions. The definition of this control structure is given by:

Syntax:

```
<case match>          =  case <expr> is
                            <match clauses>
                            <match default>
                         end case ;
<match clauses>       =  <match clauses>
                         <match clause> ;
<match clauses>       =  <match clause> ;
<match clause>        =  <case clauses> ":"
                         <box> ";" ;
<case clauses>        =  <case clauses> "," <case clause> ;
<case clauses>        =  <case clause> ;
<match default>       =  other ":"
                         <box> ";" ;
<case clause>         =  <root pattern> ;
<root pattern>        =  <pattern cons> ;
<root pattern>        =  <ID> ;
<root pattern>        =  <constant manifeste> ;
<root pattern>        =  const <expr> ;
<pattern cons>        =  <ID> "(" <patterns *> ")" ;
<patterns *>          =  <patterns +> ;
<patterns *>          =  ;
<patterns +>          =  <patterns +> "," <pattern> ;
<patterns +>          =  <pattern> ;
<constant manifeste>  =  <INT> ;
<constant manifeste>  =  %REAL ;
<constant manifeste>  =  %CHAR ;
<constant manifeste>  =  %STRING ;
<pattern>             =  <root pattern> ;
```

Like in OLGA, the use of patterns using variables allows for linking these variables to the matching part of the expression. Expressions can be general, that is, tree expressions and so tree expression pattern matching is possible, too.

20.3.6 Formatting rules on types

We have mentioned, in paragraph 20.3.5.4, that the decompilation of an attribute is done using the format associated to its type. The definition of a formatting rule for a type is as follows:

Syntax:

```
<rule type *>         =  <rule type *>
                         <rule type> ;
<rule type *>         =  ;
<rule type>           =  wheretype <ID> pprint
                            <top box type>
                         end where ";" ;
```

```
<top box type>        =  <box type> ;
<top box type>        =  ;
<box type>            =  <string boxt> ;
<box type>            =  <compound boxt> ;
<box type>            =  <fieldref boxt> ;
<box type>            =  <pplist boxt> ;
```

These rules are very close, both on definition and on functionality level, to the rules of operator formatting. Like in the case of operators, several formats exist:

- character chain (string) box
- composite box
- access to a field of record type
- list type handling

The first two formats are similar to those corresponding to operators therefore we describe the last two ones only.

20.3.6.1 Access to a field of record type

During the decompilation of an attribute of type record it is possible to access fields of this type by indicating the name of the field. It can be conceived as the format of recursive operator decompilation even though there is no corresponding type notion for contexts.

20.3.6.2 List type handling

The principle of list type handling can be conceived as that of list operators:
Syntax:

```
<pplist boxt>         =  pplist <ID> <direction>
                            global <SEP ID>
                            <insert>
                            subbox <box type>
                         end pplist ;
```

We included an identifier in the format of list handling. It allows for referring the current element of the list in the sub–box format. So this identifier plays the role of the offspring phylum of a list operator.

20.3.7 PPAT expressions

We intended to give PPAT great expressive power, conserving, at the same time, the relative simplicity of decompilation specifications. For this purpose, in addition to classical arithmetic expressions, function calls are also permitted. These functions are defined by the user in external modules, so as not to over-charge decompilation specification but permitting to benefit from all the expressive power of OLGA.

Syntax:

```
<expr>               =  <INT> ;
<expr>               =  <REAL> ;
<expr>               =  <CHAR> ;
<expr>               =  <STRING> ;
<expr>               =  <ID> ;
<expr>               =  "(" <expr> ")" ;
<expr>               =  <function call> ;
<expr>               =  <attrib occ> ;
<expr>               =  <expr> "+" <expr> ;
<expr>               =  <expr> "-" <expr> ;
<expr>               =  <expr> "*" <expr> ;
<expr>               =  <expr> "/" <expr> ;
<expr>               =  <expr> "%" <expr> ;
<expr>               =  <expr> "=" <expr> ;
<expr>               =  <expr> "!=" <expr> ;
<expr>               =  <expr> "~" <expr> ;
<expr>               =  <expr> "!~" <expr> ;
<expr>               =  <expr> "<" <expr> ;
<expr>               =  <expr> ">" <expr> ;
<expr>               =  <expr> "<=" <expr> ;
<expr>               =  <expr> ">=" <expr> ;
<expr>               =  <expr> "&" <expr> ;
<expr>               =  <expr> "|" <expr> ;
<expr>               =  <expr> "and" <expr> ;
<expr>               =  <expr> "or" <expr> ;
<expr>               =  "-" <expr> ;
<expr>               =  "+" <expr> ;
<expr>               =  "!" <expr> ;
<function call>      =  <ID> "(" <actual *> ")" ;
<actual *>           =  <actual +> ;
<actual *>           =  ;
<actual +>           =  <actual +> "," <actual> ;
<actual +>           =  <actual> ;
<actual>             =  <expr> ;
<attrib occ>         =  <attrib name> ;
<attrib occ>         =  <attrib name> "(" <expr> ")" ;
<attrib occ>         =  <attrib name> "(" <expr> "." <qualifier> ")" ;
<attrib name>        =  %ATTR ;
<INT>                =  %INT ;
```

```
<CHAR>              =  %CHAR ;
<STRING>            =  %STRING ;
<ID>                =  %ID ;
<SEP ID>            =  %ID ;
<REAL>              =  %REAL ;
<phylum>            =  %ID ;
```

Part V

Generating Abstract Trees with the SYNTAX System

Chapter 21

Introduction and Presentation of SYNTAX

21.1 Why a Tree Constructor?

The FNC-2 system and its description language OLGA deal with attribute evaluators which take as input a tree described by some AAS. They are thus not interested in the way this tree is constructed, provided of course that it actually corresponds to the input AAS. FNC-2 cannot thus be used as a stand-alone system because this tree must be constructed by some external processor. This part describes how to to do this by parsing some source text and constructing the corresponding tree, using an analyzer generated by the SYNTAX system and an associated tree constructor generated by the tree constructor generator which is the topic of this Part, and which is named ATC. Considering that FNC-2 will mostly be used to produce (parts of) compilers and other language processors, this is the usual way to construct the initial tree. Appendix A describes how to link FNC-2 with the programming environment Mentor, in which trees are constructed interactively using a structured editor.

21.2 The SYNTAX System

This section is a brief presentation of the SYNTAX system for more details refer to [BD88]. Figure 21.1 is a functional diagram of SYNTAX and ATC, to be presented in the following chapters.

The main goal of SYNTAX is to generate efficient and powerful scanners and parsers from high-level descriptions of the languages to be recognized: regular expressions for the lexical part, and context-free grammars for the syntactic part.

21.2.1 The grammar reader *bnf* and semantic processing

The grammar reader *bnf* accepts a context-free grammar and transforms it into an internal form (a set of tables) to be used by the other constructors. *Bnf* ignores the pretty-printing specifications embedded in the productions, to be used by the pretty-printer generator *paradis*, and also all the "semantic" part of its input, i.e. everything comprised between two productions. This feature allows to use *bnf* either as a stand-alone processor or as a component of a more comprehensive constructor which would process, in addition to the purely syntactic grammar, some kind of semantic specification. What is important to note here is that the only connection between the parser and the semantic processing is done through *actions* called upon each reduction (SYNTAX uses a bottom-up LR parsing method). SYNTAX itself provides a number of such semantic processing methods:

- *action numbers* associated to each production *(semact)*;
  the semantic actions can be written in any language and must be in a separate file; they will access the parse stack in a "low-level" fashion;

- *actions "a‘ la Yacc" (yax)*,
  which are written in C and appear in the same file as the grammar; a limited form of purely synthesized attribute handling is provided (the $ notation); SYNTAX provides a translator from Yacc specifications to *yax* ones, named *ysx*;

- *abstract tree construction (semat)*;
  this is an "old" version, not to be mistaken with ATC, in which the same specification defines the abstract syntax and the way the corresponding trees are constructed; *semat* also produces skeletons of procedures to traverse the generated trees in a top-down, left-to-right manner;

- *purely synthesized attribute grammars (tabact)*,
  not to be mistaken either with FNC-2 , and in which the semantic rules are written in C;

- and *pretty-printer (paradis)*;
  this "semantic processing" method is special in that it can be used in parallel with one of the previous ones, i.e. the same input file may contain the syntactic grammar, the pretty-printing specification and one of the semantic specifications altogether.

The tree construction scheme presented in this Part acts, with respect to SYNTAX, as a new kind of semantic processing method, as shown in figure 21.1; the corresponding constructor is named ATC.

Bnf reads the grammar written in a BNF-like notation and checks that it is reduced. A grammar is *reduced* if, on one hand, all the non-terminals are reachable from the start symbol (or axiom) of the grammar and if, on the other hand, all the non-terminals can derive a terminal string. *Bnf* also detects some trivial conditions of ambiguity, e.g. two productions are identical or a non-terminal derives into itself. It also constructs the *followers* relation stating which terminals are valid followers of a given terminal, and builds a set of tables which will be used by other processors.

In addition to the actions performed upon each reduction (the "semantic" processing), SYNTAX allows the insertion of predicates and actions *inside* the productions. These can be used to process context-dependent languages, or languages whose syntax is dynamically modified by some constructs, e.g. the **typedef** declarator of C whose effect is to add new type keywords. These actions and predicates are referenced by numbers. The code implementing them must be written in C and reside in separate files.

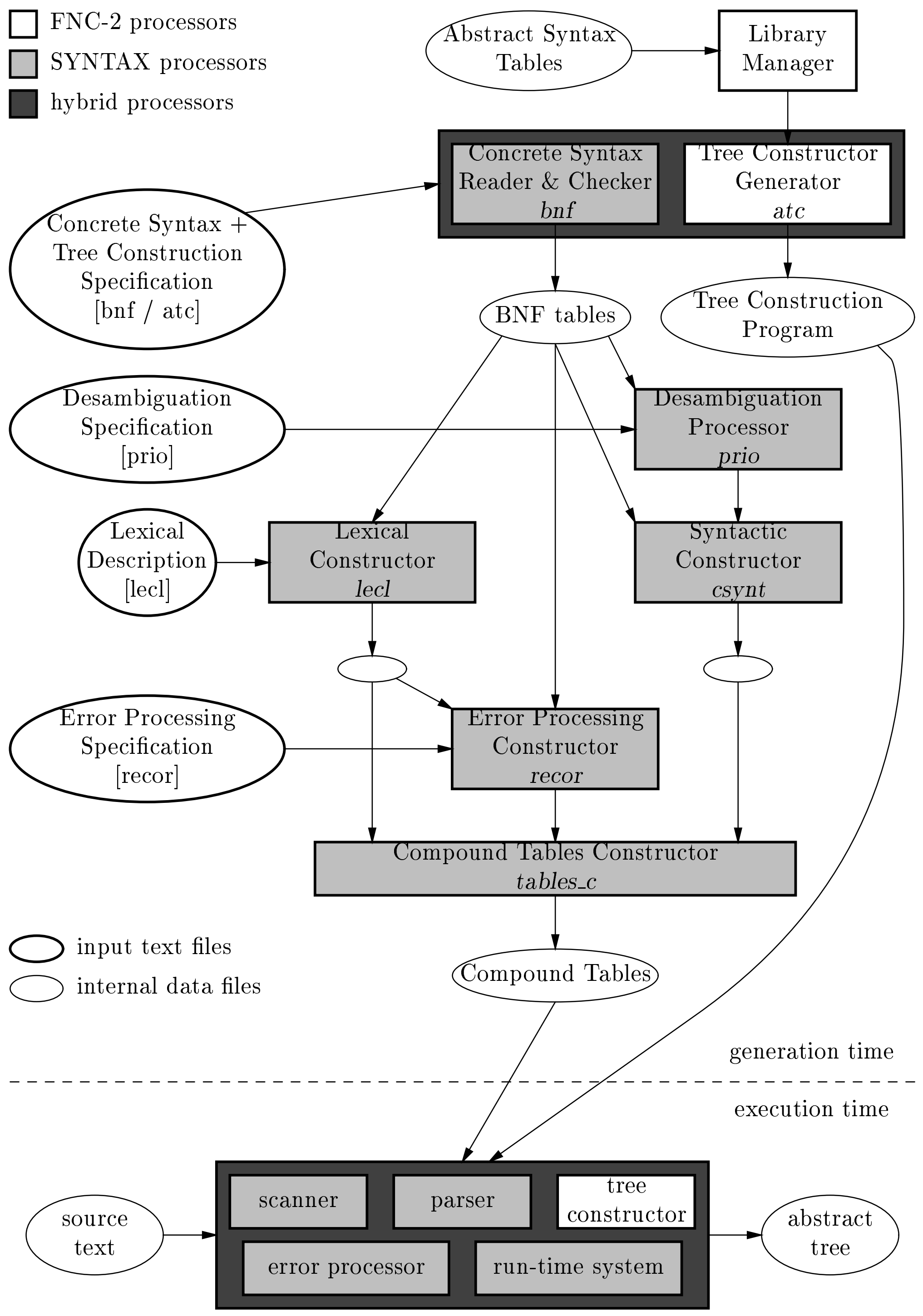


Figure 21.1: SYNTAX and Atc

21.2.2 The lexical constructor *lecl*

Next comes the lexical constructor *lecl*. It reads the internal form of the grammar, in particular the list of terminals and the *followers* relation (context table), and also the *lexical description* of the tokens of the source language. These tokens are described by *regular expressions* whose operands are *character classes*. Operations on regular expressions are sequence (concatenation), alternative, repetition—never-empty `(+)` and perhaps-empty `(*)`—, negation (on character classes only), and grouping.

A lexical description is composed of several chapters:

- "Classes" allows to define named character classes, which are sets of lexically equivalent characters. A number of usual classes (letters, digits, etc.) are predefined in *lecl*. Operations provided on character classes are the construction of a class from individual characters and ranges of characters, and the union and difference of two classes.
- "Abbreviations" allows to give names to regular expressions; these names can then be used in this and the next chapter as abbreviations for the corresponding regular expression (textual replacement). It is also possible to give names to predicates and actions (see below); operations on predicates are conjunction (sequence), disjunction (alternative), and negation.
- "Tokens" allows to define the tokens of the source language by regular expressions; this is the only mandatory chapter. It is mandatory to give regular expressions describing *comments* and the *generic* terminals of the grammar, i.e. those which do not have a fixed text (e.g. identifiers, numbers, ...), the names of which begin with a percent sign (%) in the grammar. By default, the other terminals have the text given in the grammar, but they may be redefined.
- "Synonyms" allows to give, at the lexical level, *synonyms* to some keywords; in PL/1 for instance, **proc** is a synonym of **procedure**.
- Representation clauses allow to generate a scanner for a machine with a different word and/or byte size or with a different alphabet (e.g. EBCDIC) than the host machine.

Lecl uses LR-like construction techniques which permit to obtain directly a deterministic finite-state automaton, allowing the use of an **unbounded** number of look-ahead characters to solve the conflicts. This resolution may also be helped statically using *priority* clauses attached to some regular expressions, and dynamically using predefined and/or user-defined *predicates* and *actions* for context-dependent analysis. However, by default, *lecl* tries to build a scanner using only one character of look-ahead; system-defined rules are then used to solve the conflicts.

Predicates and actions may also be used for local processing of the recognized tokens, e.g. to transform them into a normal form (upper case in Pascal, where lower and upper case are equivalent).

Lecl builds a perfect and minimal hashing function over the set of keywords[1] in order to recognize them with a single comparison.

---

[1]A keyword is a non-generic terminal having no explicit regular expression in the lexical description, but whose text, as defined in the grammar, is recognized by the regular expression defining another (generic) terminal. For instance, when describing Pascal, you will probably not explicitly describe how to recognize **begin**, but it will be recognized by the regular expression defining identifiers; hence it is a keyword.

The finite-state automaton generated by *lecl* is represented, together with associated information, by a set of tables. It is also possible to generate a specific scanner, written in C, which will replace the interpreter provided in the run-time kernel (see further).

21.2.3 The syntactic constructor *csynt* and the disambiguation processor *prio*

The internal form of the grammar is processed by the syntactic constructor *csynt*, which is actually composed of two modules. The first one, *lalr1*, builds a bottom-up stack automaton (parser) recognizing the language, using the LALR(1) method. If conflicts are detected during this construction—which means that the grammar does not belong to the LALR(1) class—, several rules are used to solve them and continue the construction: there are built-in rules and user-defined rules. The latter are written in a separate file using a *yacc*-like style, by defining the relative priorities and associativity rules of some “operators”; it is also possible to assign a priority to some production. User-defined rules are processed by the *prio* constructor, and take the precedence over system-defined rules.

User-defined predicates and actions may also be embedded in the productions and used during parsing to perform context-dependent analysis.

The resulting automaton is represented as a set of tables. It is highly optimized; in particular, *simple productions*—i.e. productions of the form `<A> = <B>;` with no explicit semantics—are totally eliminated, and reductions valid in only one context are executed without checking the look-ahead symbol (it will be checked at next shift transition).

The second component of *csynt*, named *optim*, reduces the size of the tables representing the automaton by a very important factor (typically more than 95%).

21.2.4 Error recovery

SYNTAX implements in its parser and scanner a powerful *error recovery* mechanism [BJ87]. The first level of the scheme tries to devise a *local correction* of the erroneous text. Correct pieces of text are generated and matched against a list of *patterns* which can specify any number of deletions, insertions and/or replacements of tokens or characters. The correction is validated by checking a number of tokens following the modified text. Sets of *don't insert* and *don't delete* tokens and characters may be defined in order to avoid spurious errors after a correction. As for parsing this mechanism is enhanced by the possibility to act on the token preceding the error point, and to check spelling errors on keywords.

If local correction fails, a *global recovery* is initiated. The source text is skipped up to a *key terminal*; the parse stack is popped until a state having a valid transition on this terminal is found, and the parser is restarted in this state with that key terminal as input.

The behavior of this error handling mechanism is parameterized by the grammar writer through a specification, processed by the *recor* constructor. In particular, the messages which are emitted when an error is detected are completely specified in that description, so that they can be tailored to a particular language (English, French, ...) or a particular class of users (beginners, students, experts, ...).

21.2.5 The tables constructor *table_c* and the run-time kernel

The set of tables generated by the various constructors is gathered by *table_c* and translated into a set of C structures, to be compiled and linked with the SYNTAX *run-time kernel* to form a complete analyzer. This kernel includes a scanner and a parser, which will interpret the automatons represented by the tables, and many other utility modules: string table manager, comments manager, error messages manager, source file manager, etc. It must also be linked with the module dealing with semantic processing, which is in our case the tree constructor. This process is described in chapter 23.

The run-time kernel is presently written in C, but other versions will perhaps be available in the future.

Chapter 22

The Tree Construction Language

22.1 Introduction to the Tree Constructor

The basic constraints and rationale for the design of a new tree constructor which will be the link between SYNTAX and FNC-2 are as follows:

- the trees to be generated must be described by an FNC-2 attributed abstract syntax
- they must be constructed bottom-up and thus independently from the context;
- however the tree construction actions must be powerful enough to deal with syntactic constraints, which are apparent at the concrete level but forgotten at the abstract one;
- the specification must be in an applicative style and resemble OLGA.

Since this is not a new problem, many solutions already exist, in particular in the Mentor programming environment [DGHK+80] developed at INRIA. For obvious proximity reasons, we decided to reuse as much experience as possible from Mentor. Our tree construction language, named ATC, is thus strongly inspired by Metal [KLM+83].

ATC also has very strong connections with OLGA; actually, it is nearly a subset of OLGA reduced to the features necessary for (output) tree construction and decoration as described in chapter 14 of *the* OLGA *Reference Manual*. The other main differences with OLGA are as follows:

- ATC is based on a concrete syntax rather than on an abstract one; however the generated trees are described by an AAS;
- each non-terminal has exactly one anonymous, synthesized, tree-valued attribute
- ATC doesn't not provide for defining new attribute types; actually, apart from tree types (phyla of the output AAS), ATC can only manipulate values of types *token* and *source-index*, and furthermore in a limited way.

22.2 Structure of a Tree Construction Specification

A tree construction specification is composed of a header specifying the output AAS which describes the generated trees, of optional declarations specifying new phyla and/or operators which will be used temporarily during tree construction, but which must eventually disappear, of optional functions definitions, and of a list of concrete productions and associated tree construction "actions".

Since ATC can manipulate only values of types *token* and *source-index*, the output AAS may have attributes of these types ONLY. This is checked by ATC. Furthermore, attributes of type *source-index* implicitly include the predefined attribute **$scx**, attached to every phylum. Lastly, this AAS must actually be reduced, i.e. it must not contain still unchecked empty phyla (see chapter 12 of *the* OLGA *Reference Manual*).

