This paper appears in *Attribute Grammars, Applications and Systems*, ed. by H. Alblas and B. Melichar, Prague (CZ), June 4–13, 1991, pp. 485–504, vol. 545 of *Lect. Notes in Comp. Sci.*, Springer-Verlag. Copyright © 1991 by Springer-Verlag, New-York, Heidelberg, Berlin.

Internals and Externals of the FNC-2 Attribute Grammar System

Martin JOURDAN Didier PARIGOT

INRIA*

Abstract

FNC-2 is a modern attribute grammar processing system aiming at expressive power, efficiency, ease of use and versatility. This paper provides the reader with a brief tour inside FNC-2, presenting the most important features of its "engine": efficient sequential exhaustive, parallel exhaustive and sequential incremental evaluation of strongly non-circular AGs. These methods are based on the visit-sequence paradigm; the first one makes use of extensive space optimizations. Then we describe the external features of the system—attribute coupled grammar view of an AG, specially-designed AG-description language, with provisions for true modularity, and complete environment—that make it really usable for developing large-scale applications. Experience with the system is briefly reported.

1 Introduction and Design Requirements

Since Knuth's seminal paper introducing attribute grammars (AGs) [Knu68], it has been widely recognized that this method is quite attractive for specifying every kind of syntax-directed computation, the most obvious application being compiler construction. In addition to pure specification-level features—declarativeness, structure, locality of reference—, an important advantage of AGs is that they are executable, i.e., it is possible to automatically construct a program which implements a given AG.

Unfortunately, until recently these automatically generated attributes evaluators were too inefficient in time and/or space to honorably compete with their hand-written equivalents, unless the class of accepted AGs, and hence the expressive power, are severely restricted (see [DJL88] for a good survey of existing AG-processing systems).

Furthermore, another very important reason why AGs are only scarcely used in industrial settings is their lack of modularity, which forces to program a complete application as a single, monolithic file. As applications get larger and more complicated, this scheme becomes more and more unworkable. A striking example is the ADA front-end developed using the GAG system [UDP82]: it is a single AG of more than 500 *pages*!

*Authors' address: INRIA, Projet ChLoÉ, Bât. 13, Domaine de Voluceau, Rocquencourt, BP 105, F-78153 LE CHESNAY Cedex, France. E-mail: `{Martin.Jourdan,Didier.Parigot}@inria.fr`.

Lastly, there are parts of some applications for which tools other than AGs might be more pleasant to use (e.g. attributed tree transformation systems are better than AGs to describe the optimization phases in a compiler), so attribute evaluators need to be able to interface with these other tools.

We believe that all of these issues must be addressed in the design of any system that aims at production-quality. Our goal in designing the FNC-2 AG-processing system was hence to bring up-to-date technology and recent research results together into a really usable system. We wanted FNC-2 to be:

efficient: the generated evaluators should be as efficient in time and space as hand-written programs using the same basic data structures, in particular an explicitly-built tree to represent the input text (we ruled out from the start tree-less methods such as attributes evaluation during parsing for their very poor expressive power, see next item).

powerful: this efficiency should not be achieved at the expense of expressive power, i.e., FNC-2 should be able to process AGs in a very broad class, so that an AG-developer is not (too much) constrained in his design by the system.

easy to use: the input language of FNC-2 should enforce a high degree of programming safety, reliability and productivity.

versatile: the generated evaluators should allow to be interfaced with many other tools and used in various different contexts.

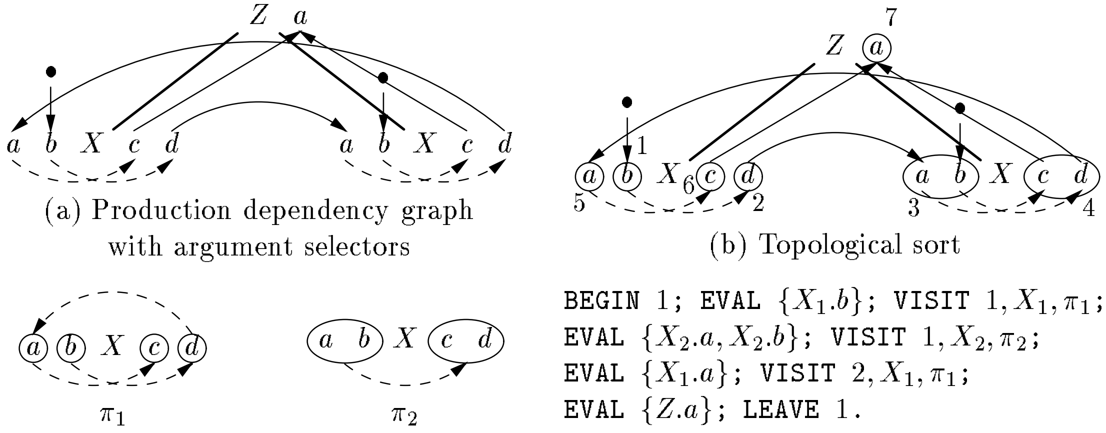
This paper presents the major features of the FNC-2 system, first from the point of view of the implementor (the "internals") and then from that of the user (the "externals").

2 Internals

This section concentrates on the "engine" of FNC-2, i.e. what is invisible from outside but makes it powerful and efficient. What is involved here is the evaluation method(s) implemented by the system. Indeed, this determines both the expected efficiency of the evaluators and the expressive power (accepted AG class) of the generator. A tremendous amount of research work has been devoted to the devising of evaluation methods that reach a good tradeoff between efficiency and expressive power [Alb91b]. To increase versatility, FNC-2 is able to generate evaluators using one of three methods: sequential exhaustive, parallel exhaustive and sequential incremental.

2.1 Sequential exhaustive evaluation

The requirement for FNC-2 to generate efficient evaluators ruled out methods based on dynamic scheduling: in addition to the execution of semantic rules, which is its basic job, an evaluator should do as little work as possible, which means that as much information as possible about the evaluation order should be embodied in the code of the evaluator itself and not computed at run-time. So we decided that FNC-2 would produce evaluators based on the visit-sequence paradigm [Alb91b, Kas91]. In addition to being time-efficient, this technique is amenable to extensive space optimizations.