Syntax:

```
<concrete grammar>    =  <header>
                         <definition*>
                         <production group+>;

<header>              =  import grammar %ID ";";

<definition*>         =  <definition*>
                         <definition>;
<definition*>         =  ;
<definition>          =  <phylum def>;
<definition>          =  <phylum extension>;
<definition>          =  <operator def>;
<definition>          =  <function def>;
```

22.3 Lexical Considerations

The productions *must* begin in column one. Any "less than" sign (<) in column one is interpreted by *bnf* as the beginning of a production; this means that non-terminals in right-hand sides of productions and in tree specifications must not appear in column one. Non-terminals appear in productions and tree specifications as a sequence of characters enclosed in angle brackets (<>); any printable character is allowed, except the right angle bracket and "line breakers" such as newline, form-feed, etc. Inside productions, non-generic terminals may appear as they must appear in source texts, or with a leading sharp sign (#) or enclosed in double quotes (see *the SYNTAX Reference Manual*). However in tree specifications only the latter form is allowed; note that they are then lexically indistinguishable from string constants; the distinction is a syntactic and semantic matter. Generic terminals appear in productions and tree specifications as C identifiers—formed with letters, digits and underscores, the first character being a letter—with a leading percent sign (%). In tree specifications, operators, keywords and pattern variables are OLGA identifiers (see chapter 6 of *the* OLGA *Reference Manual*, in this volume). In declarations preceding the first production, phyla and operators and functions are also OLGA identifiers. Integral numbers and string constants are also defined as in OLGA. Comments out of productions are OLGA comments; comments inside productions are white space and pretty-printing specifications (see the chapter on *paradis* in [BDL87]); meaningful comments of *bnf*—a star sign in column one up to the end of the line—are also allowed anywhere in ATC. Here comes the list of keywords which are reserved identifiers:

$scx **and case else elsif end grammar if import in map match message position severity then tree value with**

22.4 Tree Types and Construction Functions

As in Olga tree construction operations (see chapter 14 of *the* Olga *Reference Manual*), phyla are considered as (tree) types and operators as functions constructing values of these types. Let us recall here the definitions of these functions.

A fixed-arity operator `op` belonging to phylum `PH` and declared as

```
op -> PH1 PH2 ...  PHn ;
```

corresponds to a tree construction function named `op` with profile

function `op (PH1; PH2; ...; PHn):  PH ;`

A list operator `op` belonging to phylum `PH` and declared as

```
op -> PH1* ; or op -> PH1+ ;
```

corresponds to several tree construction functions defined as follows:

function `op ():  PH`
function `op (PH1):  PH`

are the "initialization" functions which construct a list with zero and one offspring respectively. The former is not defined for never empty list operators.

function `op-post (PH; PH1):  PH`
function `op-pre (PH1; PH): PH`

allow to add an offspring (the `PH1` parameter) to an already existing list node headed by the operator `op` (the `PH` parameter), respectively at the end and at the beginning of that list. It is an error if the `PH` parameter is not a list node headed by `op`; in this case, an error message is emitted and the nullary tree `null-PH ()` is returned; however the message is not emitted if the offending tree was already headed by `null-PH`.

function `op-merge (`**list of** `PH1;` **list of** `PH1):  PH`

allows to merge the offsprings of two list nodes into a single node headed by `op` whose offsprings are those of the first parameter followed by those of the second. What is denoted by **list of** `PH1` in the "profile" above is actually a very special, anonymous type, such that the only way to acquire a value of that type (a *forest* of trees) is to destructure a list node whose offsprings are of type `PH1` (see section 22.10 below).

function `op-all (`**list of** `PH1):  PH` creates a list node headed by `op` whose offsprings are those of another destructured list node, provided that the phylum of these sons is actually `PH1`. This allows to rename an existing list node.

It is an error if, in the two latter functions, respectively both and the unique parameter are empty lists, and `op` is a never-empty list operator; in this case a (non-customizable) error message is emitted and the result is the nullary tree headed by operator `null-PH`. Since that error message, and also the one emitted if `op-pre` or `op-post` does not receive a good parameter, will mean nothing to the end-user, it is recommended that the author of an ATC specification ensures that these situations never occur.

22.5 Generalities on Tree Construction Actions

Each non-terminal carries as an anonymous; synthesized attribute a well-formed tree whose root operator belongs to a unique phylum this association between non-terminals and phyla is a particular case of type inference and default attribute declaration, as defined in OLGA. The phylum associated to the start symbol of the concrete grammar[1] must be the root phylum of the output AAS.

Each constructed tree node has attributes, of type *token* only, as specified in the output AAS; these attributes must be explicitly computed and attached to the newly constructed nodes. Each node also has the predefined attribute **`$scx`** whose default value is the position in the source text of the first character of the string deriving from the corresponding symbol; for non-terminals deriving the empty string, this position is somewhere between the last token deriving from its left sibling and the first token deriving from its right sibling. This attribute can be redefined as explained below. Other attributes of type *source-index* must be defined explicitly as those of type *token*. A value of type *source-index* is the position of some symbol in a production.

Strings can be constructed by concatenating string constants and texts of generic terminals, and assigned as values of type *token* to suitable attributes of the constructed trees, or used as error messages.

22.6 Declarations

After the header, and before the first production, may appear a list of definitions of new phyla and/or operators which will temporarily extend the basic abstract syntax during tree construction. New phyla and operators are defined as in AASes (see *the* OLGA *Reference Manual*, chapter 12). It is also possible to add new operators to existing phyla using the `+=` construct.

Syntax:

```
<phylum def>          =  %ID "=" <operator +> ";";
<phylum extension>    =  %ID "+=" <operator +> ";";
<operator +>          =  <operator +> <operator>;
<operator +>          =  <operator>;
<operator>            =  %ID;
```

[1] which is the left-hand side non-terminal of the first production in textual order, as defined by SYNTAX; note that this non-terminal must not appear in the right-hand side of any production.

```
<operator def>        =  %ID " -> " <phylum> "+" ";";
<operator def>        =  %ID " -> " <phylum> "*" ";";
<operator def>        =  %ID " -> " <phyla*> ";";
<phyla*>              =  ;
<phyla*>              =  <phyla+>;
<phyla+>              =  <phyla+> <phylum>;
<phyla+>              =  <phylum>;
<phylum>              =  %ID;
```

It must be ensured that the tree nodes constructed with these new phyla and operators actually disappear in subsequent actions, for instance by destructuring them using pattern matching primitives (see further). The tree constructor generator will reject every specification for which this property cannot be statically verified; however the actual algorithm used for this verification is still unspecified (and not implemented).

It is also possible to define non-basic tree construction functions, as indicated in chapter 14 of *the* OLGA *Reference Manual*, but they may not be recursive.

Syntax:

```
<function def>        =  function %ID "(" <formals *> ")"
                                   ":" <phylum> is
                            <expression>
                         end function ";";
<formals *>           =  ;
<formals *>           =  <formals +>;
<formals +>           =  <formals +> ";" <param group>;
<formals +>           =  <param group>;
<param group>         =  <id +> ":" <phylum>;
<id +>                =  <id +> "," %ID;
<id +>                =  %ID ;
```

After being defined, user-defined tree construction functions are indistinguishable from functions corresponding to operators, and they belong to the same name space (which means that there may be name conflicts). However, they can not be used as pattern constructors (see below).

22.7 Tree Construction Actions Associated to Individual Productions

Syntax:

```
<production group+>   =  <production group+>
                         <production group>;
<production group+>   =  <production group>;
<production group>    =  <rule>
                         <tree spec>;
```

```
* The following rules are borrowed from SYNTAX (bnf.bnf)
<rule>              =  <vocabular_list> ";";
<vocabular_list>    =  <vocabular_list> <vocabulary>;
<vocabular_list>    =  %LHS_Non_Terminal "=";
<vocabulary>        =  %Non_Terminal ;
<vocabulary>        =  %Terminal ;
<vocabulary>        =  %Generic_Terminal ;
* End of rules borrowed from SYNTAX

<tree spec>         =  ;
<tree spec>         =  <operator>;
<tree spec>         =  <expression>;
```

An empty tree specification is only valid for *simple productions* of the form `<A> = <B>` where `<A>` and `<B>` are associated to the same phylum. In that case this means that the tree to be attached to `<A>` is the one that is attached to `<B>`.

A tree specification associated to some production

```
<NT0> = "T1" %GT1 <NT1> ...  "Tn" %GTn <NTn>;
      op
```

where the Ti's are non-generic terminals, the `%GTi`'s are generic terminals, and the `<NTi>`'s are non-terminals, and reduced to a single operator name `op` is an abbreviation standing for

```
op (<NT1>, ..., <NTn>)
```

that is the tree with root operator `op` and offsprings the trees associated to all the non-terminals in the right-hand side of the production, in the same order. This abbreviation is subject to type checking as its expanded equivalent; see sections 22.9 and 22.16 below.

The other forms of tree specification are *tree expressions* as defined below. When the parser performs a reduction by some production, the (implicit or explicit) tree expression associated to the production is evaluated, and the result is attached to the left-hand side non-terminal as its unique, anonymous, synthesized attribute. The trees attached to the right-hand side non-terminals are then discarded.

22.8 Basic Operands

Syntax:

```
<expression>        =  <primary expr>;
<primary expr>      =  %ID ;
<primary expr>      =  <symbol nt>;
<symbol nt>         =  <non-terminal>;
<symbol nt>         =  <non-terminal> "." %INT ;
<non-terminal>      =  %Non_Terminal ;
```

The basic operands of tree expressions *(primary expressions)* are the trees attached to the non-terminals in the right-hand side of the production. These values are denoted by the non-terminals themselves, with possible integer qualifiersfor disambiguation as in fixed arity productions of OLGA (chapter 13); however only the right-hand side non-terminals may be thus designated—accessing the LHS non-terminal has no meaning anyway—, which means that `<NT>.1` or `<NT>` designates the first occurrence of non-terminal `<NT>` in the RHS of the production, even if `<NT>` is also the LHS non-terminal. Each of these trees has a type which is the phylum of its root operator.

Several constructs described below also define *tree variables* obtained by destructuring an existing tree, or as formal parameters in function bodies. These variables, denoted by identifiers, can then be used as operands of tree construction functions where they are visible. No tree variable is visible at the outermost level of a tree specification associated to a production. Only the function formal parameters are visible at the outermost level of a function body.

When a tree variable is visible, its attributes can be referred in expressions for **message** constructs (section 22.13 below) and in decoration clauses (section 22.15).

22.9 Tree Construction Expressions

Syntax:

```
<expression>        =  <tree>;
<tree>              =  <operator> "(" <operand*> ")";
<operand*>          =  ;
<operand*>          =  <expression+>;
<expression+>       =  <expression+> "," <expression>;
<expression+>       =  <expression>;
```

A *tree construction expression* is the application of (one of) the tree construction function(s) associated with some operator, or some user-defined function, to some operands. These operands must conform in number and types with the profile of the function as defined in chapter 14 of *the* OLGA *Reference Manual* and above.

22.10 General Pattern Matching Expressions

A primary expression (a tree constructed else where) can be matched against one (**match ... in**) or more (**case ... match**) *patterns* and its components bound to tree variables. Those variables are then visible in subexpressions and can either be used to construct new trees or be themselves matched against other patterns.

The **case ... match** construct is very similar to the corresponding OLGA construct (chapters 9 and 14).

Syntax:

```
<expression>        =  <case match>;
<case match>        =  case <primary expr> match
                          <match clause+>
                       end case ;
```

```
<match clause+>        =  <match clause+>
                          <match clause>;
<match clause+>        =  <match clause>;
<match clause>         =  <pattern> ":" <expression> ";";
<pattern>              =  <operator> "(" <patterns *> ")";
<pattern>              =  %ID ;
<pattern>              =  %ID ":" %ID ;
<pattern>              =  other ;
<patterns *>           =  ;
<patterns *>           =  <patterns +>;
<patterns +>           =  <patterns +> "," <pattern>;
<patterns +>           =  <pattern>;
```

The expressions in the right-hand sides of the match-clauses and the match-default must all belong to the same type, which may be any tree type; that common type will be that of the whole match-expression. The selector-expression must be a primary tree expression.

The patterns in the left-hand sides of the match-clauses are terms constructed with named variables (which are represented by identifiers), anonymous variables, and tree construction operators;[2] the arguments of a pattern constructor must respect the profile of the corresponding operator, which means that there must exist the same number of arguments and that these arguments, if not variables, must be patterns which might occur in another **case ... match** expression for the argument type. The match-default clause must be present; the pattern in its left-hand side must be an anonymous variable. A new scope is opened for the right-hand side of each clause, except the default clause, in which the (named) variables occurring in the corresponding pattern are visible; their type is that of the formal parameter of a construction function to which they correspond. Such scopes nest as usual, with possible hiding of some outer identifiers; however operator names are never hidden because it is syntactically possible to distinguish them from variable identifiers.[3] If a given variable occurs more than once in a pattern, then the type determined for each of its occurrences must be the same; this also means that a value matches such a pattern only if all the occurrences of that variable have the same actual value (see further). However the anonymous variable is special in that it is not visible in the right-hand side of the clauses, and that if it appears more than once in some pattern, the corresponding tree instances will not be checked for equality.[4]

The set of patterns must be such that if some value matches several ones, then one of them must be more specific than all the others: a pattern A is more specific than a pattern B iff the set of values which match A is strictly included in the set of values which match B; note that this may be the effect of the repetition of some variable.

The dynamic semantics of a match-expression is as follows: the selector expression is evaluated. Then it is matched against all the patterns in parallel; if a variable occurs more than once in a pattern, then the corresponding values are tested for equality (using the corresponding = function, see below) to determine whether the matching succeeds. Then the most specific pattern among those which match is selected; note that the matching always

[2] However user-defined functions cannot be used as pattern constructors.

[3] which means that, as opposed to OLGA, functions and operators names on one hand, and variable names on the other hand, do not belong to the same name space.

[4] one could imagine that each occurrence of **other** denotes a different anonymous variable.

succeeds because any value matches the anonymous variable in the left-hand side of the match-default clause. That most specific pattern is always uniquely determined due to the above restriction. The variables in the pattern are bound to the corresponding values in the actual value of the selector. A new environment is set up which records these bindings and the expression in the right-hand side of the selected clause is evaluated in that environment. The value of this expression becomes the value of the whole match-expression.

A fixed-arity node is matched by a pattern whose constructor is the corresponding construction function defined above.

The only patterns which could match a list node headed by operator `op` belonging to phylum `PH` and whose offsprings belong to phylum $\mathtt{PH}_1$ are as follows:

- `op()`
  matches only an empty list; does not apply if `op` is a never-empty list operator.

- `op(P)`
  where `P` is a subpattern which can match a tree belonging to phylum $\mathtt{PH}_1$, matches if the `op` node is a one-element list whose element matches `P`.

- `op-post(`$\mathtt{P}_1$`, `$\mathtt{P}_2$`)`
  where $\mathtt{P}_1$ is a subpattern which can match a tree headed by operator `op` and $\mathtt{P}_2$ is a subpattern which can match a tree belonging to phylum $\mathtt{PH}_1$, matches if the `op` node is not empty and its last (rightmost) son matches $\mathtt{P}_2$ and the list of all sons but the last, when considered as an `op`-node, matches $\mathtt{P}_1$.

- `op-pre (`$\mathtt{P}_2$`, `$\mathtt{P}_1$`)`
  same as `op-post` with $\mathtt{P}_2$ matching the first (leftmost) son and $\mathtt{P}_1$ the list of all sons but the first.

- `op-all (X)`
  where `X` **MUST** be an **UNTYPED VARIABLE**—this is because no pattern can match a forest except the universal pattern—matches in any case; `X` will be bound to a forest containing all the offsprings of the matched node, whatever their number.

`op-merge (X, Y)` is not a possible pattern because there would be no way to draw the limit between the sublists `X` and `Y`.

The universal pattern, denoted either by **other** or by a single pattern variable (possibly qualified by a type name), is also available as a top-level pattern; in the latter case this variable is also visible in the RHS of the clause.

Finer patterns may be constructed by combining these constructors; for instance, in the pattern `op-pre (X, op-pre (Y, Z))`, `X` would correspond to the first offspring, `Y` to the second and `Z` to a list node containing all the other offsprings, provided there are at least two of them (three if `op` is a never-empty list operator).

If a phylum $\mathtt{PH}_1$ is included in a phylum $\mathtt{PH}_2$, and if a variable appearing in some pattern is determined to be of type $\mathtt{PH}_2$, then, as a special case, it will match any tree of type $\mathtt{PH}_1$ or $\mathtt{PH}_2$ and will be visible in the RHS of the clause as having type $\mathtt{PH}_2$; to have it match only trees of type $\mathtt{PH}_1$ and be visible with type $\mathtt{PH}_1$, use an explicit type constraint.

When some pattern variable appears more than once in some pattern, equality of the corresponding subtrees must be tested. Two subtrees are equal (in the sense of operator `=`)

if and only if they have the same syntactic structure and are decorated by the same (in the sense of usual equality) attributes, not taking into account the **$scx** attributes.

If some variable corresponding to some output tree node is used more than once in the final value of the right-hand side expression of a match clause, the corresponding tree is *duplicated* for each of these instances, so that the output tree actually remains a tree and does not become a graph. More generally, if any tree is used more than once to construct a larger tree, it is duplicated, along with its attributes.

22.11 Unconditional Pattern Matching Expressions

The *unconditional match* construct provides a useful abbreviation to the general matching construct when some primary expression is statically known to have a given shape; this is the case for instance when this expression is some non-terminal such that the tree specifications associated to all the productions defining it produce trees headed by the same operator; a particular case is when this non-terminal is defined by only one production.

Syntax:

```
<expression>          =  <unconditional match>;
<unconditional match> =  match <match-in clause+>
                         in <expression>;
<match-in clause+>    =  <match-in clause+> ","
                         <match-in clause>;
<match-in clause+>    =  <match-in clause>;
<match-in clause>     =  <primary expr> with <pattern>;
```

The expression

```
match prim-expr-1 with patt-1,
   prim-expr-2 with patt-2
in expr
```

is equivalent in every respect to

```
case prim-expr-1 match
   patt-1:  case prim-expr-2 match
               patt-2 :  expr;
               other :  null-PH ();
         end case ;
   other:  null-PH ();
end case
```

where PH is the phylum (type) of expr; the extension to more than two match clauses is obvious. Note that even if the shape of some primary expression is statically known, the match may fail due to the effect of global error recovery (see further). No error message is emitted if the match fails.

22.12 Conditional Expressions and Relational Expressions

Syntax:

```
<expression>          =  <conditional>;
<conditional>         =  if <relational expr> then
                            <expression>
                         <elsif part>
                         else
                            <expression>
                         end if ;
<elsif part>          =  <elsif part>
                         elsif <relational expr> then
                            <expression>;
<elsif part>          =  ;
<relational expr>     =  <expression> "=" <expression>;
<relational expr>     =  <expression> "!=" <expression>;
<relational expr>     =  <primary expr> match <pattern>;
<relational expr>     =  <relational expr> and <relational expr>;
```

The conditional expressions involving only comparisons (using `=` and `!=`) are defined as in Olga. The equality of two trees is defined in previous section. The non-strict **and** conjunction operator is defined as in Olga.

A **match** relational expression is true iff the left-hand side expression matches the right-hand side pattern, but it has the additional side-effect to bind the pattern variables to the corresponding tree instances and to make them visible in the expression following the **then**. This is thus a special case of the general matching primitive. If a **match** relational expression is the left-hand side of an **and** conjunction, the results of the matching are also visible in the right-hand side. It is possible to mix matchings and comparisons in a same conjunction, with obvious semantics.

22.13 Error Messages

Syntax:

```
<expression>          =  <message>;
<message>             =  message <position or string>
                            position <position or string>
                            severity %INT
                            value <expression>;
<position or string>  =  %ID ;
<position or string>  =  <primary symbol>;
<position or string>  =  <primary symbol> "." %INT ;
<position or string>  =  <position or string> "+"
                         <position or string>;
<position or string>  =  %ATTR "(" %ID ")";
```

```
<primary symbol>      =  <non-terminal>;
<primary symbol>      =  %Generic_Terminal ;
<primary symbol>      =  %STRING ;
```

The **message** construct is the same as in OLGA with the following differences and restrictions:

- the string expression following the **message** keyword is the concatenation of string constants, possibly qualified generic terminal—the actual text of the corresponding instance will be substituted—and attributes (of type *token* only) of tree variables when they are visible (the *token* value will be converted to a string);
- the position of the message must uniquely identify one of the symbols of the production: LHS and RHS non-terminals, generic terminals and non-generic terminals (enclosed in quotes, which makes them look like string constants); integer qualifiers can be used for disambiguation; alternatively, it can be a tree variable, in which case the value of the **$scx** attribute of its root node is used as the position of the message; it is also possible to use the value of an explicit attribute of a tree variable, provided it is of type *source-index*;
- the severity level must be an integer constant.

22.14 Abbreviations

Syntax:

```
<expression>          =  <abbreviation>;
<abbreviation>        =  let <let clause+>
                         in <expression>;
<let clause+>         =  <let clause+> ","
                         <let clause>;
<let clause+>         =  <let clause>;
<let clause>          =  %ID ":=" <expression>;
```

Abbreviations to name complex subexpressions are introduced by the **let** construct which is defined as in OLGA, except that it is valid only for tree expressions (but not for e.g. relational expressions), and that no type constraint is necessary.

22.15 Tree Decoration

Syntax:

```
<expression>          =  <tree with>;
<tree with>           =  <operator> "(" <operand*> ")" <with>;
<tree with>           =  <primary expr> <with>;
<with>                =  with <attribute +>
```

```
                              end with ;
<attribute +>          =      <attribute +> ","
                              <attribute>;
<attribute +>          =      <attribute>;
<attribute>            =      <attr name> ":=" <position or string>;
<attr name>            =      %ATTR ;
<attr name>            =      $scx ;
```

Each newly constructed tree must be decorated by its attached attributes, if any. It is also allowed to change the attributes of an existing node. This is achieved by decoration clauses similar to those described in section 14.3 of *the* OLGA *Reference Manual.*

With ATC, attributes are of two kinds:

- explicit ones, which may be of type *token* or *source-index*; these attributes must be explicitly defined for nodes to which they are attached; the values of the former are defined by strings as described above for the **message** construct, transformed into a value of type *token* by a conversion function as in OLGA (chapter 7); the values of the latter are defined as the position of a message, i.e. as the source index of some symbol appearing in the production or as that of some visible tree node;
- the predefined attribute **$scx**, which is implicitly attached to every node; this attribute may be defined explicitly or receives a default value; in the latter case, and if it is the only attribute of some node, the decoration clause is omitted.

Decoration clauses are *forbidden* in user-defined functions, which means that nodes constructed by a user-defined function must have no attribute[5] except possibly the root node which must be decorated outside the function. Newly constructed nodes—be it with an operator function or a user-defined function—must be decorated in the context of the production.

If **$scx** is explicitly defined for some node, its value must be defined as the position of some symbol or tree node. Otherwise its default value is as follows:

1. for the node corresponding to the LHS non-terminal, i.e. for the root of the tree constructed as the result of the top-level tree expression associated to the production, the source index is the first character of the terminal string derived from that non-terminal or, if the production has an empty RHS, somewhere between the last token deriving from its left sibling and the first token deriving from its right sibling;
2. for other non-nullary nodes, it is the source index of their leftmost offspring;
3. for nullary nodes having a unique attribute of type *token* which is explicitly defined as the text of a unique generic terminal, the default value is the position of the first character of this terminal;
4. for other nullary nodes, it is the source index of its right sibling, provided that this sibling can be statically identified in the production or in the function body; otherwise the default value is undefined and must be provided explicitly;

[5] apart from **$scx** which will automatically receive its default value.

5. for nullary nodes of the form `null-PH` constructed automatically in some error situations described above or during syntactic error recovery (see below), there is a default value which is otherwise unspecified.

The second and fourth cases above are the only situations applicable to interior nodes constructed in user-defined functions.

Some or all of the attributes of existing nodes, accessible by a primary expression, may be changed by specifying new decoration clauses. The rules are the same as for newly constructed nodes, except that it is not mandatory to give a definition for every explicit attribute; the old value is assumed instead.

22.16 Miscellaneous

Type checking of all the above constructs is defined as in OLGA, except it is much simpler since there is no polymorphism and no overloading (at least, they can be treated in an *ad hoc* way). There is also the same "overloading" provision for phylum inclusion as in OLGA (see chapter 14 of the OLGA *Reference Manual*).

The tree constructor generator will try to attach to each non-terminal the exact list of root operators which can label the trees it carries; this is finer than just the corresponding phylum and will help making stronger verifications, especially regarding pattern matching primitives.

From a syntactic point of view, the different constructs have the following associativity rules and relative priority:

- expressions with highest priority are basic operands (non-terminals and variables), function calls, and closed constructs, namely **case ... match**, **if ... then**;
- the **with** decoration construct is not associative;
- relational and string expressions come next; the `=` and `!=` operators are not associative, whereas the `+` and **and** operators are left-associative;
- the open constructs **match ... in**, **message** and **let ... in** all associate to the right and have the same lowest priority.

Use the **let ... in** construct to turn a low priority expression into a higher priority one.

22.17 Syntactic Error Recovery

The normal operation of the tree constructor is as follows: the source text is parsed bottom-up, and a stack containing partial trees is maintained in parallel with the parse stack. Upon reduction by a given production, the trees attached to non-terminals in the right-hand side are retrieved from the stack, and then the tree specification is evaluated. The tree stack is popped by the length of the production and the new tree is pushed upon it, i.e. it is attached to the left-hand side non-terminal. This mechanism is slightly altered when a syntactic error is encountered.

SYNTAX uses a powerful error repair and recovery scheme involving two phases. The first one tries to *locally correct* the erroneous text by performing deletions, insertions and replacements of tokens. This correction is driven by a number of user-defined *patterns* (see *the SYNTAX Reference Manual*). If this local correction succeeds, the new source text is syntactically correct and the tree construction actions can proceed as usual. The only difference from an initially correct text occurs when a generic terminal is inserted or replaces an original token: the parser cannot choose a semantically meaningful text for this new token. Instead the *token* value of this inserted terminal will be the predefined value *error-token* (see *the* OLGA *Reference Manual*, section 7.3.6). Semantic rules must be prepared to deal with such values; see the example of *simproc* in appendix A. Any string expression in which such a terminal appears will have the following value:

- if it is to be assigned to some attribute, the value of the whole expression will also be *error-token* (but see special case below);
- if it is used as an error message, the value of the string is as if that terminal had an empty text.

A special case of the first situation is when some nullary tree has a unique attribute of type *token*, defined as (the text of) a unique generic terminal. If so, and when this generic terminal has no meaningful text due to a (repaired) syntactic error, the corresponding `null-PH ()` tree will be constructed instead, with *error-token* as the value of its attribute. There are thus two means to recognize such "erroneous" terminals, either by checking its root operator or by checking the value of its attribute.

If unhappily local correction fails, the second *global recovery* phase is entered. The source text is scanned without parsing until a *key terminal* is encountered; the parse stack is then searched until a state with a valid transition on this key terminal is encountered; the stack is then popped, the transition is performed, the terminal is shifted and the parser is restarted; since this state corresponds to some non-terminal, this is as if the portion of text which is derived from "popped non-terminals" and the one which is skipped were together derived from this non-terminal. If no such state is found, the text is scanned up to the next key terminal and the process is repeated. This will eventually terminate since "end of file" is always a key terminal; the corresponding state is (at worse) the initial state and the corresponding non-terminal is the start symbol of the grammar.

The net effect of this scheme is twofold: a whole portion of text is skipped without analysis, and some already constructed trees are popped and forgotten. In that case, the tree constructor will actually forget the already constructed trees corresponding to popped entries—their validity is indeed discussable since they are part of a portion of text which is globally erroneous—and push in their place a tree constructed with the nullary operator `null-PH` if `PH` is the phylum associated to the "recovery" non-terminal. This explains why semantic rules must be provided in each attribute grammar for those nullary operators. This also explains why unconditional matches (section 22.10 above) may fail, even if the shape of correct trees is statically known; note that the default rule in this case is that erroneous trees propagate through unconditional match, but they are caught in other tree construction expressions. Attributes of type *token* attached to such erroneous trees will all receive the value *error-token*; the value of the **`$scx`**; attribute will be the beginning of the skipped portion of text, whereas the value of other attributes of type *source-index* will be the end of the skipped portion of text.

Chapter 23

Constructing a Tree Constructor

This chapter presents how to operate Atc and the SYNTAX system to generate, from several description files, a scanner/parser/tree-constructor. Apart from *bnf*, the description of the operation of SYNTAX is rather sketchy refer to [BDL87] for more details. See also chapter 21 of this Part.

23.1 The Grammar Reader and the Tree Constructor Generator

As stated in chapter 21, the grammar reader *bnf* and the tree constructor generator Atc take their input from the same description file and process it simultaneously in a single run.

23.1.1 Operation

When you type the following command:

```
tutorial% atc simproc.atc
```

bnf and Atc analyze the specified source file and, with default options, output the following on the terminal:

```
Release 1.8 of tomorow
simproc.atc:
   Scanner & Parser & Abstract Tree
   Symbol Table Processing

<DECL> = proc <id> "(" val <id> "," var <id> ")"

simproc.atc, line 18:  column 1:  Warning:  The right hand-side of production
#7 seems too long.  Check for a missing semi-colon.
   Property Test
   Follow Construction
   Tree Construction Actions Processing
      Semantics:  *SSSSSSSS
      Translator:rrrrrrrrrrrrrrrrrrrrrr
```

```
Listing Output
Tree Construction Listing Output
```

The first line identifies the version of ATC you are using. Then, for each source file—*atc/bnf* can compile several files in a single run—, its name is echoed, and the work is divided into several phases:

- *Scanner & Parser & Abstract Tree*
  performs lexical and syntactic analysis of the source text, with *bnf* and ATC alternatively taking control on their respective inputs; lexical and syntactic errors are detected and give rise to messages, but analysis proceeds. Note that pretty-printing directives for Paradis (see [BDL87] for more details on this pretty-printer generator) may be embedded into the productions; they will be ignored by both *bnf* and ATC, which means that the same file may also be used to specify a pretty-printer for the input language. Syntactic actions and predicates numbers may also be embedded in the productions; they will be processed by *bnf*.

- *Symbol Table Processing*
  bnf now takes control. This phase constructs an internal representation of the concrete grammar and performs a few consistency checks. The warning message above exemplifies one of these tests: since it is very easy to forget the semicolon terminating a production, a warning message is emitted each time the right-hand side of a production contains more than a certain number of symbols, specified with the −**rhs** option (see below), which happened in this run. You can admire how clearly SYNTAX displays error messages. The only other test performed in this phase is to check that the start symbol of the grammar, defined as the left-hand side non-terminal of the first production in textual order, does not appear in the right-hand side of any other production; if not so, processing is terminated at the end of the phase.

- *Property Test*
  checks that the concrete grammar is reduced as defined in chapter 21. Easy sufficient ambiguity conditions—e.g. a non-terminal deriving into itself through a chain of simple productions, or the equality of two productions[1]—are also checked in this phase. If the grammar fails any of the tests, its processing is terminated.

- *Follow Construction*
  constructs the relations $First_1$ for each non-terminal and *Follow* for each terminal of the grammar; the latter is used by the lexical constructor `lecl` to determine the possible contexts.

- *Tree Construction Actions Processing*
  ATC now takes control. If `my-grammar` is the output AAS of the tree constructor, as specified in the header of the specification, the AAS tables in file `my-grammar.ast` are read in. This file must pre-exist in one of the directories specified in the −**d** option(s) (see below), otherwise a fatal error is signaled and the processing of the current text is

[1]Pedantics consider that, in this case, the grammar is not a grammar, because the set of productions is not a set any more. The authors prefer to consider this as an ambiguity problem.

immediately terminated. It is also a fatal error if this grammar is incomplete (i.e. not reduced) in the sense of section 22.2 of previous chapter.

- *Semantics*
  checks typing and static semantic constraints on actions associated with productions, and attaches to each non-terminal the phylum of the corresponding tree. The number of `S`'s reflect the number of passes over the grammar. It is also checked that temporary operators and phyla disappear from complete trees[2]. Each check may detect errors. If any severe error is detected here or in a previous phase, processing terminates at the end of the phase.

- *Translator*
  generates the C code corresponding to the specification.

- *Tables Output*
  `bnf` takes control back and produces a file containing an unreadable set of tables. These tables will be used by the other modules of SYNTAX (see chapter 21 and below).

- *Listing Output* and *Tree Construction Listing Output*
  produce the listing of the compilation in two phases, one for `bnf` and the other for ATC. Its contents is detailed below.

23.1.2 Contents of the Listing

23.1.2.1 Header

The header of the listing specifies:

- the base-name of the source file;
- the name (ATC), purpose and version number of the processor;
- the processing options used in this run, stripped from their leading "–";
- the date and time of the compilation.

23.1.2.2 Source text and error messages

Then comes the source text. Each line is numbered in the left margin. Each production is also numbered in the left margin, on the line where it begins. Error messages are embedded in this listing, together with a marker indicating the exact place where the error occurred. The messages themselves should be self-explanatory. This listing is followed by a summary of the number of warnings and errors reported.

23.1.2.3 Reduceness information

If the concrete grammar proves to be not reduced, this is indicated by two kinds of messages:

- *The following non-terminals are unreachable from the axiom:* `<XXX> <YYY> ...`

[2]This is presently not implemented at generation time but at run time after each tree construction.

- states that it is not possible to derive non-terminal `<XXX>` from the start symbol, i.e. it is not possible to find a correct tree with root the start symbol containing an instance of `<XXX>`.
- *The following non-terminals cannot derive a terminal tree:* `<XXX> <YYY> ...`
- states that it is not possible to find a correct tree with root `<XXX>` whose leaves are all labeled with terminals (including the empty string).

23.1.2.4 Non-terminals Listing

Non-terminals are listed in alphabetic order. Each comes with its internal code (of no practical use for the casual user), its name, and the numbers of the productions in which it appears, either as the left-hand side or in the right-hand side.

23.1.2.5 Terminals Listing

Terminals are listed in alphabetic order. Each comes with its internal code, its name, and the numbers of the productions in which it appears in the right-hand side.

23.1.2.6 Context Table

The context table is a square matrix, indexed by the terminals of the grammar. An "X" at the intersection of the row corresponding to terminal A and the column corresponding to terminal B states that B can follow A. This table may extend over several pages.

23.1.2.7 Non-terminals Typing

The last cross-reference table indicates, for each non-terminal, the name of the root phylum of the trees it carries. This information is determined by Pass 4 of Atc processing.

23.1.3 Command line and options

The basic syntax to invoke Atc is as follows:

```
atc [ options... ] [ input_files... ]
```

If no `input_files` argument is present, the standard input is read. Otherwise, each `input_file` is analyzed.

Options may appear in any order as long as they appear before the `input_files`. Only the last occurrence of a given option is taken into account, except for the **–d** option. Available options are as follows:

- **–v**, **–verbose**
  Animate the user's screen by displaying information about what happens. (Default)
- **–nv**, **–noverbose**
  Execute silently.

- **–d** `path`, **–directory** `path`
  Add `path` to the list of directories to search for the tables describing the output AAS. The order of these options is significant. (Default: **–d** . only)

- **–sc**, **–source**
  Produce a short listing, including the items described in sections 23.1.2.1 to 23.1.2.3 above. (Default)

- **–nsc**, **–nosource**
  Do not produce any listing; implies **–nls**.

- **–ls**, **–list**
  Produce a long listing, including the items described in sections 23.1.2.1 to 23.1.2.7 above; implies **–sc**. (Default)

- **–nls**, **–nolist**
  Do not produce in the listing, if any, the items described in sections 23.1.2.4 to 23.1.2.7 above.

- **–rhs** `N`, **–max_right_hand_side** `N`
  Produce a warning message when the right-hand side of a production contains more than `N` symbols. (Default: **–rhs** 10)

Default options are such that the command

```
atc
```

is equivalent to

```
atc -v -sc -ls -d .  --rhs 10
```

Note that the **–language_name** (**–ln**) option of `bnf` is not available with ATC. This is because the "name of the language" is derived from either the name or the contents of the input file (the latter occurs when standard input is read), so that it is not necessary to specify it by external means.

23.1.4 Files

If the base name of the input file is `concrete.atc` and the name of the output AAS, as specified in that input file, is `abstract`—if standard input is read, it is as if the input file was called `abstract.atc`—, the following files are read or created:

- `concrete.atc`
  is the input file containing the concrete grammar and the description of tree construction actions. It must be accessible for reading and may be located in any directory, provided you specify its path in the command line.

- `abstract.ast`
  is the file containing the internal representation (tables) for AAS `abstract`. This file **MUST** have been generated by a previous run of ASX (see chapter 17), be accessible for reading and located in one of the directories specified with the **–d** option.

- `concrete.bn.l`
  is the file containing the listing, as described above. This file is created in the working directory.

- `concrete.bt`
  is the file containing the internal representation (tables) of the concrete grammar, together with information regarding the connection with the tree constructor (namely action numbers). This file is created in the working directory, and will be used by other components of SYNTAX to generate the parser (see below).

- `abstract_atc.c`
  is the file containing C code and data implementing the tree construction actions. This file is created in the working directory. Its use is explained below.

23.2 Other Processors of SYNTAX

This section briefly describes the other operations you have to do to generate the front-end of your system. Refer to *the SYNTAX Reference Manual* [BDL87] for more details.

23.2.1 Input files

To generate a complete analyzer for your input language, you will have to prepare the following files in addition to `concrete.atc`:

- `concrete.lecl`
  is the specification of the scanner for your language. Tokens are specified as regular expressions over a set of characters. You may also specify actions and predicates for context-sensitive processing.

- `concrete.recor` is the specification of how to display, correct and recover from errors.

- `concrete.prio` is the (optional) specification of disambiguation rules to be used by the LALR(1) parser generator to solve conflicts.

- `concrete.tdef` is an optional file specifying C names for terminals, to be used if necessary in the following files.

- `concrete_sact.c` is an optional file containing C code for scanning time actions and predicates.

- `concrete_pact.c` is an optional file containing C code for parsing time actions and predicates.

- `concrete_main.c` is C code for the top level of your language processor (the "main" procedure). It will have to call the scanner/parser/tree-constructor and other code generated by FNC-2 from your other OLGA specifications. An example of such a procedure is available in file `simproc_main.c` in directory `examples` in your installation of FNC-2 .

23.2.2 Construction of the compound tables

You will then have to operate the following processors:

- *lecl* is the scanner generator. It reads the lexical description from `concrete.lecl` and the grammar tables from `concrete.bt` and produces a deterministic finite state automaton described by a set of tables into `concrete.st`. `lecl` can also produce the scanner as a C program instead of a set of tables, but this will not be developed here.

- *recor* is the error processing constructor. It reads the lexical tables from `concrete.st` and the grammar tables from `concrete.bt` and produces a set of tables into `concrete.rt`.

- *prio* is the disambiguation processor. It reads the disambiguation description from `concrete.prio` and the grammar tables from `concrete.bt` and produces a set of tables into `concrete.dt`, to be used by `csynt`.

- `csynt` is the syntactic constructor. It is composed of two "piped" modules: *lalr1* is the LALR(1) parser generator, which reads the grammar tables from `concrete.bt` and the disambiguation tables from `concrete.dt`, and produces a deterministic pushdown automaton described by a set of tables. `optim` takes control thereafter and optimizes the size of these tables by a very important ratio. The final tables are stored into `concrete.pt`.

- `tables_c` gathers the tables in `concrete.pt`, `concrete.st` and `concrete.rt` and translates them into initialized C structures to be stored into `concrete_t.c`.

- `tdef` reads the grammar tables from `concrete.bt` and the naming specification in `concrete.tdef` and produces a set of preprocessor constants (`#define`'s) into file `concrete.td.h`, which is to be `#include`'d into relevant C code, e.g. `concrete_sact.c`.

23.3 Compiling the Resulting C Code

This section gives only sketchy information. It will be improved in future releases of this manual.

23.3.1 The generated files

You will end up with some or all of the following C files:

- `concrete_t.c`
  the tables describing the scanner, the parser and other language-dependent information for the SYNTAX run-time system. This file `#include`'s `sxunix.h`. The top-level structure is called "sxtables", so that if you have to mix several languages into one program, you have to compile this file with option `-Dsxtables=concrete_sxtables` (for instance). Also, the name of the procedure which will be called upon each reduction is `SEMACT`. In the case of a tree constructor generated by ATC as described above, you also have to specify `-DSEMACT=f2atc` when compiling these tables. The role of `f2atc` is explained below. If you use actions and predicates at scanning and/or parsing time, you will have to define (using –**D** options) the names `SCANACT` and/or "PARSACT" respectively to be the name of the corresponding procedure.

- `abstract_atc.c`
  the tables and procedures defining the tree constructor, to be used by `f2atc`. There is one procedure for (almost) every production of the grammar. Those proced ures are addressed indirectly through a table. The top-level structure is called `F2_tables`, so th at if you have to mix several languages into one program, you have to compile this file with option `-DF2_tables=abstract_F2_tables` (for instance). This file `#include`'s `f2unix.h` and `f2atc.h`.

- `concrete.td.h`
  this optional "include" file was described in previous section.

23.3.2 C files you have to write

As stated in section 23.3.1, you will also have to write yourself the "main" procedure in file `concrete_main.c` (for instance). This file will `#include f2unix.h` and `f2atc.h` which itself includes `sxunix.h`.

If necessary, you will also have to write yourself your own scanning and/or parsing time actions and predicates, say in files `concrete_sact.c`and `concrete_pact.c` respectively. These files will include `sxunix.h`.

23.3.3 Library code

23.3.3.1 SYNTAX run-time library

The library `libsx.a` contains an important number of subprograms to be used in the generated analyzers: the scanner and the parser themselves, a module for error recovery, a string table manager[3], an error messages manager, a source manager, a procedure for listing output, etc. See *the SYNTAX Reference Manual* for more information.

The casual FNC-2 user has only to know that this library must be searched at linking time, using the **-l** option of `ld(1)`.

23.3.3.2 FNC-2 run-time library

File `f2atc.o` in directory `lib` in your installation of FNC-2 contains the "interpreter" of the tree construction tables generated by ATC, in the same way that the scanner and parser of SYNTAX run-time library are interpreters for the tables generated by SYNTAX. The most important procedure in this file is `f2atc`, which is called by the parser upon each reduction and in turn calls the corresponding tree construction action through the tables. The file also contains generic utility procedures for e.g. building a node.