(a) Production dependency graph with argument selectors

(b) Topological sort

(c) The two partitions generated for X

```
BEGIN 1; EVAL $\{X_1.b\}$; VISIT 1,$X_1,\pi_1$;
EVAL $\{X_2.a, X_2.b\}$; VISIT 1,$X_2,\pi_2$;
EVAL $\{X_1.a\}$; VISIT 2,$X_1,\pi_1$;
EVAL $\{Z.a\}$; LEAVE 1.
```

(d) The corresponding visit-sequence

Figure 1: The classical SNC to *l*-ordered transformation

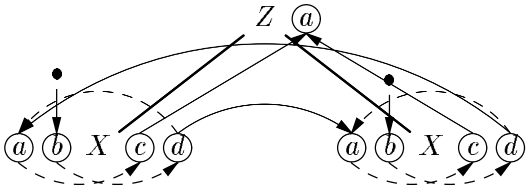
2.1.1 SNC to *l*-ordered transformation

It is possible to build visit-sequences for a given AG only if a total evaluation order can statically be determined. The largest class of AGs for which such a total order exists is the *l*-ordered class [EnF82]. Unfortunately, on one hand this class is not as broad as could be dreamed of, and above all its characterization is an NP-complete problem. There exist two possible approaches for solving this problem:

- restrict the accepted AG class to one more easily characterizable, such as the OAG class [Kas80] for which there exists a polynomial test; however this decreases the expressive power;
- use a variant of the non-circular to *l*-ordered transformation [EnF82] that applies to the strongly (i.e., absolutely) non-circular AG class [CoF82] and is compatible with the visit-sequence paradigm [Rii83].

We used the latter approach for Fnc-2. This rather well-known transformation is described in [EnF82, Fil87, Rii83]; it relies on the computation of *several* totally-ordered partitions of the attributes of each non-terminal, each order being a possible evaluation order for these attributes. The transformation proceeds as follows (see Fig. 1):

1. Compute the argument selectors (IO-graphs).
2. In top-down passes over the AG, compute for each non-terminal a set of totally-ordered partitions of its attributes. For each production:
   (a) pick a totally-ordered partition for the LHS non-terminal;
   (b) paste it together with the production dependency graph and the argument selectors of the RHS non-terminals (Fig. 1(a));
   (c) topologically order the resulting graph (Fig. 1(b)); this determines on each RHS non-terminal a totally-ordered partition (Fig. 1(c)); memorize the mapping from the production number and partition for the LHS non-terminal to the partitions for the RHS non-terminals.



(a) Production dependency graph with π_1 on both X_1 and X_2

```
BEGIN 1; EVAL {X_1.b}; VISIT 1,X_1,π_1;
EVAL {X_2.b}; VISIT 1,X_2,π_1;
EVAL {X_2.a}; VISIT 2,X_2,π_1;
EVAL {X_1.a}; VISIT 2,X_1,π_1;
EVAL {Z.a}; LEAVE 1.
```

(b) Corresponding visit sequence

Figure 2: π_1 can replace π_2 in the production

Repeat until fixed point is reached.

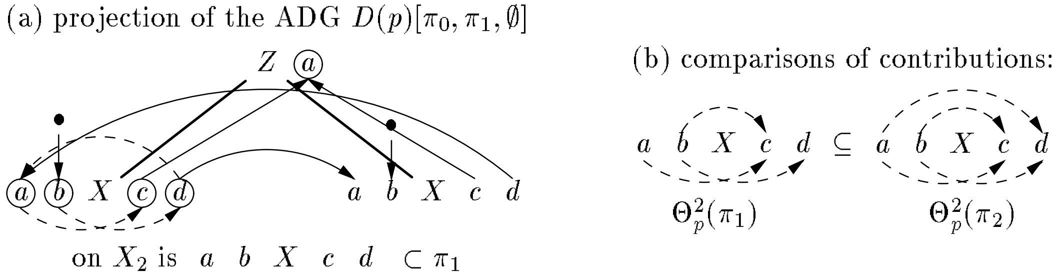
3. The resulting AG has as non-terminals the pairs ⟨old non-terminal, one of its partitions⟩ and productions as determined by the above construction (memorization step). Note however that this AG needs not be explicitly built; rather, during the construction, a visit-sequence-based evaluator can be built, in which recursive `VISIT` instructions carry an additional parameter that identifies the partition to use on the visited node (there exists one visit-sequence per pair ⟨production, partition of its LHS non-terminal⟩) (Fig. 1(d)).

This construction reaches its goal, in that the generated evaluators are completely deterministic, thus fast, while keeping the great expressive power of the SNC class. In fact, it is equivalent (while much simpler to understand) to the well-known construction by Kennedy and Warren [KeW76]. Unfortunately, it has exponential complexity, and the generated evaluators can be exponentially large (more precisely there can be an exponential number of totally-ordered partitions per non-terminal).

One of our works was to reduce this factor, by defining a coarser but still correct notion of equivalence of partitions which allows to strongly limit their proliferation [Par88]. It is based on the observation, illustrated in Fig. 2, that, provided that you accept a slight increase in the number of visits, and under some conditions, you can replace one partition by another one and still be able to generate a correct evaluator. For instance, Fig. 2(a) shows that it is possible to replace π_2 by π_1 on X_2 because the resulting augmented dependency graph remains acyclic. Fig. 2(b) shows the corresponding visit sequence, which calls for two visits to X_2 instead of one in Fig. 1(d). Let's note right now that, on all the practical AGs we have used, the increase in the number of visits is less than 2% in average, and since pure tree-walking accounts only for a very small fraction of the evaluator running time (the rest is for the computation of the semantic rules), the dynamic effect is unnoticeable.