File `f2atedit.o` contains the procedure `f2atedit` which pretty-print;s an FNC-2 tree, showing the name (operator) and the values of the attributes (of predefined types only) of each node. You can use this procedure after tree construction to visualize the result of your specification.

File `f2smp.o` contains the procedure `f2smp` which traverses from left to right and top to bottom an FNC-2 tree, calling at each visit of each node a procedure passed as parameter.

[3] This module is used to implement the OLGA type *token*.

23.3.4 Linking the whole

All you have to do now is compile and assemble the parts so that everything fits together.

23.4 Automatizing the Construction

You will find in directory `doc` in your installation of FNC-2 a number of files for `make(1)` which allow to specify the details of the construction process only once. You can choose one of these files and edit it to adapt it to your particular configuration.

Part VI

Appendices

Appendix A

A Short Example: *Simproc*

This fully commented example is the specification in Olga of the first pass of a compiler for the toy language *simproc*, used as an example in many other publications by members of the "Langages et Traducteurs" project at INRIA [Gau80, Des80, Jou84, Bon86].

A.1 Informal Description of *simproc*

Simproc—simple language with procedures—is a small language, which nevertheless embodies rather sophisticated notions dealing with procedures: block structure, recursion, parameters passed by value and by reference. It was originally designed to study how to express those notions using abstract data types [GDM78]. The concrete syntax of *simproc* is defined in section A.7 below.

Simproc is a block-structured language; the compilation unit is a block. Each block is composed of a perhaps-empty list of declarations followed by a never-empty list of statements; the declaration list of the outer block (main program) may not be empty, but this is only for semantic reasons: the statements available in *simproc* are such that at least one variable must be declared.

Simproc has only one simple type, integer, and one structured type, unidimensional array of integers with lower bound 1. A declaration introduces either an integer variable, an array variable with its size, or a procedure. As said before, an array always has a lower bound equal to one, so an array declaration only specifies its upper bound, which is the same as its size; this value must be a strictly positive integer constant. A procedure declaration specifies its name, the names of its formal parameters, and its body, which is a block. A procedure always has two integer parameters; the first one is passed by value whereas the second one is passed by reference.

Scope rules are borrowed from Algol, which means that an identifier is visible in the whole block in which it is declared, and also in nested blocks, unless it is redeclared in one of these. A given identifier can be declared only once in a given block. The identifiers denoting the formal parameters of a procedure are considered to be declared in the block which is its body. Procedures are recursive.

Available statements are integer assignment, procedure call and a simplified conditional. The latter statement allows to test the equality of two integer expressions and execute one of two never-empty statement lists according to the result of that comparison. Procedure calls have the same semantics as in e.g. Pascal: the actual parameter corresponding to the first formal parameter is passed by value and thus must be an expression, whereas the actual parameter corresponding to the second formal parameter is passed by reference and thus must be a variable. Inside the body of the procedure, the first parameter behaves as a local variable initialized with the value of the actual parameter, whereas any reference to the second parameter is a reference to the actual one.

Expressions are formed with variables, whose value is fetched, integer constants, usual arithmetic operators (addition, subtraction, multiplication and division) and parentheses for disambiguation. All the operators have the same precedence and associate from left to right. Variables are either simple variables, array variables indexed by an integer expression, or formal parameters.

Every identifier used in a block must be statically visible in that place according to the scope rules. Any use of an identifier must correspond to its declaration.

A.2 Description of the Translation of *simproc*

The goal of the attribute grammar to be described is to translate the source text of a program written in *simproc* into a term of the abstract data type describing that language [GDM78], provided that this program complies with the static semantic constraints of the language, as described above.

Since an abstract data type without equations is hardly more than an abstract syntax, we shall use an AAS (see section A.5 below) to describe that abstract data type. The generated term will then be a decorated tree described by that grammar.

Points to check with respect to semantic constraints are:

- uniqueness of the declaration of each identifier in each block;
- compliance of each use of an identifier with its declaration (simple variable or formal parameter, array variable, procedure);
- validity of parameter passing (by value or by reference).

There exist no type checking issue since all the variables and expressions are of type integer.

The attribute grammar will be split into two passes: the first one will check the semantic constraints whereas the second will produce the term. To stay simple, the latter will be designed so as to be run only if the result of the former is that the source text is completely correct. This must be ensured by external means; see *the* FNC-2 *Reference Manual.*

A.3 Abstract Syntax of *simproc*

Here comes *an* example of an attributeless abstract syntax for *simproc*. Many others are possible. It will reside in the file *simproc-base.asx* in the directory *examples* of the distribution of the FNC-2 system.

```
grammar simproc-base is

{ An attributeless abstract syntax for the language simproc }
```

```
{ with an example of a { nested } comment }

root is PROGRAM ;

{ program and blocks }

PROGRAM              =  program;
program              -> BLOCK;
BLOCK                =  block;
block                -> DECLS STMT;

{ declarations }

DECLS                =  decls;
decls                -> DECL*;
DECL                 =  int-decl array-decl proc-decl;
int-decl             -> ID {the variable id};
array-decl           -> ID {the array id}
                        NUMBER {the dimension};
proc-decl            -> ID {the procedure id}
                        ID {the value parameter id}
                        ID {the reference parameter id}
                        BLOCK;

{ statements }

STMT                 =  stmt-list assign call ifthenelse;
stmt-list            -> STMT+;
assign               -> VAR EXPR;
call                 -> ID { the procedure identifier }
                        EXPR { the value parameter }
                        VAR { the reference parameter };
ifthenelse           -> EXPR EXPR { the only comparison is equality }
                        STMT {true part} STMT {false part};

{ expressions }

EXPR                 =  bin-expr constant VAR { phylum inclusion };
bin-expr             -> OP EXPR EXPR;
constant             -> NUMBER;
OP                   =  plus minus mul div;
plus                 -> ;
minus                -> ;
mul                  -> ;
div                  -> ;
```

```
{ variables }

VAR                  =  simple-var indexed-var;
simple-var           -> ID { the variable identifier };
indexed-var          -> ID { the array id } EXPR { the index };

{ terminals }

NUMBER               =  number;
number               -> ;
ID                   =  id;
id                   -> ;

end grammar; { simproc-base }
```

The trees to be input to the first phase of the *simproc* compiler will be constructed during parsing, and are described by a simple AAS based on the attributeless AS defined above. It will reside in the file `simproc-in.asx` in the distribution.

```
grammar simproc-in is

import grammar simproc-base;

root is PROGRAM;

{ attributes }

$text (NUMBER): token;
$id (ID): token;

end grammar;
```

A.4 The Semantic Check Pass

A.4.1 The Symbol Table

The most important data structure in this pass is the symbol table. Such a table must associate with each declared identifier its *mode*, i.e. whether it is a variable, an array, a procedure or a formal parameter. It must also implement the block-structure.

A.4.1.1 The "symbol-table" module

We shall describe a parameterized module implementing such a symbol table, with one DCM and two DFMs. The specification is as follows:

- the module is called *symbol-table*;
- it is parameterized by two types and one function: *key-type* is the type of the keys to be searched in a symbol table; it must be an ordered type[1], and the `<` comparison operator on that type will be a parameter; *info-type* is the type of the information to associate with each identifier; this type can be any type;
- it declares an opaque type, *symbol-table*, which is the type of a symbol table, a parameterless exception *key-not-found*, an opaque value *empty-table* which is the empty symbol table, and a number of functions: *lookup* takes a key and a symbol table and returns the information associated to that key in that table, provided that the key exists; otherwise it will raise the exception *key-not-found*; if the key has been entered in several blocks, *lookup* will return the last (innermost) declaration; *lookup-in-block* takes a key and a symbol table and returns true if and only if the key has been entered in the current (innermost) block; *insert* takes a key, an information and a symbol table and returns a new symbol table in which the given key has been entered in the current block, associated to the given information; if that key was already present in the current block, the new declaration replaces the old one; lastly, *enter-block* takes a symbol table and returns a new symbol table in which a new empty block has been created in the innermost block.

Here comes the DCM, which will be found in file `symbol-table.olga`.

```
declaration module symbol-table
      (type key-type;
      function < (x, y:  key-type):  bool;
      type info-type) is

{ definition of a symbol table for block-structured languages }

type
   symbol-table;

exception
   key-not-found ();

value
   empty-table :  symbol-table;

function lookup (key:  key-type; st:  symbol-table):  info-type
   raise key-not-found;

function lookup-in-block (key:  key-type; st:  symbol-table):
   bool;

function insert (key:  key-type; info:  info-type;
```

[1] This is actually not required for the first implementation, but we wished to have a single DCM for both implementations.

```
        st: symbol-table): symbol-table;

function enter-block (st: symbol-table): symbol-table;

end module;
```

The first implementation uses a linear list. It is very simple. It will be found in the file `linear-list.olga`.

```
definition module symbol-table is

{ implementation of a symbol table for block-structured languages
using linear lists }

type
   true-element   =  record
                        key:  key-type;
                        info  info-type;
                     end record;
   symbol-table-element    =  union
                                 void
                                 { for separating blocks };
                                 true-element;
                              end union;
   symbol-table = list of symbol-table-element;

value
   empty-table := symbol-table (symbol-table-element (null));

function lookup (key: key-type; st: symbol-table): info-type
   raise key-not-found is
   if empty (st) then
      raise key-not-found ()
   else
      case head (st) match
         symbol-table-element (null: void):
            lookup (key, tail (st));
            { skip over block boundaries }
         symbol-table-element (node: true-element):
            if node.key = key then
               node.info
            else
               lookup (key, tail (st))
            end if;
      end case
   end if
```

```
end function { lookup };

function lookup-in-block (key:  key-type; st:  symbol-table):
      bool is
{ here we don't need to test the emptiness since there is always a
   block in a "symbol-table" and we don't search past it }
   case head (st) match
      symbol-table-element (null:  void):
         false { do not skip over block boundaries };
      symbol-table-element (node:  true-element):
         if node.key = key then
            true
         else
            lookup-in-block (key, tail (st))
         end if;
   end case
end function { lookup-in-block };

function insert (key:  key-type; info:  info-type;
            st:  symbol-table):  symbol-table is
   symbol-table (symbol-table-element (true-element (key, info)))
      + st
end function { insert };

function enter-block (st:  symbol-table):  symbol-table is
   symbol-table (symbol-table-element (null)) + st
end function { enter-block};
end module;
```

The second implementation uses as basic structure a binary search tree for each block, linked in a linear list starting with the innermost block up to the outermost one. It is somewhat more complicated than the first but is expected to have better performance for large source texts. It will be found in the file `binary-tree.olga`.

```
definition module symbol-table is

{ implementation of a symbol table for block-structured languages
using a list of binary trees }

type
   tree-node   =  record
                     key :  key-type;
                     info :  info-type;
                     left-son, right-son :  bin-tree;
                  end record;
and
```

```
   bin-tree =  union
                 void { for leaves };
                 tree-node;
              end union;
   symbol-table = list of bin-tree;

value
   leaf := bin-tree (null) { this value is local };
   empty-table := symbol-table (leaf);

function lookup-in-tree (key:  key-type; node:  bin-tree):
      info-type raise key-not-found is
   { this function is local to the module }
   case node match
      bin-tree (null:  void):
         raise key-not-found ();
      bin-tree (tree-node (the-key, the-info, left-s, right-s)):
         if the-key = key then
            the-info
         elsif the-key < key then
            lookup-in-tree (key, left-s)
         else
            lookup-in-tree (key, right-s)
         end if;
   end case
end function { lookup-in-tree };

function lookup (key:  key-type; st:  symbol-table):  info-type
   raise key-not-found is
   if empty (st) then
      raise key-not-found ()
   else
      lookup-in-tree (key, head (st))
      catch
         key-not-found:
            lookup (key, tail (st));
      end catch
   end if
end; function { end lookup }

function lookup-in-block (key:  key-type; st:  symbol-table):
         bool is
{ here we shall use a "bad trick" to avoid rewriting
   "lookup-in-tree" }
declare
   function dummy-eval (x:  info-type):  bool is
```

```
            true
        end function;
    in
        dummy-eval (lookup-in-tree (key, head (st)))
        catch
            key-not-found:
                false;
        end catch
    end function { end lookup-in-block };

    function insert-in-tree (key:  key-type; info:  info-type;
                    node:  bin-tree):  bin-tree is
    { this function is also local }
        case node match
            bin-tree (null:  void):
                { insert here, while sharing as much as possible }
                bin-tree (tree-node (key, info, node, node));
            bin-tree (tree-node (the-key, the-info, left-s, right-s)):
                if the-key = key then
                    { replace old entry }
                    bin-tree (tree-node (key, info, left-s, right-s))
                elsif the-key < key then
                    bin-tree (tree-node (the-key, the-info,
                        insert-in-tree (key, info, left-s),
                        right-s))
                else
                    bin-tree (tree-node (the-key, the-info, left-s,
                        insert-in-tree (key, info, right-s)))
                end if;
        end case
    end function { end insert-in-tree };

    function insert (key:  key-type; info:  info-type;
                st:  symbol-table):  symbol-table is
        symbol-table (insert-in-tree (key, info, head (st)))
            + tail (st)
    end function { insert };

    function enter-block (st:  symbol-table):  symbol-table is
        symbol-table (leaf) + st
    end function { enter-block};

    end module;
```

The function *insert-in-tree* needs some further explanation. What must be implemented is a non-destructive insertion, such that the old symbol table and the new one share as

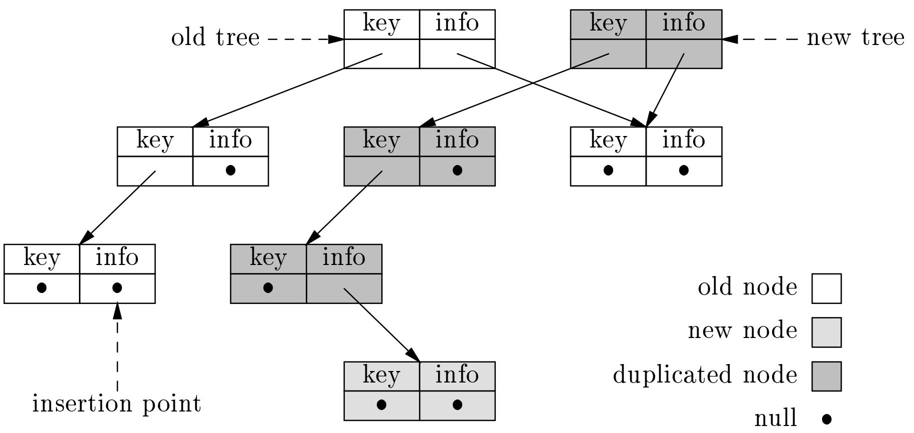


Figure A.1: shared-tree

many nodes as possible; this is achieved by duplicating the nodes on the path from the root of the tree to the insertion point, as shown in Fig. A.1. Only one node is shared in that example (the rightmost one in second row), but this sharing can be more effective for larger trees. Note that the applicative character of OLGA somewhat enforces this programming style. Non-destructive shared data structures have been shown very useful for attribute grammars [Mye84].

In next release the same module will also be implemented with 2-3 trees.

A.4.1.2 Tailoring the "symbol-table" module to "simproc"

For *simproc* we shall choose *token* as the *key-type*, and an enumerated type *mode* describing the possible denotations of an identifier as the *info-type*.

Since attributes of type *symbol-table* will be attached to the output AAS of the first phase, and since it is forbidden to instantiate a parameterized module in an AAS, we shall define a module *simproc-types* defining the attributes types used in the *simproc* compiler. The corresponding DCM and DFM will be found in files `simproc-types.olga` and `simproc-types-def.olga` respectively.

```
declaration module simproc-types is

   type
      mode  = (varid, refid, arrayid, procid);
      { respectively for simple variables and value parameters,
         reference parameters, arrays and procedures }
      operator = (wrong-op, addition, subtraction,
            multiplication, division);

   from symbol-table (token, <, mode) import all;

end module;

definition module simproc-types is
```

```
end module;
```

Let us now come back to the *simproc* compiler. Since an identifier is visible in the whole block in which it is declared, in particular *before* its declaration, two attributes of type *symbol-table* are needed in each block: a compound one `$temp-st` which contains the symbol table being elaborated in the declaration part, and a global one `$symtab` which contains the complete table, to be propagated to the statement part and to the inner blocks. The latter attribute will appear on some nodes of the output AAS.

A.4.2 The output AAS

The output AAS will be the same as the input one except for attributes.

```
grammar simproc-out is

import grammar simproc-base
from simproc-types import { all } symbol-table, operator, mode;

root is PROGRAM;

{ attributes }

$symtab (BLOCK) : symbol-table;
$op (OP) : operator;
$value (NUMBER) : int { replaces $text };
$id (ID) : token { the only one also appearing in "simproc-in" };
$correct (PROGRAM) : bool;
end grammar;
```

A.4.3 Other Attributes

Two boolean attributes check the static correctness of parts of the source text: `$correct1` reflects the correctness of the declaration part of each bock, and `$correct2` that of the statement part and of the inner blocks. It was necessary to split this notion of correctness into two attributes for circularity reasons.

A.4.4 The Attribute Grammar

The following attribute grammar will reside in the file `simproc-check.olga` in the directory `example` in the distribution of the FNC-2 system.

```
declaration module simproc-check is
import grammar simproc-in;
function simproc-check (r:PROGRAM;r1:PROGRAM) : bool;
end module;
```

```
attribute grammar simproc-check (simproc-in in PROGRAM; simproc-out out PROGRAM) : ($correct:bool) is

from simproc-types import all;

attribute
   { working attributes }
   compound $s-temp-st and $h-temp-st (BLOCK, DECLS, DECL):
                        symbol-table;
   synthesized $correct1 (BLOCK, DECLS, DECL): bool;
   synthesized $correct2 (BLOCK, DECLS, DECL, STMT, EXPR, VAR,
               NUMBER): bool;
   { kinds of result attributes }
   synthesized $value ();
   synthesized $op ();
   global $symtab ();
   synthesized $correct ();

{ let's go for semantic rules }
{ program and blocks }

where program -> BLOCK use
   $correct := $correct1(BLOCK) & $correct2(BLOCK);
   $h-temp-st(BLOCK) := empty-table;
   $symtab(BLOCK) := $s-temp-st(BLOCK);
end where;

where null-PROGRAM -> use
   $correct := false;
end where;

where block -> DECLS STMT use
   $correct1 := $correct1(DECLS);
   $correct2 := $correct2(DECLS) & $correct2(STMT);
   $s-temp-st := $s-temp-st(DECLS);
end where;

where null-BLOCK -> use
   $correct1 := false;
   $correct2 := false;
end where;

{ declarations }

where decls -> DECL* use
   $correct1 :=
               map left &
```

```
                        value true
                        other $correct1(DECL)
                     end map;
   $correct2 :=
                     map left &
                        value true
                        other $correct2(DECL)
                     end map;
   $h-temp-st(DECL):=
   { this definition appears here only to provide an example of a
   case position construct; it could have been generated
   automatically using the default rules }
            case position is
               first :  $h-temp-st;
               other :  $s-temp-st(DECL.left);
            end case;
end where;

where null-DECLS -> use
   $correct1 := false;
   $correct2 := false;
end where;

where int-decl -> ID
declare
   value
      correct :=
               if $id(ID) = error-token then
                  false
               elsif lookup-in-block ($id(ID), $h-temp-st)
                              then
                  message "Identifier already declared."
                     position ID
                     severity 2
                     value false
               else
                  true
               end if;
use
   $correct1 := correct;
   $correct2 := true;
   $s-temp-st :=
               if correct then
                  insert ($id(ID), varid, $h-temp-st)
               else
                  $h-temp-st
```

```
               end if;
end where;

where array-decl -> ID NUMBER
declare
   value
      correct :=
               (if $id(ID) = error-token then
                  false
               elsif lookup-in-block ($id(ID),
                              $h-temp-st) then
                  message "Identifier already declared."
                     position ID
                     severity 2
                     value false
               else
                  true
               end if) &
               ($correct2(NUMBER) and
               if $value(NUMBER) = 0 then
                  message "Size cannot be null"
                     position NUMBER
                     severity 2
                     value false
               else
                  true
               end if);
use
   $correct1 := correct;
   $correct2 := true;
   $s-temp-st :=
               if correct then
                  insert ($id(ID), arrayid, $h-temp-st)
               else
               $h-temp-st
               end if;
end where;

where proc-decl -> ID1 ID2 ID3 BLOCK
declare
   value
      correct :=
            (if $id(ID1) = error-token then
               false
            elsif lookup-in-block ($id(ID1),
                           $h-temp-st) then
```

```
                message "Identifier already declared."
                   position ID1
                   severity 2
                   value false
             else
                true
             end if) &
             (if $id(ID2) = error-token or
                $id(ID3) = error-token then
                false
             elsif $id(ID2) = $id(ID3) then
                message
             "Formal parameters may not have the same name."
                   position ID3
                   severity 2
                   value false
             else
                true
             end if);
use
   $correct1 := correct;
   $correct2 := $correct1(BLOCK) & $correct2(BLOCK);
   $s-temp-st :=
             if correct then
                insert ($id(ID1), procid, $h-temp-st)
             else
                $h-temp-st
             end if;
   $h-temp-st(BLOCK) :=
                insert ($id(ID3), refid,
                      insert ($id(ID2), varid,
                         enter-block ($symtab[BLOCK])));
   $symtab(BLOCK):= $s-temp-st(BLOCK);
end where;

where null-DECL -> use
   $correct1 := false;
   $correct2 := false;
end where;

{ statements }

where stmt-list -> STMT+ use
   $correct2 :=
             map left &
                value true
```

```
          other $correct2(STMT)
        end map;
end where;

where assign -> VAR EXPR use
  $correct2 := $correct2(VAR) & $correct2(EXPR);
end where;

where ifthenelse -> EXPR1 EXPR2 STMT1 STMT2 use
  $correct2 := $correct2(EXPR1) & $correct2(EXPR2) &
        $correct2(STMT1) & $correct2(STMT2);
end where;

where call -> ID EXPR VAR use
  $correct2 :=
        (if $id(ID) = error-token then
          false
        else
          if lookup ($id(ID), $symtab[BLOCK])
            != procid then
            message "Not a procedure."
            position ID
            severity 2
            value false
          else
            true
          end if
          catch
            key-not-found:
            message "Undeclared identifier."
              position ID
              severity 2
              value false;
          end catch
        end if) &
        $correct2(EXPR) & $correct2(VAR);
end where;

where null-STMT -> use
  $correct2 := false;
end where;

{ expressions and operators }

where bin-expr -> OP EXPR1 EXPR2 use
  $correct2 := $correct2(EXPR1) & $correct2(EXPR2) &
```

```
            $op(OP)!= wrong-op;
end where;

where constant -> NUMBER use
end where;

where EXPR -> VAR use
end where;

where null-EXPR -> use
   $correct2 := false;
end where;

where plus -> use
   $op := addition;
end where;

where minus -> use
   $op := subtraction;
end where;

wheremul -> use
   $op := multiplication;
end where;

where div -> use
   $op := division;
end where;

where null-OP -> use
   $op := wrong-op;
end where;

{ variables }

where simple-var -> ID use
   $correct2 :=
            if $id(ID) = error-token then
               false
            else
               let mode:= lookup ($id(ID),
                           $symtab[BLOCK])
               in if mode = varid or
                  mode = refid then
                     true
                  else
```

```
                    message "Not a variable."
                        position ID
                        severity 2
                        value false
                end if
            catch
                key-not-found:
                message "Undeclared identifier."
                    position ID
                    severity 2
                    value false;
            end catch
        end if;
end where;

where indexed-var -> ID EXPR use
    $correct2 :=
            (if $id(ID) = error-token then
                false
            else
                if lookup ($id(ID), $symtab[BLOCK])
                    != arrayid then
                    message "Not an array."
                        position ID
                        severity 2
                        value false
                else
                    true
                end if
                catch
                    key-not-found:
                    message "Undeclared identifier."
                        position ID
                        severity 2
                        value false;
                end catch
            end if) & $correct2(EXPR);
end where;

{ numbers }

where number -> use
    $correct2 := $text!= error-token;
    $value :=
        if $text = error-token then
            0
```

```
        else
           int (string ($text))
        end if;
end where;

end grammar { simproc-check };
```

Note the various uses of strict and non-strict boolean operators and of conditional expressions in the different definitions of the "correct" local declared values so that as many errors as possible are detected in a single scan of the text, even if some syntactic errors occurred.

A.5 Description of *simproc* Abstract Data Type

As said before, the description of that abstract data type (types and operations) will take the form of an attributed abstract syntax. It will reside in file `simproc-adt.asx`.

```
grammar simproc-adt is

{ To stay coherent with usual notations for abstract data types, both
phyla and operator names will be written as capitalized, lower-case
identifiers.
This has the additional effect that those names will not conflict
with keywords and/or predefined identifiers.  }

root is Program;

Program             =  Execute;
Execute             -> Source-modif;

{ source modifications, i.e. statements and alike }

Source-modif        =  Void Concat Assignint Assignvar Possesses
                       Condit Procdecl Intdecl Arraydecl Proccall;
Void                -> ;
Concat              -> Source-modif+;
Assignint           -> Var Int;
Assignvar           -> Refvar Int;
Possesses           -> Procid;
Condit              -> Int Int Source-modif Source-modif;
Procdecl            -> Procid Varid Refid Procid Source-modif;
Intdecl             -> Varid Procid;
Arraydecl           -> Arrayid Int Procid;
Proccall            -> Procid Int Var Procid;
Call                =  Currentcall;
Currentcall         -> ;
```

```
{ identifiers }

Varid                = VarId;
VarId                -> ;
Refid                = RefId;
RefId                -> ;
Arrayid              = ArrayId;
ArrayId              -> ;
Procid               = ProcId;
ProcId               -> ;

{ variables and alike }

Var                  = Designates Refersto Element;
Designates           -> Varid Call;
Refersto             -> Refvar;
Element              -> Arrayid Call Int;
Refvar               = Refdesignates;
Refdesignates        -> Refid Call;

{ integers, that is expressions }

Int                  = Plus Difference Times Quotient Valueof Const;
Plus                 -> Int Int;
Difference           -> Int Int;
Times                -> Int Int;
Quotient             -> Int Int;
Valueof              -> Var;
Const                -> NumberConst;
NumberConst          = Number;
Number               -> ;

{ Attributes attached to the resulting term }

$value (NumberConst) :  int;
$id (Varid, Refid, Arrayid, Procid) :  token;

end grammar;
```

A.6 The Translation Pass

The attribute grammar will be found in the file `simproc-term.olga`.

declaration module `simproc-term` **is**

```
from simproc-types import all ;
import grammar simproc-in;
import grammar simproc-adt;

function simproc-term (r:PROGRAM) : Program;

end module;


attribute grammar simproc-term (simproc-out in PROGRAM) : ($Program:  Program) is

from simproc-types import all;

attribute
   { for computing the resulting term of the source ADT }
   synthesized $Program (PROGRAM): Program;
   synthesized $Model (BLOCK, DECLS, DECL, STMT)
                        :  Source-modif;
   synthesized $Var (VAR): Var;
   synthesized $Value (EXP, NUMBER): Int;
   inherited $procid (BLOCK): token { name of enclosing proc};

{ let's go for semantic rules }
{ program and blocks }

where program -> BLOCK use
   $Program := Execute ($Model(BLOCK));
   $procid(BLOCK) := token ("Prog") { we have to invent it...  };
end where;

where null-PROGRAM -> use
   { should never occur, as any other null-XXX, since the source
   text has been checked to be completely correct when this
   attribute grammar is applied.  }
   $Program := Execute (Void ());
end where;

where block -> DECLS STMT use
   $Model:= { try to build the minimum tree }
         case $Model(DECLS) match
            Void () { no declaration }:
            $Model(STMT);
            Concat-all (D) { several declarations }:
            case $Model(STMT) match
               Concat-all (S) { several statements }:
               Concat-merge (D, S);
               S { a unique statement }:
```

```
            Concat-post ($Model(DECLS), S);
         end case;
         D { a unique declaration }:
         case $Model(STMT) match
            Concat-all (S) { several statements }:
            Concat-pre (D, $Model(STMT));
            S { a unique statement }:
            Concat (D, S);
         end case;
      end case;
end where;

where null-BLOCK -> use
   $Model := Void ();
end where;

{ declarations }

where decls -> DECL* use
   $Model :=
      case arity is
         0 :  Void ();
         1 :  $Model(DECL);
         other :  map left Concat-post
                  value Concat ()
                  other $Model(DECL)
               end map;
      end case;
end where;

where null-DECLS -> use
   $Model := Void ();
end where;

where int-decl -> ID use
   $Model := Intdecl (VarId ()
            with $id := $id(ID)
            end with,
            ProcId ()
            with $id := $procid[BLOCK]
            end with);
end where;

where array-decl -> ID NUMBER use
   $Model :=
         Arraydecl (ArrayId ()
```

```
                with $id := $id(ID)
                end with,
                $Value(NUMBER),
                ProcId ()
                with $id := $procid[BLOCK]
                end with);
end where;

where proc-decl -> ID1 ID2 ID3 BLOCK use
   $Model :=
             Procdecl (ProcId ()
                with $id := $id(ID1)
                end with,
                VarId ()
                with $id:= $id(ID2)
                end with,
                RefId ()
                with $id := $id(ID3)
                end with,
                ProcId ()
                with $id := $procid[BLOCK]
                end with,
                $Model(BLOCK));
   $procid(BLOCK) := $id(ID1);
end where;

where null-DECL -> use
   $Model := Void ();
end where;

{ statements }

where stmt-list -> STMT+ use
   $Model :=
         case arity is
            1 :  $Model(STMT);
            other :  map left Concat-post
                     value Concat ()
                     other $Model(STMT)
                  end map;
         end case;
end where;

where assign -> VAR EXPR use
   $Model :=
         case $Address(VAR) match
```

```
            Refersto (AR) :
               Assignvar (AR, $Value(EXPR));
            other :
               Assignint ($Address(VAR), $Value(EXPR));
         end case;
end where;

where ifthenelse -> EXPR1 EXPR2 STMT1 STMT2 use
   $Model :=
            Condit ($Value(EXPR1), $Value(EXPR2),
               $Model(STMT1), $Model(STMT2));
end where;

where call -> ID EXPR VAR use
   $Model :=
            Proccall (ProcId () { callee }
               with $id := $id(ID)
               end with,
               $Value(EXPR),
               $Address(VAR),
               ProcId () { caller }
               with $id := $procid[BLOCK]
               end with);
end where;

where null-STMT -> use
   $Model := Void ();
end where;

{ expressions and operators }

where bin-expr -> OP EXPR1 EXPR2 use
   $Value :=
         case $op(OP) is
            addition, wrong-op :
               Plus ($Value(EXPR1), $Value(EXPR2));
            subtraction :
               Minus ($Value(EXPR1), $Value(EXPR2));
            multiplication :
               Times ($Value(EXPR1), $Value(EXPR2));
            division :
            Quotient ($Value(EXPR1), $Value(EXPR2));
         end case;
end where;

where constant -> NUMBER use
```

```
end where;

where EXPR -> VAR use
   $Value := Valueof ($Address(VAR));
end where;

where null-EXPR -> use
   $Value :=
         Const (Number ()
            with $value := 0
            end with);
end where;

{ No new attributes on phylum OP }

{ variables }

where simple-var -> ID use
   $Address :=
         let id := $id(ID)
         in case lookup (id, $symtab[BLOCK]) is
               refid :
               Refersto (Refdesignates (
                        RefId ()
                        with $id := id
                        end with,
                        Currentcall ()));
               other :  { includes "varid" }
               Designates (VarId ()
                     with $id := id
                     end with,
                     Currentcall ());
            end case;
end where;

where indexed-var -> ID EXPR use
   $Address :=
            Element (ArrayId ()
               with $id := $id(ID)
               end with,
               Currentcall (),
               $Value(EXPR));
end where;

{ numbers }
```

```
where number -> use
   $Value :=
         Const (Number ()
            with $Value := $value
            end with);
end where;

end grammar { simproc-term };
```

Note the various uses of tree construction and decoration functions, and of pattern matching.

A.7 Concrete Representation of *simproc*

Although the concrete representation of programs written in *simproc* is of no concern for the FNC-2 system, we present it hereafter in order to make this example self-contained and "executable". The following files are to be processed by the SYNTAX system and *atc* to generate a scanner, a parser and a tree constructor; see Part V in this volume.

A.7.1 Concrete Syntax and Tree Construction Actions

The following specification will be found in the file `simproc.atc`. Note that all the tree construction actions have been completely explicited, but most of them could be abbreviated as a single operator name.

```
import grammar simproc-in;

{ program and blocks }
<PROGRAM>              =  <BLOCK>;
                       program (<BLOCK>) { could have been abbreviated as "program" }
<BLOCK>                =  <DECL *> <STMT +>;
                       block (<DECL *>, <STMT +>)

{ declarations }
<DECL *>               =  <DECL *> <DECL>;
                       decls-post (<DECL *>, <DECL>)
<DECL *>                  =;
                       decls ()
<DECL>                 =  integer <id> ";";
                       int-decl (<id>)
<DECL>                 =  array <id> "[" <int> "]" ";";
                       array-decl (<id>, <int>)
<DECL>                 =  proc <id> "(" val <id> "," var <id> ")"
                          begin <BLOCK> end ";";
                       proc-decl (<id>.1, <id>.2, <id>.3, <BLOCK>)

{ statements }
```

```
<STMT +>            =  <STMT +> ";" <STMT>;
                    stmt-list-post (<STMT +>, <STMT>)
<STMT +>            =  <STMT>;
                    stmt-list (<STMT>)
<STMT>              =  <VAR> ":=" <EXPR>;
                    assign (<VAR>, <EXPR>)
<STMT>              =  call <id> "(" <EXPR> "," <VAR> ")";
                    call (<id>, <EXPR>, <VAR>)
<STMT>              =  if <EXPR> eq <EXPR> then <STMT +> else <STMT +> fi;
                    ifthenelse (<EXPR>.1, <EXPR>.2, <STMT +>.1, <STMT +>.2)

{ expressions and operators }
<EXPR>              =  <EXPR> <OP> <TERM>;
                    bin-expr (<OP>, <EXPR>, <TERM>)
<EXPR>              =  <TERM>;
<TERM>              =  <VAR>;
<TERM>              =  "(" <EXPR> ")";
                    <EXPR>
<TERM>              =  <int>;
                    constant (<int>)
<OP>                   = "+";
                    plus ()
<OP>                =  "-";
                    minus ()
<OP>                =  "*";
                    mul ()
<OP>                =  "/";
                    div ()

{ variables }
<VAR>               =  <id>;
                    simple-var (<id>)
<VAR>               =  <id> "[" <EXPR> "]";
                    indexed-var (<id>, <EXPR>)

{ terminals }
<id>                =  %ID;
                    id () with $id := %ID end with
<int>               =  %INT;
                    number () with $text := %INT end with
```

A.7.2 Lexical Description

The following specification will be found in the file `simproc.lecl`.

Classes

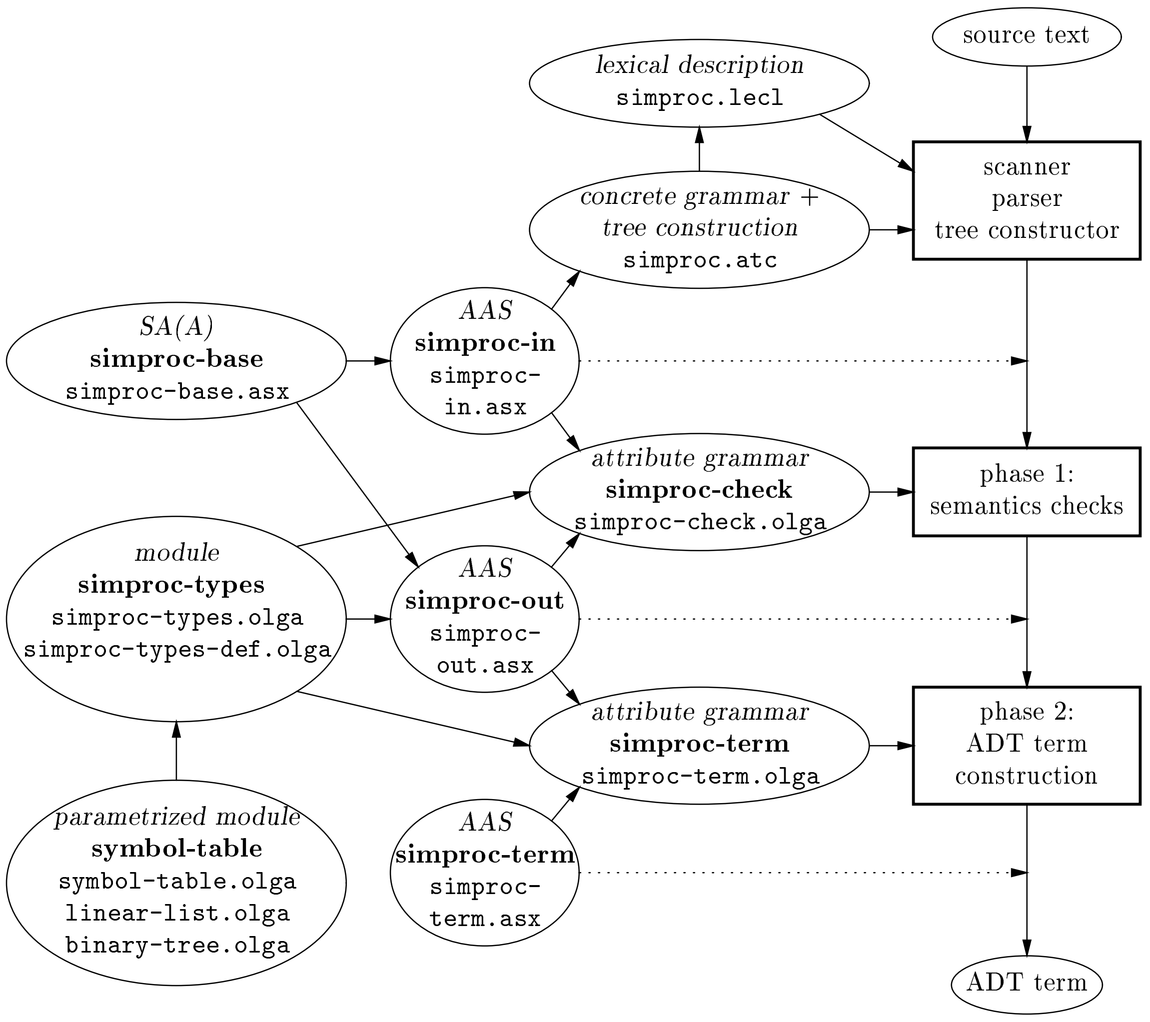


Figure A.2: simproc-org

```
SPACE    = SP + HT + NL + FF;
```

Tokens

```
Comments        = -{SPACE | "%" "%" {^EOL}* EOL }+;
%ID             = LETTER {[_"] (LETTER | DIGIT)}*;
%INT            = {DIGIT}+;
```

A.8 Module Dependency Graph for *simproc*

The organization of the *simproc* compiler and of the files composing its description is depicted in Fig. A.2. The processors used to generate the different components of the compiler from the specification files are not drawn. The generated compiler is shown on the right, and the downward arrows represent the flow of data. The modules composing the description are shown on the left, together with their kind—e.g. module or attribute grammar—, their

OLGA name and the name(s) of the file(s) which compose them. The arrows represent module dependencies, except for arrows from an AAS to an arrow in the compiler which tell that the trees flowing through the latter arrow are described by the AAS.

Appendix B

OLGA Lexicon and Concrete Syntax Summary

This chapter summarizes the specification of the concrete representation of the OLGA language and of its sublanguage for abstract syntaxes descriptions, ASX.

B.1 Lexical Description

This description applies to both OLGA and ASX.

```
-- ---------------------------------------------------------
-- Olga.lecl
-- ---------------------------------------------------------
-- Lexical description of "OLGA" and "asx" for SYNTAX scanner
-- generator "lecl"
-- See "the OLGA Reference Manual", chapter 6
-- ---------------------------------------------------------
```

Classes

```
   LETTER          = "a".."z" + "A".."Z";
   DIGIT           = "0".."9";
   OCTAL           = "0".."7";
   TRULY_PRINTABLE = " "..#176 - "^\'\"\\";
         -- printable ASCII characters except those
         -- participating in escape sequences
   CONTROL_CHAR    = "A".."_";
         -- printable characters which correspond to
         -- control characters with the "↑" notation
         -- except @ (^@ == NUL character, forbidden)
   BLANKS          = SP + HT + VT + FF + LF + NL + CR;
         -- the above identifiers are predefined in the
         -- lexical metalanguage of SYNTAX
```

Abbreviations

```
MEANINGFUL     = "{" @1 {^"{}"&True | "{"&True @1 | "}"&1 @2}*
                           ("}"&2 @2 | @7);
      -- initialization:  comment-level = 0
      -- &1:  true if comment-level > 1
      -- &2 = not &1
      -- @1:  increase comment-level
      -- @2:  decrease comment-level
      -- @7:  (on EOF only) complain if comment-level!= 0
ID             = LETTER {["_-"] (LETTER | DIGIT)}*;
SIGN           = [-"+" | "-"];
INT            = {DIGIT}+;
PRINTABLE      = TRULY_PRINTABLE
      | -"^" CONTROL_CHAR @4
      | -"\\" (TRULY_PRINTABLE @5 |
         "^\'\"\\" |
         OCTAL&True [OCTAL& True [OCTAL& True]] @6);
      -- @4:  convert a control character from "↑" notation
      -- @5:  recognize and convert \c escape sequence
      -- @6:  convert a character from its octal value
      -- and complain if NUL
```

Tokens

```
Comments       = -{BLANKS | MEANINGFUL}+;
%ID            = ID;
%INT           = INT;
%REAL          = INT "." INT [("e" | "E" @Lower_Case SIGN INT];
%CHAR          = -"'" PRINTABLE -"'";
%STRING        = -"\"" {PRINTABLE} -"\"";
%ATTR          = "$" ID;
%TVAR          = "@" ID;
```

The following operators, separators and keywords are the other lexical units of OLGA:

`- ! < > [ ] . -> | & >= <= !~ ~ != % / * + .. = # , ) ( := : ; [| |]`

$arity $position $scx all and arity attribute case catch compound const declaration declare define definition else elsif end exception external first from function global grammar if import in inherited inline is last left let letrec list map match message module of or other phase position raise range record right set severity synthesized test then tree type undef union use value where with

Those which do not appear in the syntactic description of the abstract syntax description language ASX are to be considered as illegal tokens. ASX also has the keyword **root** and the separator +=.