This idea of replacing a partition by an "equivalent" one to reduce their total number is similar to that of *covering* [LoP75], a notion originally devised to reduce the complexity of non-circularity tests. It's quite easy to modify step 2(c) of the above transformation to check for possible replacements [JoP90, LoP75, Par88].

There exist several correct variations of this notion of equivalence of totally-ordered partitions (see [Par88] for more details and the formal proofs). With our "best" variation, called *strong inclusion* and illustrated in Fig. 3, practical tests show that, on most practical AGs, we end up with exactly or very close to *one* partition per non-terminal, when the classical transformation ends up with sometimes more than five partitions per



(a) projection of the ADG $D(p)[\pi_0, \pi_1, \emptyset]$

(b) comparisons of contributions:

Z a
a b X c d
a b X c d
on X_2 is a b X c d $\subset \pi_1$
a b X c d $\subseteq$ a b X c d
$\Theta^2_p(\pi_1)$
$\Theta^2_p(\pi_2)$

Figure 3: Testing strong inclusion

non-terminal on the average. This is clearly a good saving on the evaluators size. An additional benefit is that the running time of the transformation seems to be directly related to the total number of partitions computed for the whole AG, so that our optimized transformation runs much faster that the classical one and in almost-linear time.

2.1.2 Space Management

As described in another paper in this volume [Kas91], the necessity to have a statically-determinable total evaluation order to produce visit-sequence-based evaluators has the beneficial side-effect that we can use this order to conduct very fine static analysis of the lifetime of each attribute instance, which in turn allows to determine the most efficient way to store it. Of course we have integrated these works in FNC-2 (for sequential exhaustive evaluation).

Our achievements [JuP90] give exact conditions to decide whether a temporary attribute can be stored in a global variable rather than in a stack (all temporary attributes can be, at worst, stored in a stack) and whether a non-temporary one can be stored in a global variable. In addition, our algorithm for packing together several variables or several stacks uses a cost criterion (the number of copy rules a potential grouping would eliminate) rather than a mere feasibility criterion; also, since finding an optimal packing is NP-complete, we use heuristics, based on a static traversal of the visit-sequences, to keep an acceptable complexity while achieving near-optimal results.

Presently we work on the last category of attributes, namely the non-temporary ones which cannot be stored in global variables. We think of using a technique similar to that of “bindings” presented in [SwV91], or “cactus stacks,” to store these attributes, and hence *all* the attributes, out of the tree. This will of course improve storage consumption, but the most important consequence is a yet to be estimated, but surely great, increase in the expressive power of AGs. Indeed, since with that scheme the only purpose of the tree is to conduct the evaluator, it needs not be a physical object any more; it only has to *mimic* a physical tree. For instance, attributes evaluation on DAGs (i.e., trees with shared subtrees) comes for free. Other applications need yet to be explored.

2.2 Parallel exhaustive evaluation

We have devised and implemented a method for evaluating attributes on a shared-memory multiprocessor machine [Mar90]. This method is based on the visit-sequence paradigm, which allows to efficiently exploit useful parallelism. It is described more at length elsewhere in this volume [Jou91].

2.3 Incremental evaluation

In addition, we provided for Fnc-2 to be able to generate incremental attributes evaluators [Alb91a, Rep84]. We devised a new incremental evaluation method [Par88] whose outline is as follows:

1. Generate an exhaustive evaluator which is able to start at any node in the tree. This is achieved by computing argument selectors as for the SNC class, but these selectors must be closed both "from below" and "from above" [Fil87]. This construction hence applies only to a subclass of SNC AGs called doubly non-circular (DNC). This class is however larger than the l-ordered class, and our SNC to l-ordered transformation allows to actually use this method for any SNC AG. Also, the DNC class is characterizable in polynomial time. This evaluator is efficient since it is completely deterministic.

2. Add to this exhaustive evaluator a set of "semantic control" functions allowing to limit the reevaluation process to affected instances. This control is based on the determination of the status of each attribute instance (changed, unchanged or unknown) and the comparison of its old and new values. Note that the notion of equality used in this comparison can be adapted to the problem at hand, which enhances versatility.

Parigot [Par88] also describes how this method can accommodate multiple subtree replacements.

Reps [Rep84] proposes two incremental evaluation methods, one for (plainly) non-circular AGs and one for ordered AGs (see also [Eng84, Yeh83]). Our method should prove more efficient than the former and as efficient and more powerful than the latter. We believe that it is quite attractive, in particular because of its versatility, but this must be confirmed by practical experiments that we have not yet conducted.

2.4 Implementation: The Evaluator Generator

The evaluator generator (see Figure 4) is the "engine" of Fnc-2. It receives as input an abstract AG (syntax and local dependencies) and produces an abstract evaluator (visit sequences with calls to evaluate the semantic rules, memory map and storage manipulation operations).

The first step in the construction is the SNC test. The whole process aborts if the AG fails this test, but then an interactive circularity trace system [Par85] allows to easily discover the origin of the failure; this allows to take full advantage of the power of the SNC class. Then the DNC test is performed; if it succeeds, the OAG test is performed. Actually there exists an infinity of incomparable OAG(k) classes [Bar84]; by default Fnc-2 performs the OAG(0) test but can be directed to test for the OAG(k) class for any given k. If either the DNC or OAG test fails, the SNC to l-ordered transformation is invoked. Note that cascading these phases costs the same as performing the OAG test from scratch, since the first phase of the OAG test is the DNC test, and the first phase of the latter is the SNC test.

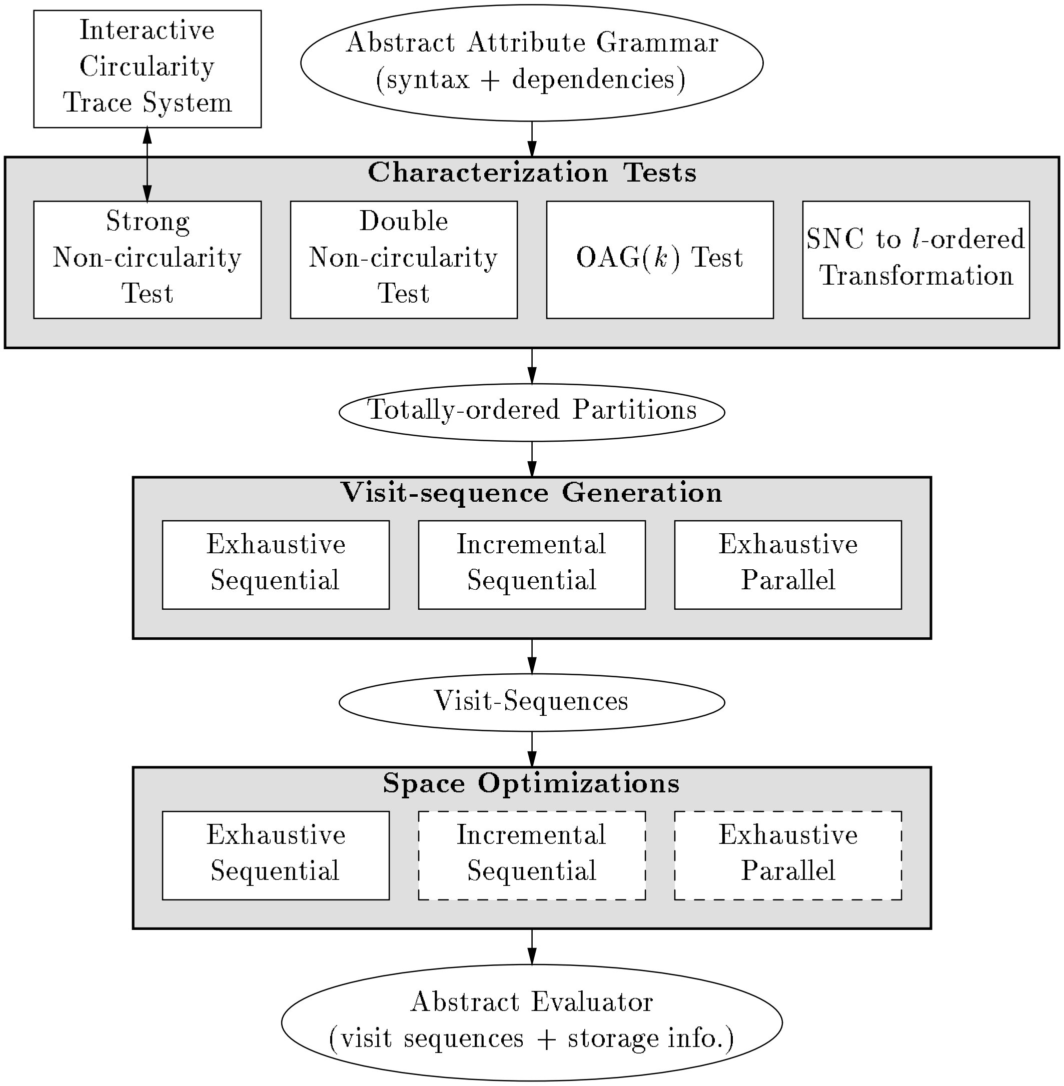

Interactive Circularity Trace System
Abstract Attribute Grammar (syntax + dependencies)
Characterization Tests
Strong Non-circularity Test
Double Non-circularity Test
OAG(k) Test
SNC to l-ordered Transformation
Totally-ordered Partitions
Visit-sequence Generation
Exhaustive Sequential
Incremental Sequential
Exhaustive Parallel
Visit-Sequences
Space Optimizations
Exhaustive Sequential
Incremental Sequential
Exhaustive Parallel
Abstract Evaluator (visit sequences + storage info.)

Figure 4: The evaluator generator

The next step is the production of the visit-sequences from the total orders computed either in the OAG test or in the transformation. There are three methods to do so, as described earlier, and only one of them is chosen by the user.

For sequential exhaustive evaluation, the final step is the space optimization, which produces the memory map and enriches the visit-sequences with storage manipulation operations. We are working on other space optimization techniques for our two other evaluation paradigms.

All the algorithms cited above belong to the Grammar Flow Analysis framework [MöW91] and make heavy use of our improvements to this technique [JoP90]. Thus the evaluator generator is quite fast: an evaluator for a complete Pascal to P-code compiler is produced in less than two minutes on a Sun-3/60 (see also Table 1).

AG	1	2	3	4	5	ave.
# phyla	6	12	35	35	74	
# operators	27	47	131	131	319	
# occ. attr.	20	98	94	320	992	
# sem. rules	99	417	444	1420	5318	
class	OAG(0)	OAG(0)	OAG(0)	OAG(0)	DNC	
% variables	80	86	33	61	60	56.5
% stacks	20	7	62	35	28	36
% non-temp.	0	7	5	4	12	7.5
% elim./copy	65	64	20	20	30	27
% elim./poss.	74	88	89	84	94	91
time	1.78	9.03	10.00	45.31	489.41	

Table 1: Statistics gathered for the evaluator generator

2.5 Efficiency of the Evaluator Generator

Table 1 presents some figures gathered from the execution of the evaluator generator (as part of Fnc-2) on various AGs, when generating a sequential exhaustive evaluator. These AGs describe parts of the Fnc-2 system itself.

The top part of Table 1 gives an idea of the size of each AG. Phyla and operators correspond roughly to non-terminals and productions in a concrete grammar, see section 3.2.2. The number of attribute occurrences is the sum on all phyla of the number of attributes attached to each.

The so-called class of an AG is the smallest class to which it belongs, as determined by the evaluator generator. AG 5 is not OAG(k) for any k, which shows that it is advantageous to have a system accepting a class larger than OAG. On this AG, our SNC to l-ordered transformation ends up with 1.04 totally-ordered partitions per non-terminal in average (max. 2), whereas the classical transformation ends up with 4.15 partitions per non-terminal in average (max. 29)! We also have found an AG in the sources of Fnc-2 that is not OAG(0) but is OAG(1).