B.2 Syntactic Description of the AAS Description Language

This specifies the concrete syntax of language Asx.

```
* asx.bnf
* Concrete syntax of the OLGA sublanguage for AAS description
* See "the OLGA Reference Manual", chapter 12

<abstract syntax>      =  <header>
                             <import *>
                             <definition +>
                          end grammar ";";

<header>               =  grammar %ID is ;

<import *>             =  <import *>
                          <import>;
<import *>             =  ;
<import>               =  import grammar %ID ";";
<import>               =  from %ID import all ";";
<import>               =  from %ID import <import ids +> ";";
<import>               =  import %ID ";";
<import ids +>         =  <import ids +> "," <import id>;
<import ids +>         =  <import id>;
<import id>            =  %ID ;

<definition +>         =  <definition +>
                          <definition>;
<definition +>         =  <definition>;

<definition>           =  <phylum def>;
<definition>           =  <operator def>;
<definition>           =  <attribute decl>;
<definition>           =  <root def>;

<root def>             =  root is %ID ";";

<attribute decl>       =  %ATTR "(" <phylum, +> ")" ":" <type id> ";";
<type id>              =  %ID "#" %ID ;
<type id>              =  %ID ;
<phylum, +>            =  <phylum, +> "," <phylum-name>;
<phylum, +>            =  <phylum-name>;
<phylum-name>          =  %ID ;

<phylum def>           =  %ID "=" <op-or-phy *> ";";
<phylum def>           =  %ID "+=" <op-or-phy *> ";";
```

```
<op-or-phy *>          =  <op-or-phy *> <op-or-phy>;
<op-or-phy *>          =  ;
<op-or-phy>            =  %ID ;

<operator def>         =  %ID "->" <phylum> "+" ";";
<operator def>         =  %ID "->" <phylum> "*" ";";
<operator def>         =  %ID "->" <phyla *> ";";
<phyla *>              =  ;
<phyla *>              =  <phyla +>;
<phyla +>              =  <phyla +> <phylum>;
<phyla +>              =  <phylum>;
<phylum>               =  %ID;
```

B.3 Syntactic Description of Olga

This summarizes the concrete syntax of Olga. The order of the chapters is not respected, rather a "top-down" approach is used.

```
* olga.bnf
* Concrete syntax of the Olga language
* See "the Olga Reference Manual"; the chapter and section numbers
*                      refer to that manual.
* The axiom of the grammar is <compilation unit>

* Compilation units; see chapters 11 and 15
<compilation unit>     =  <attribute grammar>;
<compilation unit>     =  <DC module>;
<compilation unit>     =  <DF module>;

* Declaration modules:  section 11.3.1
<DC module>            =  declaration module <ID> <DCM parm *> is
                             <DCM decl +>
                          end module ";";
<DCM decl +>           =  <DCM decl +>
                          <DCM decl>;
<DCM decl +>           =  <DCM decl>;
<DCM decl>             =  <DCM const part>;
<DCM decl>             =  <DCM value part>;
<DCM decl>             =  <DCM type part>;
<DCM decl>             =  <function header> ";";
<DCM decl>             =  <exception part>;
<DCM decl>             =  <import clause>;
<DCM const part>       =  const
                             <DCM const decl +>;
<DCM const decl +>     =  <DCM const decl +>
```

```
                          <DCM const decl>;
<DCM const decl +>    =  <DCM const decl>;
<DCM const decl>      =  <const decl>;
<DCM const decl>      =  <ID> ":" <type id> ";";
<DCM value part>      =  value
                            <DCM value decl +>;
<DCM value decl +>    =  <DCM value decl +>
                          <DCM value decl>;
<DCM value decl +>    =  <DCM value decl>;
<DCM value decl>      =  <ID> ":" <type id> ":=" <expression> ";";
<DCM value decl>      =  <ID> ":=" <expression> ";";
<DCM value decl>      =  <ID> ":" <type id> ";";
<DCM type part>       =  type
                            <DCM type decl +>;
<DCM type decl +>     =  <DCM type decl +>
                          <DCM type decl>;
<DCM type decl +>     =  <DCM type decl>;
<DCM type decl>       =  <type decl>;
<DCM type decl>       =  <ID> ";";

* Definition modules:  section 11.3.2
<DF module>           =  definition module <ID> is
                            <decl *>
                          end module ";";

* Parametrized modules:  section 11.3.3
<DCM parm *>          =  ;
<DCM parm *>          =  "(" <DCM parm +> ")";
<DCM parm +>          =  <DCM parm +> ";"
                          <DCM parm>;
<DCM parm +>          =  <DCM parm>;
<DCM parm>            =  type <ID>;
<DCM parm>            =  function <function id>
                            "(" <DCM param group *> ")" ":" <type id>
                            <DCM func excp *>;
<DCM param group *>   =  ;
<DCM param group *>   =  <DCM param group +>;
<DCM param group +>   =  <DCM param group +> "," <DCM param group>;
<DCM param group +>   =  <DCM param group>;
<DCM param group>     =  <identifier +> ":" <type id>;
<DCM func excp *>     =  ;
<DCM func excp *>     =  raise <DCM func excp +>;
<DCM func excp +>     =  <DCM func excp +> "," <DCM excp id>;
<DCM func excp +>     =  <DCM excp id>;
<DCM excp id>         =  <ID>;
<DCM excp id>         =  <ID> "(" <DCM param group *> ")";
```

```
* Utility non-terminals:  chapter 6
* <ID> is used in declaring a new identifier,
* except in the productions defining <id>.
<ID>                  =  %ID ;
<INT>                 =  %INT ;

* Attribute grammars and attribute passes:  section 15.1
<attribute grammar>   =  <AG header>
                            <AG decl *>
                            <semantics>
                         end grammar ";";
<AG header>           =  attribute grammar <ID> <AG profile> is ;
<AG profile>          =  "(" <AAS name> "," <AAS name> ")";
<AG profile>          =  "(" <AAS name> ")" ":" "(" <AAS *> ")";
<AAS *>               =  ;
<AAS *>               =  <AAS +>;
<AAS +>               =  <AAS +> "," <qual AAS name>;
<AAS +>               =  <qual AAs name>;
<qual AAS name>       =  <AAS name>;
<qual AAS name>       =  <AAS name> ":" %ATTR;
<AAS name>            =  <ID>;
<AG decl *>           =  ;
<AG decl *>           =  <AG decl +>;
<AG decl +>           =  <AG decl +>
                         <AG decl>;
<AG decl +>           =  <AG decl>;
<AG decl>             =  <decl>;
<AG decl>             =  <attribute part>;
<semantics>           =  <production groups>;
<semantics>           =  <attribute phases>;
<production groups>   =  <production groups>
                         <production group>;
<production groups>   =  ;
<attribute phases>    =  <attribute phases>
                         <attribute phase>;
<attribute phases>    =  <attribute phase>;
<attribute phase>     =  phase <ID> is
                            <decl *>
                            <production groups>
                         end phase ";";

* Level 1 declarations:  section 11.2
<decl *>              =  ;
<decl *>              =  <decl +>;
<decl +>              =  <decl +>
```

```
                         <decl>;
<decl +>             =  <decl>;
<decl>               =  <const part>;
<decl>               =  <value part>;
<decl>               =  <type part>;
<decl>               =  <function def>;
<decl>               =  <exception part>;
<decl>               =  <import clause>;

* Import clauses for non-parameterized modules:  section 11.4.1
<import clause>      =  from <module id> import all ";";
<import clause>      =  from <module id> import <import id +> ";";
<import clause>      =  import <module id> ";";
<module id>          =  %ID;
<import id +>        =  <import id +> "," <import id>;
<import id +>        =  <import id>;
<import id>          =  <function id>;
* <id> designates a previously declared identifier
<id>                 =  <ID>;
<id>                 =  <ID> "#" <function id>;

* Import clauses for parameterized modules:  section 11.4.2
<import clause>      =  from <module id> "(" <DCM actual parm +> ")"
                            import all ";";
<import clause>      =  from <module id> "(" <DCM actual parm +> ")"
                            import <import id +> ";";
<import clause>      =  import <ID> "="
                         <module id> "(" <DCM actual parm +> ")" ";";
<DCM actual parm +>  =  <DCM actual parm +> "," <DCM actual parm>;
<DCM actual parm +>  =  <DCM actual parm>;
<DCM actual parm>    =  <id>;
<DCM actual parm>    =  <operator>;

* Constant expression:  section 9.11
<constant expr>      =  <expression>;

* Constant declarations:  section 9.11
<const part>         =  const
                            <const decls>;
<const decls>        =  <const decls>
                         <const decl>;
<const decls>        =  <const decl>;
<const decl>         =  <ID> ":=" <constant expr> ";";

* Value declarations:  section 9.11
<value part>         =  value
```

```
                            <value decls>;
<value decls>         =  <value decls>
                         <value decl>;
<value decls>         =  <value decl>;
<value decl>          =  <ID> <type constraint> ":=" <expression> ";";

* Type declarations:  section 7.2
<type part>           =  type
                            <type decls>;
<type decls>          =  <type decls>
                         <type decl>;
<type decls>          =  <type decl>;
<type decl>           =  <simple type decl>;
<type decl>           =  <recursive types>;
<simple type decl>    =  <ID> "=" <type> ";";
<type>                =  <type id>;
<type id>             =  <id>;
<type id>             =  %TVAR;
<type>                =  <enumerated type>;
<type>                =  <subrange>;
<type>                =  <record type>;
<type>                =  <union type>;
<type>                =  <set type>;
<type>                =  <list type>;
<recursive types>     =  <recursive types>
                         and
                            <simple type decl>;
<recursive types>     =     <simple type decl>
                         and
                            <simple type decl>;

* Enumerated types:  section 7.4
<enumerated type>     =  "(" <identifier +> ")";
<identifier +>        =  <identifier +> "," <ID>;
<identifier +>        =  <ID>;

* Subrange types:  section 7.6
<subrange>            =  range <constant expr> ".." <constant expr>;

* Record types:  section 7.7
<record type>         =  record
                            <fields>
                         end record ;
<fields>              =  <fields>
                         <field>;
<fields>              =  <field>;
```

```
<field>                = <identifier +> ":" <type id> ";";

* Union types:  section 7.8
<union type>           = union
                            <union-types>
                         end union ;
<union-types>          = <union-types>
                         <union-type>;
<union-types>          = <union-type>;
<union-type>           = <type id> ";";

* Set types:  section 7.9
<set type>             = set of <scalar type>;
<scalar type>          = <type id>;

* List types:  section 7.10
<list type>            = list of <type id>;

* Function definitions:  section 8.1
<function def>         = <simple function def>;
<function def>         = <recursive functions>;
<recursive functions> =     <recursive functions>
                         and
                            <simple function def>;
<recursive functions> =     <simple function def>
                         and
                            <simple function def>;
<simple function def> = <function header> is
                         <function body> ";";
<function header>      = <inline> function <function id>
                            <formal parameters> <type constraint>
                            <exc +>;
<function header>      = <inline> function <function id>
                            <formal parameters> <type constraint>;
<inline>               = inline ;
<inline>               = ;
<function id>          = %ID ;
<function id>          = <operator>;
<operator>             = "+";
<operator>             = "-";
<operator>             = "*";
<operator>             = "/";
<operator>             = "%";
<operator>             = "=";
<operator>             = "!=";
<operator>             = "~";
```

```
<operator>            =  "!~";
<operator>            =  "<";
<operator>            =  ">";
<operator>            =  "<=";
<operator>            =  ">=";
<operator>            =  "&";
<operator>            =  "|";
<operator>            =  "!";
<formal parameters>   =  "(" <param *> ")";
<param *>             =  ;
<param *>             =  <param +>;
<param +>             =  <param +> ";" <param group>;
<param +>             =  <param group>;
<param group>         =  <identifier +> ":" <type expr>;
<param group>         =  <ID>;
<type constraint>     =  ;
<type constraint>     =  ":" <type expr>;
<type expr>           =  <type id>;
<type expr>           =  list of <type expr>;
<type expr>           =  set of <type expr>;
<function body>       =  <top expression>
                         end function ;
<function body>       =  declare
                            <decl +>
                         in
                            <top expression>
                         end function ;
<function body>       =  external ;
<function body>       =  external <ID>;
<top expression>      =  <expression>;
<exc +>               =  raise <exc id>;
<exc +>               =  <exc +> "," <exc id>;
<exc id>              =  <id>;

* Exception declarations:  section 10.2
<exception part>      =  exception
                            <exception decls>;
<exception decls>     =  <exception decls>
                         <exception decl>;
<exception decls>     =  <exception decl>;
<exception decl>      =  <identifier +> <formal parameters> ";";

* Attributes declarations:  section 13.2.1
<attribute part>      =  attribute
                            <attribute decls>;
<attribute decls>     =  <attribute decls>
```

```
                           <attribute decl>;
<attribute decls>      =   <attribute decl>;
<attribute decl>       =   <attribute kind> %ATTR "(" <phyla*> ")"
                                       <type constraint> ";";
<attribute decl>       =   compound %ATTR and %ATTR "(" <phyla*> ")"
                                    <type constraint> ";";
<attribute kind>       =   synthesized ;
<attribute kind>       =   inherited ;
<attribute kind>       =   global ;
<attribute kind>       =   ;
<phyla*>               =   ;
<phyla*>               =   <phyla+>;
<phyla+>               =   <phyla+> "," <phylum>;
<phyla+>               =   <phylum>;
<phylum>               =   %ID;

* Production groups and productions:  section 13.4
<production group>     =   where <production> use
                              <sem rules>
                           end where ";";
<production group>     =   where <production>
                           declare
                              <decl +>
                           use
                              <sem rules>
                           end where ";";
<production>           =   <ID>>"" <phylum> "+";
<production>           =   <ID>>"" <phylum> "*";
<production>           =   <ID>>"" <phyla *>;
<phyla *>              =   ;
<phyla *>              =   <phyla +>;
<phyla +>              =   <phyla +> <phylum>;
<phyla +>              =   <phylum>;
<sem rules>            =   <sem rules>
                           <sem rule>;
<sem rules>            =   ;

* Semantic rules:  section 13.4
<sem rule>             =   <attrib occ> ":=" <expression> ";";

* Attribute occurrences:  sections 13.4,  13.5 and 13.6
<attrib occ>           =   <attrib name>;
<attrib occ>           =   <attrib name> "(" <symbol> ")";
<attrib occ>           =   <attrib name> "[" <up symbol +> "]";
<attrib name>          =   %ATTR;
<attrib name>          =   $scx ;
```

```
<attrib name>         =  $arity ;
<attrib name>         =  $position ;
<symbol>              =  <ID>;
<symbol>              =  <ID> "." <qualifier>;
<up symbol +>         =  <up symbol +> "," <up symbol>;
<up symbol +>         =  <up symbol>;
<up symbol>           =  <ID>;
<qualifier>           =  <INT>;
<qualifier>           =  left ;
<qualifier>           =  right ;
<qualifier>           =  first ;
<qualifier>           =  last ;

* Basic expressions:  section 9.1 and section 13.4
<expression>          =  <INT>;
<expression>          =  %REAL;
<expression>          =  %CHAR;
<expression>          =  %STRING;
<expression>          =  <id>;
<expression>          =  <set constructor>;
<expression>          =  <list constructor>;
<expression>          =  <function call>;
<expression>          =  <field access>;
<expression>          =  "(" <expression> ")";
<field access>        =  <expression> "." <ID>;
<expression>          =  <attrib occ>;

* Function calls:  section 8.2
<function call>       =  <id> "(" <actual *> ")";
<actual *>            =  ;
<actual *>            =  <actual +>;
<actual +>            =  <actual +> "," <actual>;
<actual +>            =  <actual>;
<actual>              =  <expression>;

* Set constructors:  section 7.11.5
<set constructor>     =  <set type name> "[|" <set elt *> "|]";
<set type name>       =  <type id>;
<set elt *>           =  ;
<set elt *>           =  <set elt +>;
<set elt +>           =  <set elt +> "," <set elt>;
<set elt +>           =  <set elt>;
<set elt>             =  <expression>;
<set elt>             =  <expression> ".." <expression>;

* List constructors:  section 7.11.6
```

```
<list constructor>    =  <list type name> "[" <actual *> "]";
<list type name>      =  <type id>;

* Operators:  section 9.2
<expression>          =  <expression> "+" <expression>;
<expression>          =  <expression> "-" <expression>;
<expression>          =  <expression> "*" <expression>;
<expression>          =  <expression> "/" <expression>;
<expression>          =  <expression> "%" <expression>;
<expression>          =  <expression> "=" <expression>;
<expression>          =  <expression> "!=" <expression>;
<expression>          =  <expression> "~" <expression>;
<expression>          =  <expression> "!~" <expression>;
<expression>          =  <expression> "<" <expression>;
<expression>          =  <expression> ">" <expression>;
<expression>          =  <expression> "<=" <expression>;
<expression>          =  <expression> ">=" <expression>;
<expression>          =  <expression> "&" <expression>;
<expression>          =  <expression> "|" <expression>;
<expression>          =  "-" <expression>;
<expression>          =  "+" <expression>;
<expression>          =  "!" <expression>;

* Non-strict boolean operators:  section 9.3
<expression>          =  <expression> and <expression>;
<expression>          =  <expression> or <expression>;

* Let expressions:  section 9.4
<expression>          =  <let>;
<let>                 =  let <let clauses>
                         in <expression>;
<let clauses>         =  <let clauses> ","
                         <let clause>;
<let clauses>         =  <let clause>;
<let clause>          =  <ID> <type constraint> ":=" <expression>;

* Conditional expressions:  section 9.5
<expression>          =  <conditional>;
<conditional>         =  if <top expression> then
                            <expression>
                         <elsif part>
                         else
                            <expression>
                         end if ;
<elsif part>          =  <elsif part>
                         elsif <top expression> then
```

```
                              <expression>;
<elsif part>          =  ;

* Selection expressions:  section 9.6.
<expression>          =  <case>;
<case>                =  case <top expression> match
                              <case clauses>
                         end case ;
<case clauses>        =  <case clauses>
                         <case clause>;
<case clauses>        =  <case clause>;
<case clause>         =  <pattern> ":" <expression> ";";
<pattern>             =  <named pattern>;
<pattern>             =  <unnamed pattern>;
<pattern>             =  <universal pattern>;
<named pattern>       =  <ID> ":=" <unnamed pattern>;
<unnamed pattern>     =  <id> "(" <pattern *> ")";
<unnamed pattern>     =  <list type name> "[" <pattern *> "]";
<unnamed pattern>     =  undef ;
<unnamed pattern>     =  undef ":" <type expr>;
<unnamed pattern>     =  "[" <case constant +> "]";
* it was necessary to expand <type constraint> here to solve (with
*                     a screwdriver!)  the shift/reduce conflict on ":"
<universal pattern>   =  <ID>;
<universal pattern>   =  <ID> ":" <type expr>;
<universal pattern>   =  other ;
<pattern *>           =  ;
<pattern *>           =  <pattern +>;
<pattern +>           =  <pattern +> "," <pattern>;
<pattern +>           =  <pattern>;
<case constant +>     =  <case constant +> "," <case constant>;
<case constant +>     =  <case constant>;
<case constant>       =  <constant expr>;
<case constant>       =  <constant expr> ".." <constant expr>;

* Range tests:  section 9.7
<expression>          =  <range test>;
<range test>          =  test <expression> type <type id>;
<range test>          =  test <expression>
                         range <expression> ".." <expression>;

* Error messages:  section 9.8
<expression>          =  <message>;
<message>             =  message <expression>
                              position <position>
                              severity <expression>
```

```
                          value <expression>;
<position>           =  <symbol>;
<position>           =  value <expression>;

* Letrec expressions:  section 9.9.1
<expression>         =  <letrec>;
<letrec>             =  letrec <let clauses>
                        in <expression>;

* undef/define expressions:  section 9.9.2
<expression>         =  <undef>;
<expression>         =  <define>;
<undef>              =  undef <type constraint>;
<define>             =  define <variable> ":=" <expression>;
<variable>           =  <expression>;

* List traversal expressions:  section 9.10
<expression>         =  <map list>;
<map list>           =  map list <expression> <map direction>
                           <map id> value <expression>
                        end map ;
<map direction>      =  left ;
<map direction>      =  right ;
<map id>             =  <id>;
<map id>             =  <operator>;

* Exception raising:  section 10.3.1
<expression>         =  <exc raise>;
<exc raise>          =  raise <id> "(" <actual *> ")";

* Exception catching:  section 10.3.2
<expression>         =  <exc catch>;
<exc catch>          =  <expression>
                        catch
                           <exception handler +>
                           <default handler>
                        end catch ;
<exception handler +> =  <exception handler +>
                        <exception handler>;
<exception handler +> =  <exception handler>;
<exception handler>  =  <id> ":"
                           <expression> ";";
<default handler>    =  other ":"
                           <expression> ";";
<default handler>    =  ;
```

```
* Expressions for semantic rules of list productions
* section 13.6
<expression>          =  <case position>;
<case position>       =  case position is
                            <first son clause>
                            <last son clause>
                            <other son clause>
                         end case ;
<case position>       =  case position is
                            <last son clause>
                            <first son clause>
                            <other son clause>
                         end case ;
<case position>       =  case position is
                            <first son clause>
                            <other son clause>
                         end case ;
<case position>       =  case position is
                            <last son clause>
                            <other son clause>
                         end case ;
<first son clause>    =  first ":" <expression> ";";
<last son clause>     =  last ":" <expression> ";";
<other son clause>    =  other ":" <expression> ";";
<expression>          =  <case arity>;
<case arity>          =  case arity is
                            <fixed arity part>
                            <other arity clause>
                         end case ;
<fixed arity part>    =  <fixed arity part>
                         <fixed arity clause>;
<fixed arity part>    =  <fixed arity clause>;
<fixed arity clause>  =  <INT> ":" <expression> ";";
<other arity clause>  =  other ":" <expression> ";";

* Map expressions:  section 13.7.1
<expression>          =  <map>;
<map>                 =  map <map direction> <map id>
                            <opt-other>
                            value <expression>
                            <opt-other>
                         end map ;
<opt-other>           =  ;
<opt-other>           =  other <actual +>;
```

```
* Tree decoration expressions:  section 14.3
<expression>          =  <decoration expr>;
<decoration expr>     =  <expression>
                            with <decoration clauses>
                         end with ;
<decoration clauses>  =  <decoration clauses> ","
                         <decoration clause>;
<decoration clauses>  =  <decoration clause>;
<decoration clause>   =  <attrib name> ":=" <expression>;
```