The next part of Table 1 deals with space optimization. The first three figures give the proportion of attribute occurrences which are stored in global variables, global stacks or at tree nodes; the latter class corresponds to non-temporary attributes because we have not yet implemented the test for storing non-temporary attributes in global variables. These figures are static, i.e., they refer to the AG itself rather than to its dynamic execution. Dynamic measures [JuP90] show a decrease of the number of attribute storage cells by a factor of 4 to 8 in the execution of AG 5 on various source texts. These figures are already quite good and will be even better when our space optimization scheme is fully implemented. Then we come to the effects of our grouping algorithm. On AG 5 for instance, it cuts the number of global variables from 595 down to 106 and the number of global stacks from 278 to 49. More important are the figures in the table which describe the proportion of copy rules which have been eliminated. The first one gives this proportion w.r.t. the total number of copy rules; again, this is a static figure. It may seem low, but it must be noticed that not all copy rules can be eliminated: for instance, storing two different attribute occurrences in the same variable makes the evaluator incorrect when both are live at the same time. The next figure—the proportion of rules that have *actually* been eliminated w.r.t. those which could *theoretically* be eliminated—show that we reach close to the optimum (remember that achieving the optimum is NP-complete).

The last figure is the CPU time on a Sun-3/60 machine. The whole process is clearly non-linear but also non-exponential. We believe that the efficiency of the evaluator generator is reasonable given the extensive optimizations it performs.

3 Externals

This section is devoted to the "body" of the FNC-2 system, i.e. what is directly visible to the user and makes it easy to use and versatile. Indeed, as far as programming and developing applications are concerned, the efficiency of the "engine" does not matter much, provided of course that it stays reasonable. Much more important is the way the programmer interacts with the system, i.e. the input language and the general development paradigm.

3.1 What an application is for FNC-2

In order to tackle the "programming-in-the-large" problems described in the introduction, the paradigm we chose to put at work in FNC-2 is to consider that an AG specifies, and an attribute evaluator implements, an attributed-tree to attributed-tree mapping, i.e. an evaluator takes as input an attributed tree and produces as output another attributed tree (or zero or more than one, actually). This paradigm was called *attribute coupled grammars* (ACGs) by its inventors [GaG84]. It responds to the above-quoted problems as follows:

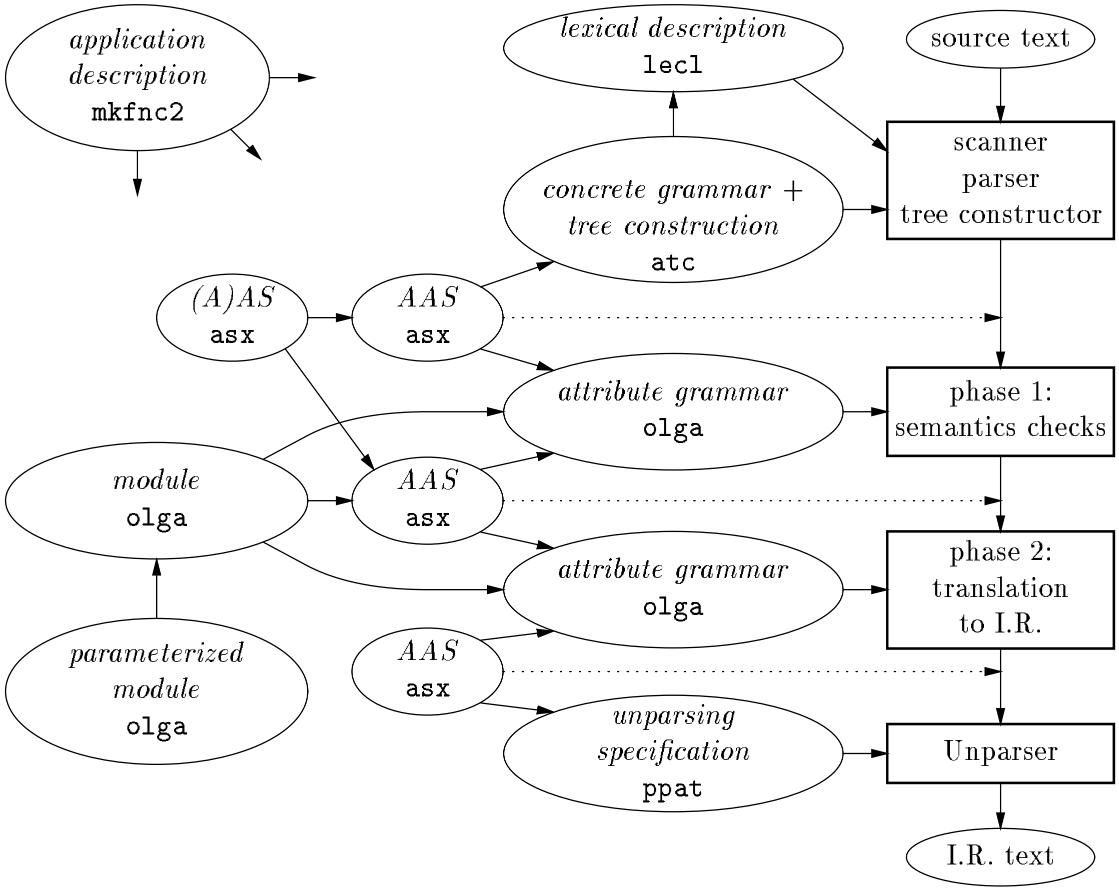
1. A large application can be split into a sequence of passes, each pass taking as input the intermediate representation produced by the previous one and transforming it into another intermediate representation to be fed to the next pass. Each pass can be described by an AG, of course, and each of them is thus much smaller and hence much more easily manageable than a single AG specifying the same translation as the whole sequence of passes.

2. In this sequence of passes, some can be described by AGs while others can be described and implemented by other tools. The tree-to-tree mapping paradigm is indeed quite general and many tools are based on it.

Note here that, for FNC-2, trees are *abstract trees* rather than concrete ones, see section 3.2.2.

In this scheme, the intermediate trees are described by grammars extended with attribute declarations (to avoid confusion we call such things *attributed abstract syntaxes* (AAS)). Each such AAS potentially "belongs" to more than one AG: it is the input AAS of one or more AGs and the output AAS of one or more other AGs. To avoid duplication of work and possible inconsistencies, and to facilitate modularity and reusability, we have decided to separate the specification of these AASes from the AGs themselves.

It should be clear by now that the most important components of (the specification of) an application, at least as far as FNC-2 is concerned, are attribute grammars, which spec-



application description mkfnc2
lexical description lecl
source text
scanner parser tree constructor
concrete grammar + tree construction atc
(A)AS asx
AAS asx
attribute grammar olga
phase 1: semantics checks
module olga
AAS asx
attribute grammar olga
phase 2: translation to I.R.
parameterized module olga
AAS asx
unparsing specification ppat
Unparser
I.R. text

Figure 5: Structure of a typical application

ify the computations performed by the various passes, and attributed abstract syntaxes, which specify the input and output data (attributed abstract trees) of these passes.

In addition, FNC-2 comes with a number of companion processors that help build complete applications:

- a generator of abstract tree constructors driven by parsers (*atc*); two instantiations of *atc* have been implemented, one on top of SYNTAX,[1] and one on top of *Lex* and *Yacc*;
- a generator of unparsers of attributed abstract trees (*ppat*), based on the TEX-like notion of nested boxes of text;
- and a tool for describing the modules composing an application and managing their processing (*mkfnc2*).

Note that the input languages of all these tools have much in common with OLGA, the language for describing AGs, and *asx*, the language for describing AASes.

Fig. 5, which depicts the organization of a typical application as can be constructed by FNC-2 and its companions alone, should now be easily understandable. More details will be given in the next section, which briefly describes OLGA. Let us add only a few

[1] SYNTAX is a trademark of INRIA

words: the application is the front-end of a compiler. The first pass checks that the contextual constraints of the source language are verified. The second pass translates the source tree to some intermediate representation. The final unparsing pass would be used for debugging purposes and needs not be present if the IR is to be further processed by a complete back-end. However there exist applications, e.g. source-to-source translation, in which the output is some high-level language; in that case *ppat* can generate a suitable "back-end" (see e.g. Fig. 8).

3.2 The OLGA AG-description language

OLGA was designed for the description of all aspects of an attribute grammar. This of course involves constructs to declare attributes, access them in semantic rules, etc., but also constructs to describe pure calculations without resorting to a foreign language. Hence, in addition to being a specialized language for describing AGs, OLGA is also a general-purpose applicative language. We'll briefly present these two aspects now; more details can be found in [JLP90, JoP89].

3.2.1 OLGA as a general-purpose applicative language

First of all, OLGA is a *purely applicative* language, which means that there is no assignable variable and side effects, just pure expressions and functions. This is a strong step towards programming safety and reliability. However, OLGA is not (yet) a *functional* language in the sense of ML. The basic objects of OLGA are thus values and functions. There are no control-flow constructs but value-selection ones; iteration is replaced by recursion.

OLGA is a *strongly typed* language, the other necessary condition for programming safety. The collection of predefined types is rather classical. Type constructors are enumerations, subranges of scalar types, records, discriminated unions, sets of scalar values and homogeneous lists. Structured types may be recursive. Each type definition involves the definition of corresponding construction or conversion functions. In addition, in a "functional" AG there are tree types and tree values (see below). Functions, including user-defined ones, may have a *polymorphic* profile as in ML, although OLGA polymorphism is restricted to set and list types. We also have a *type inference* algorithm.

OLGA is of course *block-structured*. The basic scope rules are similar to those of Ada. As in Ada also, function names can be *overloaded*; we found this feature very convenient to enhance the readability of OLGA programs.

OLGA supports the notion of *modules*, in which one can define a set of related objects (types, functions, constants, values, i.e. run-time constants, and exceptions). OLGA modules are similar to those found in e.g. Ada and Modula-2. A module is split in two actual compilation units, a *declaration module* (DCM) which declares the objects which are visible from outside, and a *definition module* (DFM) in which the actual implementation of these objects (and maybe other, non-visible objects) is given. Type checking is performed across module boundaries. Types and other objects in a declaration module may be specified as *opaque*. In addition, a module may be *parameterized* by types and/or functions. This supports the notion of abstract data types. Two such modules are depicted in Fig. 5.

Olga provides a powerful *pattern matching* construct unifying structural matching, selection driven by a scalar value and test for a union tag. Olga also provides for the declaration, activation and handling of *exceptions*. User-defined exceptions may return parameters. There also exist some constructs which somewhat violate the applicative character of Olga but which we felt we had to include anyhow because they are so useful: production of *error messages*, interface with *external functions* written in some foreign language, and construction of *circular structures*.

3.2.2 Olga as an attribute grammar description language

There are two kinds of AGs in Olga, side-effect and functional, respectively roughly equivalent to classical AGs and ACGs. A side-effect AG is one in which there is exactly one output tree which is exactly the same as the input one except that it carries different attributes. A functional AG has zero, one or more output trees, generally different from the input one, which must be constructed piecewise in semantic rules and carried by (synthesized) attributes. In Fig. 5 for instance, phase 1 is a side-effect AG, since no translation is involved here, only information is computed and added to the tree; phase 2 is a functional AG, since its output is syntactically different from its input.

Note that this provision for modularity-through-composition entails no loss of efficiency, since it is possible to mechanically construct, from two AGs coupled into a "pipe," a single AG which performs the same translation as the original sequence but without physically constructing the intermediate tree. This is achieved through mere merging (for side-effect AGs) or *descriptional composition* [GaG84].