Since this grammar is ambiguous, it must be completed by a specification of the relative priorities of the operators:

```
* -----------------------------------------------------
* olga.prio
* -----------------------------------------------------
* Specification of the priorities and associativity rules
* of the OLGA operators
* See "the OLGA Reference Manual" section 9.12
* -----------------------------------------------------

%nonassoc catch
%right message let define
%nonassoc range
%right and or
%left "&" |
%nonassoc = != "~" "!~" "<" > "<=" >=
%left + -
%left * / %
%right !
%left .
%nonassoc with
```

```
<decoration expr >    =  <expression>
                            with <decoration clauses>
                            end with ;        %prec with
<field access>        =  <expression> "." <ID>%prec .
<expression>          =  "-" <expression>; %prec !
<expression>          =  "+" <expression>; %prec !
<let>                 =  let <let clauses>
                         in <expression>;     %prec let
```

```
<letrec>           =  letrec <let clauses>
                      in <expression>;      %prec let
<range test>       =  test <expression>
                      type <type id>;       %prec range
<range test>       =  test <expression>
                      range <expression> ".." <expression>;
                                            %prec range
<message>          =  message <expression>
                         position <position>
                         severity <expression>
                         value <expression>;  %prec message
<define>           =  define <variable> ":=" <expression>;
                                            %prec define
<exc catch>        =  <expression>
                         catch
                            <exception handler +>
                            <default handler>
                         end catch;         %prec catch
```

Appendix C

Summary of Predefined OLGA Objects

This appendix lists all the predefined OLGA objects (types, functions, etc.) in the form of a declaration module named OLGA-*predef*. This module, more precisely the DCM, since the DFM does not exist in OLGA, is actually available in the distribution.

```
declaration module OLGA-predef is

{predefined types and their constants/values:  section 3.3 }
   type
      int;
      real;
      bool = (false, true);
      char;
      string;
      token;
   value
      error-token:  token;
   type
      source-index;
   value
      beginning-of-text, end-of-text:  source-index;
   type
      void;
   const
      null:  void;

{predefined exceptions:  section 6.2 }
   exception
      ARITY-ERROR, CONVERSION-ERROR, DEFINED, EMPTY-LIST,
      RANGE-ERROR, TREE-ERROR, UNDEFINED, ZERO-DIVIDE;
```

```
{conversion functions:  section 3.11 }
   {scalars from/to integers }
function int (x:  bool):  int raise UNDEFINED;
function int (x:  char):  int raise UNDEFINED;
   function bool (x:  int):  bool raise RANGE-ERROR, UNDEFINED;
   function char (x:  int):  char raise RANGE-ERROR, UNDEFINED;
   {integers from/to reals }
function real (i:  int):  real raise UNDEFINED;
function round (r:  real):  int raise RANGE-ERROR, UNDEFINED;
function trunc (r:  real):  int raise RANGE-ERROR, UNDEFINED;
   {integers from/to strings }
function string (i:  int):  string raise UNDEFINED;
function string (i:  int; base:  range 2..16):  string
                              raise UNDEFINED;
function int (s:  string):  int
         raise CONVERSION-ERROR, RANGE-ERROR, UNDEFINED;
function int (s:  string; base:  range 2..16):  int
         raise CONVERSION-ERROR, RANGE-ERROR, UNDEFINED;
   {reals from/to strings }
function string (r:  real):  string raise UNDEFINED;
function string (r:  real; total-width:  int):  string
                              raise UNDEFINED;
function string (r:  real; total-width, frac-width:  int)
                       :  string raise UNDEFINED;
function real (s:  string):  real
         raise CONVERSION-ERROR, RANGE-ERROR, UNDEFINED;
   {booleans from/to strings }
function string (x:  bool):  string raise UNDEFINED;
function bool (s:  string):  bool
               raise CONVERSION-ERROR, UNDEFINED;
   {characters from/to strings }
function string (c:  char):  string raise UNDEFINED;
function char (s:  string; x:  integer):  char
                  raise RANGE-ERROR, UNDEFINED
               {not really a conversion function!  };
   {tokens from/to strings }
function string (t:  token):  string raise UNDEFINED;
function token (s:  string; check:  bool):  token
                              raise UNDEFINED;
function token (s:  string):  token raise UNDEFINED;

{other functions (mainly operators):  section 4.3.1 }

   {(in)equality and (non-)identity polymorphic predicates }
function = (x, y:  @T): bool {equality };
function != (x, y:  @T): bool {inequality };
```

```
function sim (x, y: @T): bool {identity test };
function !sim (x, y: @T): bool {non-identity test };

   {dealing with integers: }
   {arithmetic }
function + (x, y: int): int raise RANGE-ERROR, UNDEFINED
                          {addition };
function + (x: int): int raise UNDEFINED {identity };
function - (x, y: int): int raise RANGE-ERROR, UNDEFINED
                          {subtraction };
function - (x: int): int raise UNDEFINED {opposite };
function * (x, y: int): int raise RANGE-ERROR, UNDEFINED
                       {multiplication };
function / (x, y: int): int raise ZERO-DIVIDE, UNDEFINED
                       {Euclidean division };
function % (x, y: int): int raise ZERO-DIVIDE, UNDEFINED
                       {remainder/modulus };
   {logical }
function & (x, y: int): int raise UNDEFINED
                          {bitwise AND };
function | (x, y: int): int raise UNDEFINED {bitwise OR };
function! (x: int): int raise UNDEFINED {bitwise NOT };
   {comparisons }
function < (x, y: int): bool raise UNDEFINED;
function > (x, y: int): bool raise UNDEFINED;
function <= (x, y: int): bool raise UNDEFINED;
function >= (x, y: int): bool raise UNDEFINED;
   {misc. }
function pred (x: int): int
                 raise RANGE-ERROR, UNDEFINED;
function succ (x: int): int
                 raise RANGE-ERROR, UNDEFINED;

   {dealing with reals }
   {arithmetic }
function + (x, y: real): real raise RANGE-ERROR, UNDEFINED
                          {addition };
function + (x: real): real raise UNDEFINED {identity };
function - (x, y: real): real raise RANGE-ERROR, UNDEFINED
                          {subtraction };
function - (x: real): real raise UNDEFINED {opposite };
function * (x, y: real): real raise RANGE-ERROR, UNDEFINED
                       {multiplication };
function / (x, y: real): real raise ZERO-DIVIDE, UNDEFINED
                          {division };
   {comparisons }
```

```
function < (x, y: real): bool raise UNDEFINED;
function > (x, y: real): bool raise UNDEFINED;
function <= (x, y: real): bool raise UNDEFINED;
function >= (x, y: real): bool raise UNDEFINED;

   {dealing with booleans }
   {comparisons }
function < (x, y: bool): bool raise UNDEFINED;
function > (x, y: bool): bool raise UNDEFINED;
function <= (x, y: bool): bool raise UNDEFINED;
function >= (x, y: bool): bool raise UNDEFINED;
   {logical }
function & (x, y: bool): bool raise UNDEFINED {AND };
function | (x, y: bool): bool raise UNDEFINED {OR };
function! (x: bool): bool raise UNDEFINED {NOT };
   {misc. }
function pred (x: bool): bool
                  raise RANGE-ERROR, UNDEFINED;
function succ (x: bool): bool
                  raise RANGE-ERROR, UNDEFINED;

   {dealing with characters }
   {comparisons }
function < (x, y: char): bool raise UNDEFINED;
function > (x, y: char): bool raise UNDEFINED;
function <= (x, y: char): bool raise UNDEFINED;
function >= (x, y: char): bool raise UNDEFINED;
   {misc. }
function pred (x: char): char
                  raise RANGE-ERROR, UNDEFINED;
function succ (x: char): char
                  raise RANGE-ERROR, UNDEFINED;

   {dealing with strings }
function + (x, y: string): string raise UNDEFINED
                        {concatenation };
   {comparisons }
function < (x, y: string): bool raise UNDEFINED;
function > (x, y: string): bool raise UNDEFINED;
function <= (x, y: string): bool raise UNDEFINED;
function >= (x, y: string): bool raise UNDEFINED;
   {misc. }
function empty (x: string): bool raise UNDEFINED;
function length (x: string): int raise UNDEFINED
                     {number of characters };
```

```
   {dealing with tokens }
   {comparisons }
function < (x, y: token): bool raise UNDEFINED;
function > (x, y: token): bool raise UNDEFINED;
function >= (x, y: token): bool raise UNDEFINED;
function <= (x, y: token): bool raise UNDEFINED;
   {misc. }
function length (x: token): int raise UNDEFINED
                    {number of characters };

   {dealing with source-indexes }
function file (x: source-index): string raise UNDEFINED
                          {file name };
function line (x: source-index): int raise UNDEFINED
                          {line number };
function column (x: source-index): int raise UNDEFINED
                       {column number };

   {dealing with sets }
   {set-theoretical operations }
function + (x, y: set of @T): set of @T raise UNDEFINED
                          {union };
function - (x, y: set of @T): set of @T raise UNDEFINED
                          {difference };
function - (x: set of @T): set of @T raise UNDEFINED
                          {complement };
function * (x, y: set of @T): set of @T raise UNDEFINED
                          {intersection };
function * (x: @T; y: set of @T): bool raise UNDEFINED
                       {membership test };
   {comparisons }
function < (x, y: set of @T): bool raise UNDEFINED;
function > (x, y: set of @T): bool raise UNDEFINED;
function <= (x, y: set of @T): bool raise UNDEFINED;
function >= (x, y: set of @T): bool raise UNDEFINED;
   {misc. }
function empty (x: set of @T): bool raise UNDEFINED;
function card (x: set of @T): int raise UNDEFINED
                       {number of elements };

   {dealing with lists }
   {construction }
   function + (x, y: list of @T): list of @T
               raise UNDEFINED {concatenation };
function cons (x: @T; y: list of @T): list of @T
               raise UNDEFINED {front addition };
```

```
function tcons (x: list of @T; y: @T): list of @T
               raise UNDEFINED {back addition };
function ncons (x: @T): list of @T {one element cons. };
   {access }
function head (x: list of @T): @T
         raise EMPTY-LIST, UNDEFINED {first element };
function tail (x: list of @T): list of @T
         raise EMPTY-LIST, UNDEFINED {all but first };
   {misc. }
function empty (x: list of @T): bool raise UNDEFINED;
function length (x: list of @T): int raise UNDEFINED
                        {number of elements };

end module;
```

Appendix D

ATC Lexicon and Concrete Syntax Summary

This chapter summarizes the specification of the concrete representation of the ATC tree construction language which is presented in chapter 22 of Part V.

D.1 Lexical Description

```
-- Lexical description of "atc" for SYNTAX scanner generator "lecl"
```

Classes

```
   LETTER= "a".."z" + "A".."Z";
   DIGIT    = "0".."9";
   OCTAL    = "0".."7";
   TRULY_PRINTABLE= "..#176 - "^\'\"\\";
            -- printable ASCII characters except those
            -- participating in escape sequences
   CONTROL_CHAR= "A".."_";
            -- printable characters which correspond to
            -- control characters with the "↑" notation
            -- except @ (↑@ == NUL character, forbidden)
   BLANKS         = SP + HT + VT + FF + LF + NL + CR;
      -- the above identifiers are predefined in the
      -- lexical metalanguage of SYNTAX
   PRINT    = ANY - #000..#040 - #177;
   T_HEADER       = PRINT - "<#;\"~";
   NT_BODY        = PRINT + SP - ">";
```

Abbreviations

```
   STARLINE    = "*"&Is_First_Col {↑EOL} EOL;
   &Not_In_First_Col  = ^ &Is_First_Col;
   MEANINGFUL  = "{" @1 {^"{}<"&True | "{"&True @1 | "}"&1 @2 |
            "<"&Not_In_First_Col}*
```

```
          ("}"&2 @2 | @3);
      -- difference with "OLGA.lecl" is in processing "<" in
      -- column one which must be considered as EndOfFile
      -- (because "bnf" says so!)
      -- initialisation:  comment-level = 0
      -- &1:  true if comment-level > 1
      -- &2 = not &1
      -- @1:  increase comment-level
      -- @2:  decrease comment-level
      -- @3:  found beginning of rule or physical EOF inside
      -- comment:  complain as in OLGA and, if beginning of
      -- rule, add message on < so that user doesn't get lost
   ID    = LETTER {["_-"] (LETTER | DIGIT)}*;
   PRINTABLE   = TRULY_PRINTABLE
         | -"^" CONTROL_CHAR @4
         | -"\\" (-"b"&True @Put (BS) |
               -"t"&True @Put (HT) |
               -"n"&True @Put (NL) |
               -"v"&True @Put (VT) |
               -"f"&True @Put (FF) |
               -"r"&True @Put (CR) |
               -"s"&True @Put (SP) |
               TRULY_PRINTABLE |
               "^\'\"\\" |
         @Mark OCTAL&True [OCTAL&True[OCTAL&True]] @6);
      -- @4:  convert a control character from "↑" notation
      -- @6:  convert a character from its octal value and
      -- complain if NUL
```

Tokens

```
   Comments       = -{BLANKS | MEANINGFUL | STARLINE}+;
   %ID            = ID;
   %INT           = {DIGIT}+;
   %STRING        = -"\"" {PRINTABLE} -"\"";
   %Non_Terminal  = "<" {NT_BODY} ">";
   %Generic_Terminal
            = "%" LETTER {["_"] (LETTER | DIGIT)};
   %ATTR          = "$" [("sh" | "SH"@lower_case) "."&True] ID;

   Eof            = "<"&Is_First_Col @Release @Reset(1);
```

D.2 Syntactic Description of ATC

This summarizes the concrete syntax of ATC. The order of the sections is not respected, rather a "top-down" approach is used.

```
* ------------------------------------------------------
* atc.bnf
* ------------------------------------------------------
* Concrete syntax of the ATC language
* ------------------------------------------------------

<concrete grammar>    =  <header>
                         <definition*>
                         <production group+>;

<header>              =  import grammar %ID ";";

<definition*>         =  <definition*>
                         <definition>;
<definition*>         =  ;
<definition>          =  <phylum def>;
<definition>          =  <phylum extension>;
<definition>          =  <operator def>;
<definition>          =  <function def>;

<phylum def>          =  %ID "=" <operator +> ";";
<phylum extension>    =  %ID "+=" <operator +> ";";
<operator +>          =  <operator +> <operator>;
<operator +>          =  <operator>;
<operator>            =  %ID;

<operator def>        =  %ID "->" <phylum> "+" ";";
<operator def>        =  %ID "->" <phylum> "*" ";";
<operator def>        =  %ID "->" <phyla*> ";";
<phyla*>              =  ;
<phyla*>              =  <phyla+>;
<phyla+>              =  <phyla+> <phylum>;
<phyla+>              =  <phylum>;
<phylum>              =  %ID ;

<function def>        =  function %ID "(" <formals *> ")"
                            ":" <phylum> is
                            <expression>
                         end function ";";
<formals *>           =  ;
<formals *>           =  <formals +>;
<formals +>           =  <formals +> ";" <param group>;
<formals +>           =  <param group>;
<param group>         =  <id +> ":" <phylum>;
<id +>                =  <id +> "," %ID;
```

```
<id +>                =  %ID ;

<production group+>   =  <production group+>
                         <production group>;
<production group+>   =  <production group>;
<production group>    =  <rule>
                            <tree spec>;

* The following rules are borrowed from SYNTAX (bnf.bnf)
<rule>                =  <vocabulary_list> ";";
<vocabulary_list>     =  <vocabulary_list> <vocabulary>;
<vocabulary_list>     =  %LH_Non_Terminal "=";
<vocabulary>          =  %Non_Terminal;
<vocabulary>          =  %Terminal;
<vocabulary>          =  %Generic_Terminal;
* End of rules borrowed from SYNTAX

<tree spec>           =  ;
<tree spec>           =  <operator>;
<tree spec>           =  <expression>;

<expression>          =  <primary expr>;
<primary expr>        =  %ID;
<primary expr>        =  <symbol nt>;
<symbol nt>           =  <non-terminal>;
<symbol nt>           =  <non-terminal> "." %INT;
<non-terminal>        =  %Non_Terminal;

<expression>          =  <tree>;
<tree>                =  <operator> "(" <operand*> ")";
<operand*>            =  ;
<operand*>            =  <expression+>;
<expression+>         =  <expression+> "," <expression>;
<expression+>         =  <expression>;

<expression>          =  <case match>;
<case match>          =  case <primary expr> match
                            <match clause+>
                         end case ;
<match clause+>       =  <match clause+>
                         <match clause>;
<match clause+>       =  <match clause>;
<match clause>        =  <pattern> ":" <expression> ";";
<pattern>             =  <operator> "(" <patterns *> ")";
<pattern>             =  %ID;
<pattern>             =  %ID ":" %ID;
```

```
<pattern>             =  other ;
<patterns *>          =  ;
<patterns *>          =  <patterns +>;
<patterns +>          =  <patterns +> "," <pattern>;
<patterns +>          =  <pattern>;

<expression>          =  <unconditional match>;
<unconditional match> =  match <match-in clause+>
                         in <expression>;
<match-in clause+>    =  <match-in clause+> ","
                         <match-in clause>;
<match-in clause+>    =  <match-in clause>;
<match-in clause>     =  <primary expr> with <pattern>;

<expression>          =  <conditional>;
<conditional>         =  if <relational expr> then <expression>
                         <elsif part>
                         else <expression>
                         end if ;
<elsif part>          =  <elsif part>
                         elsif <relational expr> then <expression>;
<elsif part>          =  ;
<relational expr>     =  <expression> "=" <expression>;
<relational expr>     =  <expression> "!=" <expression>;
<relational expr>     =  <primary expr> match <pattern>;
<relational expr>     =  <relational expr> and <relational expr>;

<expression>          =  <message>;
<message>             =  message <position or string>
                            position <position or string>
                            severity %INT
                            value <expression>;
<position or string>  =  %ID;
<position or string>  =  <primary symbol>;
<position or string>  =  <primary symbol> "." %INT;
<position or string>  =  <position or string> "+"
                               <position or string>;
<position or string>  =  %ATTR "(" %ID ")";
<primary symbol>      =  <non-terminal>;
<primary symbol>      =  %Generic_Terminal;
<primary symbol>      =  %STRING;

<expression>          =  <abbreviation>;
<abbreviation>        =  let <let clause+>
                         in <expression>;
<let clause+>         =  <let clause+> ","
```

```
                           <let clause>;
<let clause+>          =   <let clause>;
<let clause>           =   %ID ":=" <expression>;

<expression>           =   <tree with>;
<tree with>            =   <operator> "(" <operand*> ")" <with>;
<tree with>            =   <primary expr> <with>;
<with>                 =   with <attribute +>
                           end with ;
<attribute +>          =   <attribute +> ","
                           <attribute>;
<attribute +>          =   <attribute>;
<attribute>            =   <attr name> ":=" <position or string>;
<attr name>            =   %ATTR;
<attr name>            =   $scx ;

<map direction>        =   left ;
<map direction>        =   right ;
<operator with>        =   <operator>;
<operator with>        =   <operator> <with>;
<opt-other>            =   ;
<opt-other>            =   other <expression+>;
```

Since this grammar is ambiguous, it must be completed by a specification of the relative priorities of the operators:

```
* -----------------------------------------------------
* atc.prio
* -----------------------------------------------------
* Specification of the priorities and associativity rules
* of the ATC operators
* -----------------------------------------------------

%left and
%left +
```

Appendix E

PPAT Lexicon and Concrete Syntax Summary

This chapter summarizes the specification of the concrete representation of the PPAT pretty–printer language for attributed–trees which is presented in chapter 20 of Part IV.