As said above the syntactic base of AGs written in Olga is not a concrete syntax as in the classical framework [Knu68] but rather an *abstract syntax* as in [VoM82]. Abstract grammars (see e.g. [ASU86]) express the structure, rather than the textual appearance, of the notions and constructs of the languages they model. This frees the grammar writer from cumbersome syntactic constraints which stem from a particular parsing method and, on the other hand, allows smaller trees since they do not include keywords, simple productions and such spurious things. In addition, our experience is that this allows to make the trees closer to the true semantics of a language, which makes the AGs shorter and simpler. The abstract syntax formalism used for Olga is very close to Metal [KLM83]. It provides for heterogeneous, fixed arity *operators*, similar to concrete productions, and homogeneous, variable-arity operators (*list nodes*). Operators are grouped into *phyla*, which roughly correspond to non-terminals. In a functional AG the phyla of the output AAS(es) are tree types whose constructors are the operators.

In addition to what we have already said regarding modularity with Fnc-2 and the ACG concept, let us note that an AAS can be imported in another one. This is useful to describe the "profile" of a side-effect AG: since the (attribute-less) syntax is the same for both the input and output AAS, it needs be specified only once; this "base" syntax is then imported in the actual input and output AASes, which merely specify what attributes are attached to this syntax (see Fig. 5). The types of the attributes of an AAS must be defined in a separate DCM.

An AG is structured into *phases* and, of course, productions. A phase is purely a structuring construct which has no effect on the AG in the classical sense (i.e. it is "transparent" to productions and semantic rules), except that it is a block and hence

```
where decls -> DECL* use
   $in-env(DECL) := case position is
            first: $in-env(decls);
            other: $out-env(DECL.left);
         end case;
   $out-env := case arity is
            0: $in-env;
            other: $out-env(DECL.last);
         end case;
   $correct := map left & value true
            other $correct(DECL)
         end map;
end where;
```

Figure 6: Examples of semantic rules for a list production

may contain local declarations and import clauses. For instance, an AG describing the verification of the contextual constraints for some programming language might have a phase for name analysis and one for type analysis; each phase needs not know about the functions used in the other, and they communicate only through the attributes.

Each production is also a block; values local to a production may depend on attributes of this production and hence play the role of what is usually referred to as "local attributes." Thus, the condition that an AG be in normal form is not too constraining. A production may appear several times in each AG or phase when this improves the readability but, of course, a given attribute should be defined only once.

In each AG there are three kinds of attributes. The *input* and *output* attributes are those carried by the input and output AATs and are declared in the corresponding AASes. Input attributes are constants. In side-effect AGs, output attributes must be given a *direction* (inherited or synthesized) and be defined by semantic rules, whereas in a functional AG they have no direction and must be attached to the corresponding AAT while it is being built. *Working* attributes are "local" to the AG and correspond to classical ones; for instance they carry the output AATs during their construction. They must have a direction.

In Olga, a semantic rule is written, as expected, as the assignment of some value to some attribute occurrence. The right-hand side expression can refer, in addition to attributes of the production at hand, to attributes of nodes upward in the tree and attributes of subtrees (after suitable pattern matching to control the access). Olga provides several special constructs for semantic rules in list productions. Fig. 6 presents a (hopefully self-explanatory) example of these constructs; please refer to [JLP90, JoP89] for more details.

Olga allows to leave the definition of some attribute occurrences unspecified in an AG, and Fnc-2 will try to infer the corresponding semantic rules from the context. Automatically generating *copy rules* is a rather well-known technique now, and we have implemented it in Fnc-2; in Fig. 6 for instance, with suitable declarations for `$in-env` and `$out-env`, the first two rules would have been automatically generated. However, there still remain a lot of semantic rules which "look the same" and that you have to write in whole. Thus, we have extended Fnc-2's mechanism to provide for automatic generation of non-copy rules [JLP90]. The basic mechanism is the definition of *attribute classes*, which are sets of attribute occurrences, and associated templates to specify the semantic rules which define these occurrences. The templates are actually structured in two levels, a syntactic level used to specify in which productions the templates will be applied, and a semantic level which correspond to the actual rules. When some attribute occurrence is not explicitly defined in a given production, FNC-2 searches whether this attribute belongs to some class; if so, it tries to match some syntactic template in the class definition with the production at hand and the corresponding semantic template with some context conditions that space does not permit to explain in detail here; if all these conditions are verified, the actual rule is created.

In addition to the shortening of the text of an AG, the other benefit of this feature—the most important in our opinion—is that this gives a more "semantic" view of the AG, because closely related semantic computations are grouped in a single place (the definition of a class) rather than being spread over several syntactic productions. The phase construct supplements this semantic structuring.

3.3 Experience with application development

Until now we have worked on two "pure" real-size applications:

- a compiler from the parallel logic language PARLOG to code for the SPM abstract machine [GJR87];
- a compiler from ISO-Pascal to P-code, which had been originally written in Lisp for the predecessor of FNC-2 and was translated to OLGA (work not yet completed).

In addition, FNC-2 is used in the PAGODE code generator generator [DMR90], both to develop the generator itself and to compile some of the modules it generates. However, the largest application of our system, and the one with which we have the deepest experience, is the development of FNC-2 itself, through bootstrap, and of its companions. This explains why we have decided to use it as the "test case" for this discussion.

The most important component in FNC-2 is the OLGA compiler, whose structure is depicted in Fig. 7. The OLGA reader groups the scanner, the parser and the constructor for the first tree; it is generated by *atc* and SYNTAX. The checker is the largest component in FNC-2; it performs the verification of the contextual constraints, generates the missing semantic rules and splits the AG into an abstract AG to be input to the evaluator generator, containing only the syntax (list operators are expanded into a finite set of fixed-arity productions) and attribute dependencies, and the rest of the AG (types, functions and expressions in the right-hand side of the semantic rules) which is fed to the translator(s). It is written as a combination of four AGs: the first, biggest one performs type checking; another one analyzes dependencies, generates missing semantic rules and produces the abstract AG; the two others perform various optimizations on function bodies and semantic rules (tail recursion elimination, construction of a deterministic decision tree for the pattern matching construct, etc.). All these AGs make heavy use of out-of-line functions defined in separate modules which are also written in OLGA. The evaluator generator is the other most important component in the system (see section 2.4); since the basic data structures it manipulates are graphs, and since most of the algorithms it