E.1 Lexical Description

```
-- Lexical description of "Ppat" for SYNTAX scanner generator "lecl"

Classes
   LETTER       = "a".."z" + "A".."Z";
   DIGIT        = "0".."9";
   OCTAL        = "0".."7";
   TRULY_PRINTABLE= " "..#176 - "^\'\"\\";
                -- printable ASCII characters except those
                -- participating in escape sequences
   CONTROL_CHAR= "A".."_";
                -- printable characters which correspond to
                -- control characters with the "↑" notation
                -- except @ (↑@ == NUL character, forbidden)
   BLANKS       = SP + HT + VT + FF + LF + NL + CR;
      -- the above identifiers are predefined in the
      -- lexical metalanguage of SYNTAX

Abbreviations
   MEANINGFUL  = "{" @1 {^"{}"&True | "{"&True @1 |"}"&1 @2}*
           ("}"&2 @2 | @7) ;
      -- initialisation:  comment-level = 0
      -- &1:  true if comment-level > 1
      -- &2 = not &1
      -- @1:  increase comment-level
      -- @2:  decrease comment-level
```

```
   -- @7:  (on EOF only) complain if comment-level != 0
ID          = LETTER {["_-"] (LETTER | DIGIT)}*;
SIGN        = [-"+" | "-"] ;
INT         = {DIGIT}+ ;
PRINTABLE   = TRULY_PRINTABLE
            | -"^" CONTROL_CHAR @4
            | -"\\" (-"b"&True @Put (BS) |
            -"t"&True @Put (HT) |
            -"n"&True @Put (NL) |
            -"v"&True @Put (VT) |
            -"f"&True @Put (FF) |
            -"r"&True @Put (CR) |
            -"s"&True @Put (SP) |
            TRULY_PRINTABLE |
            "^\'\"\\" |
            @Mark OCTAL&True [OCTAL&True[OCTAL&True]] @6);
            -- @4:  convert a control character from "↑" notation
            -- @6:  convert a character from its octal value and
            -- complain if NUL
```

Tokens

```
Comments    = -{BLANKS | MEANINGFUL }+;
               Context All But Comments ;
%ID         = ID;
               Context All But %ID, %INT, %REAL, "->", "-" ;
%INT        = INT ;
               -- Unbounded Context All but %INT;
%ATTR       = "$" [("sh" | "SH" @Lower_Case) "."&True] ID ;
               Context All But %ID, "-" ;
%REAL       = INT "." DIGIT&True {DIGIT}* [("e" | "E" @Lower_Case) SIGN INT] ;
               Context All But %ID ;
%STRING     = -"\"" {PRINTABLE} -"\"";
">"         = -">" ;
               Context All But "=" ;
```

E.2 Syntactic Description of Ppat

This summarizes the concrete syntax of Ppat. The order of the sections is not respected, rather a “top-down” approach is used.

```
* ------------------------------------------------------
* Ppat.bnf
* ------------------------------------------------------
* Concrete syntax of the PPAT language
* ------------------------------------------------------

** ppat program ***
<ppat prog>          =  <header>
                        <decls>
                        <rules>
                        end ppat ";" ;

** header :  name of program and type of trees to be handled ***

<header>             =  ppat <ID> "(" <ID> ")" ";" ;

** declarations ***

<decls>              =  <decls>
                        <decls sep> ;
<decls>              =  <decls>
                        <decls cxt> ;
<decls>              =  <decls>
                        <import clause> ;
<decls>              =  ;

** importation ***

<import clause>      =  from <ID> import all ";" ;

** separator declarations ***

<decls sep>          =  separator
                        <decl sep +> ;
<decl sep +>         =  <decl sep +>
                        <sep def> ;
<decl sep +>         =  <sep def> ;
<sep def>            =  "<" <SEP ID> ">" "=" <separator> ";" ;
<separator>          =  "<" <ID> <sep param *> ">" ;
<sep param *>        =  <sep param +> ;
<sep param *>        =  ;
<sep param +>        =  <sep param +> "," <sep param> ;
<sep param +>        =  <sep param> ;
<sep param>          =  <INT> ;
```

```
** context declarations ***
<decls cxt>          =  context
                        <decl cxt +> ;
<decl cxt +>         =  <decl cxt +>
                        <cxt def> ;
<decl cxt +>         =  <cxt def> ;
<cxt def>            =  <attrib name> "(" <phyla, *> ")" ":" <ID> ";" ;
<phyla, *>           =  <phyla, +> ;
<phyla, *>           =  ;
<phyla, +>           =  <phyla, +> "," <phylum> ;
<phyla, +>           =  <phylum> ;

** rule specifications ***

<rules>              =  <rule type *>
                        <rule op *> ;

** attribute decompilation specifications ***

<rule type *>        =  <rule type *>
                        <rule type> ;
<rule type *>        =  ;
<rule type>          =  wheretype <ID> pprint
                           <top box type>
                        end where ";" ;
<top box type>       =  <box type> ;
<top box type>       =  ;
<box type>           =  <string boxt> ;
<box type>           =  <compound boxt> ;
<box type>           =  <fieldref boxt> ;
<box type>           =  <pplist boxt> ;
<string boxt>        =  <STRING> ;
<compound boxt>      =  "[" <SEP ID> <boxes type> "]" ;
<boxes type>         =  <boxes type> <local sep> <box type> ;
<boxes type>         =  <box type> ;
<fieldref boxt>      =  <ID> ;
<pplist boxt>        =  pplist <ID> <direction>
                           global <SEP ID>
                           <insert>
                           subbox <box type>
                        end pplist ;

** operator decompilation specifications ***

<rule op *>          =  <rule op *>
```

```
                          <rule op> ;
<rule op *>            =  ;
<rule op>              =  where <production> pprint
                             <top box>
                          end where ";" ;
<production>           =  <ID> "->" <ID> "+" ;
<production>           =  <ID> "->" <ID> "*" ;
<production>           =  <ID> "->" <phyla *> ;
<phyla *>              =  <phyla +> ;
<phyla *>              =  ;
<phyla +>              =  <phyla +> <ID> ;
<phyla +>              =  <ID> ;

<top box>              =  <box>;
<top box>              =  ;
<box>                  =  <string box> ;
<box>                  =  <attrib occ> ;
<box>                  =  <rec call> ;
<box>                  =  <conditional> ;
<box>                  =  <compound> ;
<box>                  =  <case match> ;
<box>                  =  <pplist> ;
<string box>           =  <STRING> ;

** <attrib occ> ***

<attrib occ>           =  <attrib name> ;
<attrib occ>           =  <attrib name> "(" <expr> ")" ;
<attrib occ>           =  <attrib name> "(" <expr> "." <qualifier> ")" ;

** <symbol> ***

<symbol>               =  <ID> ;
<symbol>               =  <ID> "." <qualifier> ;


<qualifier>            =  left ;
<qualifier>            =  right ;
<qualifier>            =  first ;
<qualifier>            =  last ;
<rec call>             =  <symbol> <context list> ;
<context list>         =  "(" <context *> ")" ;
<context list>         =  ;
<context *>            =  <context +> ;
<context +>            =  <context +> "," <context>;
<context +>            =  <context> ;
```

```
<context>             =  <attrib name> ":=" <expr>;
<attrib name>         =  %ATTR ;
<attrib name>         =  $scx ;
<attrib name>         =  $arity ;
<attrib name>         =  $position ;
<conditional>         =  if <expr> then
                            <box>
                         <elsif part>
                         else
                            <box>
                         end if ;
<elsif part>          =  <elsif part>
                         elsif <expr> then
                            <box> ;
<elsif part>          =  ;
<compound>            =  "[" <SEP ID> <box +> "]" ;
<box +>               =  <box +> <local sep> <box> ;
<box +>               =  <box> ;
<case match>          =  case <expr> is
                            <match clauses>
                            <match default>
                         end case ;
<match clauses>       =  <match clauses>
                         <match clause> ;
<match clauses>       =  <match clause> ;
<match clause>        =  <case clauses> ":"
                         <box> ";" ;
<case clauses>        =  <case clauses> "," <case clause> ;
<case clauses>        =  <case clause> ;
<match default>       =  other ":"
                         <box> ";" ;
<case clause>         =  <root pattern> ;
<root pattern>        =  <pattern cons> ;
<root pattern>        =  <ID> ;
<root pattern>        =  <constant manifeste> ;
<root pattern>        =  const <expr> ;
<pattern cons>        =  <ID> "(" <patterns *> ")" ;
<patterns *>          =  <patterns +> ;
<patterns *>          =  ;
<patterns +>          =  <patterns +> "," <pattern> ;
<patterns +>          =  <pattern> ;
<constant manifeste>  =  <INT> ;
<constant manifeste>  =  %REAL ;
<constant manifeste>  =  %CHAR ;
<constant manifeste>  =  %STRING ;
<pattern>             =  <root pattern> ;
```

```
<pplist>            =  pplist <direction>
                          global <SEP ID>
                          <insert>
                          subbox <box>
                       end pplist ;
<direction>         =  left ;
<direction>         =  right ;

<local sep>         =  "<" <SEP ID> ">" ;
<local sep>         =  ;
<insert>            =  insert <STRING> ;
<insert>            =  ;

<expr>              =  <INT> ;
<expr>              =  <REAL> ;
<expr>              =  <CHAR> ;
<expr>              =  <STRING> ;
<expr>              =  <ID> ;
<expr>              =  "(" <expr> ")" ;
<expr>              =  <function call> ;
<expr>              =  <attrib occ> ;
<expr>              =  <expr> "+" <expr> ;
<expr>              =  <expr> "-" <expr> ;
<expr>              =  <expr> "*" <expr> ;
<expr>              =  <expr> "/" <expr> ;
<expr>              =  <expr> "%" <expr> ;
<expr>              =  <expr> "=" <expr> ;
<expr>              =  <expr> "!=" <expr> ;
<expr>              =  <expr> "~" <expr> ;
<expr>              =  <expr> "!~" <expr> ;
<expr>              =  <expr> "<" <expr> ;
<expr>              =  <expr> ">" <expr> ;
<expr>              =  <expr> "<=" <expr> ;
<expr>              =  <expr> ">=" <expr> ;
<expr>              =  <expr> "&" <expr> ;
<expr>              =  <expr> "|" <expr> ;
<expr>              =  <expr> "and" <expr> ;
<expr>              =  <expr> "or" <expr> ;
<expr>              =  "-" <expr> ;
<expr>              =  "+" <expr> ;
<expr>              =  "!" <expr> ;
<function call>     =  <ID> "(" <actual *> ")" ;
<actual *>          =  <actual +> ;
<actual *>          =  ;
<actual +>          =  <actual +> "," <actual> ;
<actual +>          =  <actual> ;
```

```
<actual>              =  <expr> ;
<INT>                 =  %INT ;
<CHAR>                =  %CHAR ;
<STRING>              =  %STRING ;
<ID>                  =  %ID ;
<SEP ID>              =  %ID ;
<REAL>                =  %REAL ;
<phylum>              =  %ID ;
```

Since this grammar is ambiguous, it must be completed by a specification of the relative priorities of the operators:

```
* -------------------------------------------------------
* Ppat.prio
* -------------------------------------------------------
* Specification of the priorities and associativity rules
* of the PPAT operators
* -------------------------------------------------------

%right and or
%left "&" |
%nonassoc = != "~" "!~" "<" > "<=" >=
%left + -
%left * / %
%right !
%left .

<expr> = "-" <expr> ;         %prec !
<expr> = "+" <expr> ;         %prec !
```

Bibliography

[**AM91**] H. Alblas and B. Melichar (eds.): *Attribute Grammars, Applications and Systems*, Prague, vol. 545 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, June 1991.

[**ANS83**] ANSI: *ADA Reference Manual*, American National Standards Institution, Jan. 1983.

[**BD88**] P. Boullier and P. Deschamp: *Le Système SYNTAX—Manuel D'Utilisation Et De Mise En Oeuvre Sous Unix*, INRIA, Rocquencourt, Sep. 1988.

[**BDL87**] P. Boullier, P. Deschamp, and B. Lorho: "The SYNTAX Reference Manual", Tech. rep., INRIA, Rocquencourt, Nov. 1987.

[**BJ87**] P. Boullier and M. Jourdan: "A New Error Repair and Recovery Scheme for Lexical and Syntactic Analysis", *Sci. of Comp. Progr.*, 9(4), Dec. 1987, pp. 271–286.

[**BMS80**] R. Burstall, D. B. MacQueen, and D. Sanella: "Hope: an Experimental Applicative Language", in *1980 LISP Conf.*, pp. 136–143, 1980.

[**Boc76**] G. V. Bochmann: "Semantic Evaluation from Left to Right", *Comm. of the ACM*, 19(2), Feb. 1976, pp. 55–62.

[**Bon86**] J. Bonnet: Rapport de DEA, rapport de, Univ. d'Orléans, Sep. 1986.

[**Bor88**] P. Borras: "Reference Manual & Compiler Implementation", report, SEMA–METRA, January 1988.

[**BRB+81**] B.Liskov, R.Atkinson, T. Bloom, E. Moss, J.C.Schaffert, R. Scheifler, and A. Snyder: *CLU Reference Manual*, vol. 114 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, Berlin, 1981.

[**BSI82**] BSI: *Specification for the Computer Programming Language Pascal*, vol. BS 6192, British Standards Institution, 1982.

[**Car87**] L. Cardelli: "Basic Polymorphic Typechecking", *Sci. of Comp. Progr.*, 8(2), April 1987, pp. 147–172.

[**CDD+86**] J. Chailloux, M. Devin, F. Dupont, J.-M. Hullot, B. Serpette, and J. Vuillemin: "L_Lisp Version 15.2 : le Manuel de Référence", Tech. rep., INRIA, Rocquencourt, May 1986.

[**CHP85**] G. Cousineau, G. Huet, and L. Paulson: *The ML Handbook*, INRIA and University of Cambridge, May 1985.

[**Cou84**] B. Courcelle: “Attribute Grammars: Definitions, Analysis of Dependencies, Proof Methods”, in B. Lorho (ed.): *Methods and Tools for Compiler Construction*, pp. 81–102, Cambridge Univ. Press, 1984.

[**DB85**] P. Deschamp and P. Boullier: “Paradis: Un Système de Paragraphage Dirigé par la Syntaxe”, rapport RR-455, INRIA, Rocquencourt, Nov. 1985.

[**Des80**] P. Deschamp: “Production de Compilateurs à Partir d'une Description Sémantique des Langages de Programmation: Le Système Perluette”, thèse de docteur-ingéieur, INPL, Nancy, Oct. 1980.

[**DGHK+80**] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang: “Programming Environments Based on Structured Editors: the Mentor Experience”, rapport RR-26, INRIA, Rocquencourt, July 1980.

[**DJL88**] P. Deransart, M. Jourdan, and B. Lorho: *Attribute Grammars: Definitions, Systems and Bibliography*, vol. 323 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, Aug. 1988.

[**Eng84**] J. Engelfriet: “Attribute Grammars: Attribute Evaluation Methods”, in B. Lorho (ed.): *Methods and Tools for Compiler Construction*, pp. 103–138, Cambridge Univ. Press, 1984.

[**Far84**] R. Farrow: “Generating a Production Compiler From an Attribute Grammar”, *IEEE Software*, 1(4), Oct. 1984, pp. 77–93.

[**Gau80**] M.-C. Gaudel: “Génération Et Preuve De Compilateurs Basées Sur Une Sémantique Formelle Des Langages De Programmation”, Tech. Rep. thèse d'État, INPL, Nancy, March 1980.

[**GDM78**] M.-C. Gaudel, P. Deschamp, and M. Mazaud: “Semantics of Procedures as an Algebraic Data Type”, rapport 334, —IRIAL—, 1978.

[**GG84**] H. Ganzinger and R. Giegerich: “Attribute Coupled Grammars”, in *ACM SIGPLAN '84 Symp. on Compiler Construction*, published as *SIGPLAN Notices*, 19(6), June 1984, pp. 157–170.

[**GGV86**] H. Ganzinger, R. Giegerich, and M. Vach: “Marvin: a Tool for Applicative and Modular Compiler Specifications”, Forschungsbericht 220, Fachbereich Informatik, Universität Dortmund, July 1986.

[**Gie79**] R. Giegerich: “Introduction to the Compiler Generating System MUG2”, Bericht TUM-I FO-7913, Institut für Informatik, Tech. Univ. München, May 1979.

[**Gie86**] R. Giegerich: “On the Relation between Descriptional Composition and Evaluation of Attribute Coupled Grammars”, Forschungsbericht 221, Abteilung Informatik, Universität Dortmund, July 1986.

[**Goo75**] J. B. Goodenough: "Exception Handling: Issues and a Proposed Notation", *Comm. of the ACM*, 18(12), Dec. 1975, pp. 683–696.

[**HMM86**] R. W. Harper, D. B. MacQueen, and R. Milner: "Standard ML", report ECS-LFCS-86-2, Lab. for Foundations of Comp. Sc., —DCS—, Univ. of Edinburgh, March 1986.

[**Hoa73**] C. Hoare: "Hints on Programming Language Design", STAN-CS-, CSD 73-403, Stanford University, Stanford, Ca, Oct. 1973.

[**Jou84**] M. Jourdan: PhD thesis, thèse de docteur-ingénieur, Université de Paris-VII, May 1984.

[**JP90**] M. Jourdan and D. Parigot: "Techniques for Improving Grammar Flow Analysis", in N. Jones (ed.): *European Symp. on Programming (ESOP '90)*, Copenhaguen, vol. 432 of *Lect. Notes in Comp. Sci.*, pp. 240–255, Springer-Verlag, May 1990.

[**KHZ82**] U. Kastens, B. Hutt, and E. Zimmermann: *Gag: a Practical Compiler Generator*, vol. 141, Lect. Notes in Comp. Sci., 1982.

[**KLM+83**] G. Kahn, B. Lang, B. Mélèse, and E. Marcos: "Metal: a Formalism to Specify Formalisms", *Sci. of Comp. Progr.*, 3, 1983, pp. 151–188.

[**Knu68**] D. E. Knuth: "Semantics of Context-Free Languages", *Math. Systems Theory*, 2(2), June 1968, pp. 127–145. Correction: *Math. Systems Theory*, 5(1), Mar. 1971, pp. 95-96.

[**KR78**] B. W. Kernighan and D. M. Ritchie: *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[**Lor77**] B. Lorho: "Semantic Attributes Processing in the System DELTA", in A. Ershov and C. H. A. Koster. (eds.): *Methods of Algorithmic Language Implementation*, vol. 47, pp. 21–40, Lect. Notes in Comp. Sci., 1977.

[**Mil78**] R. Milner: "A Theory of Type Polymorphism in Programming", *—JCOMS—*, 17(3), Dec. 1978, pp. 348–375.

[**MM82**] A. Martelli and U. Montanari: "An Efficient Unification Algorithm", *ACM Trans. on Progr. Languages and Systems*, 4(2), April 1982, pp. 258–282.

[**Mye84**] E. W. Myers: "Efficient Applicative Data Types", in *11th popl*, pp. 66–75, Jan. 1984.

[**Par88**] D. Parigot: PhD thesis, thèse de docteur, Université Paris XI Centre d'Orsay, May 1988.

[**PDM80**] T. Penello, F. L. DeRemer, and R. Meyers: "A Simplified Operator Identification Scheme for Ada", *sigplan*, 15(7 & 8), July 1980, pp. 82–87.

[**Rep84**] T. Reps: *Generating Language-Based Environments*, MIT Press, 1984.

[**Rob65**] J. A. Robinson: "A Machine-Oriented Logic Based on the Resolution Principle", *J. of the ACM*, 12(1), Jan. 1965, pp. 23–41.

[**RSS+83**] K.-J. Räihä, M. Saarinen, M. Sarjakoski, S. Sippu, E. Soisalon-Soininen, and M. Tienari: "Revised Report on the Compiler Writing System HLP78", report A-1983-1, Univ. of Helsinki, Jan. 1983.

[**RT89**] T. Reps and T. Teitelbaum: *The Synthesizer Generator*, vol. 323 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, Aug. 1989.

[**Ten81**] R. Tennent: *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs, 1981.

[**UDP+82**] J. Uhl, S. Drossopoulos, G. Persch, G. Goos, M. Daussmann, G. Winterstein, and W. Kirchgäßner: *An Attributed Grammar for the Semantic Analysis of ADA*, vol. 139 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, 1982.

[**VM82**] A. O. Vooglaid and M. B. Mériste: "Abstract Attribute Grammars", *Progr. and Computer Software*, 8(5), Sep. 1982, pp. 242–251.

[**Wir82**] N. Wirth: *Programming in Modula-2*, Springer-Verlag, 1982.

Index