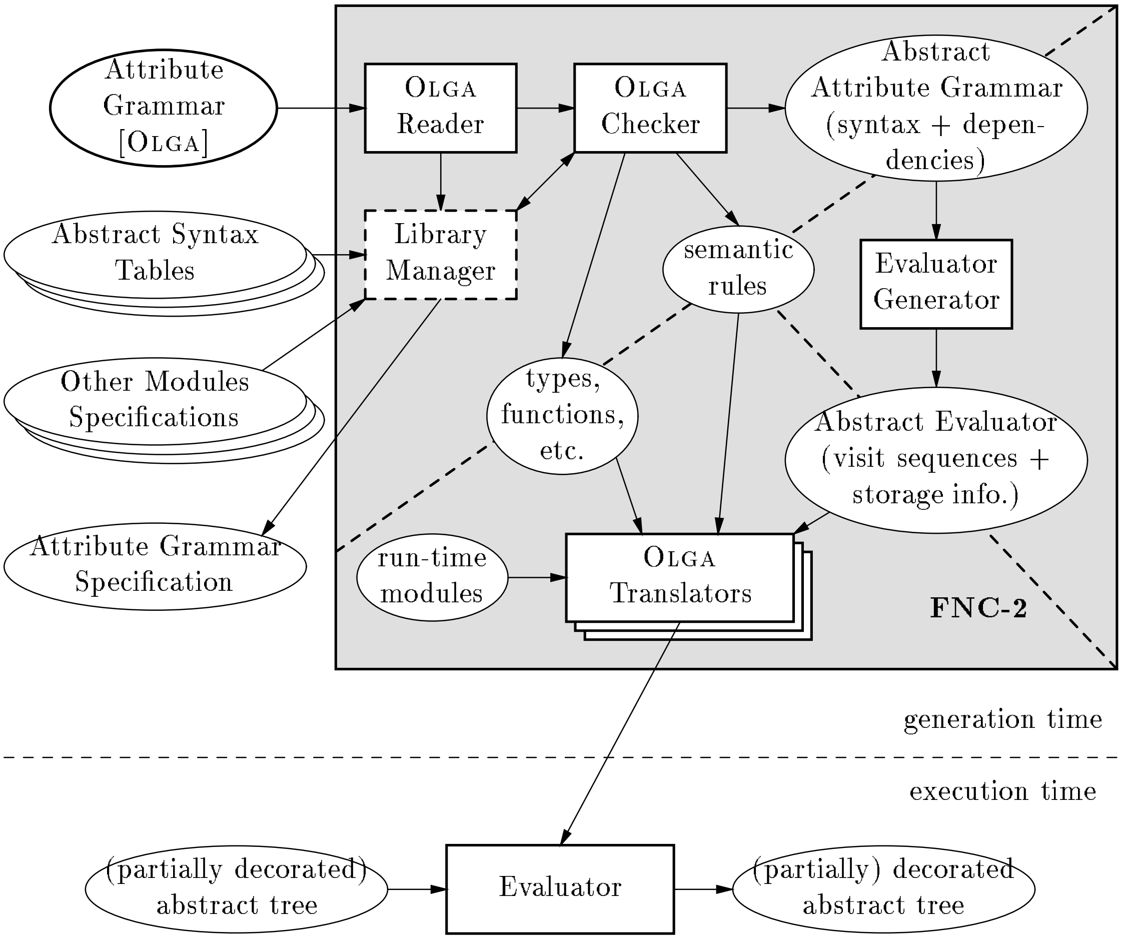

Attribute Grammar [OLGA]
OLGA Reader
OLGA Checker
Abstract Attribute Grammar (syntax + dependencies)
Abstract Syntax Tables
Library Manager
semantic rules
Evaluator Generator
Other Modules Specifications
types, functions, etc.
Abstract Evaluator (visit sequences + storage info.)
Attribute Grammar Specification
run-time modules
OLGA Translators
FNC-2
generation time
execution time
(partially decorated) abstract tree
Evaluator
(partially) decorated abstract tree

Figure 7: The FNC-2 system

uses involve fixed-point computations—two things which are cumbersome or even impossible to specify with AGs—, it is written in C. The last components of FNC-2 are the translators to the various implementation languages; presently there exist two of them, one to C and one to Lisp. They are also written in OLGA (and soon in *ppat*). The library manager does not exist yet; its job is performed out of line by *mkfnc2*.

The organization of FNC-2 into three parts (front-end, evaluator generator and back-end(s)) with well-defined interfaces allows to use and test these parts separately. For instance, the evaluator generator was running long before the rest of the system; we constructed a crude interface (textual version of our abstract AGs) which allowed to debug it and can still be used to operate it when you don't want to use OLGA; the internal representations produced by the crude interface and by the OLGA front-end have the same structure. In addition, the separation of concerns makes each part simpler; for instance, the "high-level optimization" AG quoted above acts only on the non-AG part of the OLGA language, and this is reflected in the input AAS. Lastly, the ACG paradigm makes it easy to add new features; for instance the optimization pass was not included in the first design but was very easy to insert later without major reworking of the other components.

Even in the presently incomplete status of OLGA—the most important missing features are full polymorphism, parameterized modules and exceptions—, we found it generally

	#	# lines			
	files	min.	max.	total	ave.
AGs	7	354	3,212	10,118	1,445
AASes	8	8	381	779	97
DCMs	15	28	391	2,891	193
DFMs	15	55	3,188	13,404	894
atc	4	60	2,089	2,575	644
total	49	8	3,212	29,767	607

Table 2: Source files in the FNC-2 system

satisfying. In particular, there are very few parts of the system which could not be written in OLGA; the most prominent are the evaluator generator and the parts which read from or write to files (*ppat* will provide a solution for the latter). This shows that the expressive power of the language is sufficient for our purposes. The automatic generation of most copy rules and many non-copy rules greatly improves the readability of the AGs, especially for code-generation-like applications. The only slight problem lies in a relative inefficiency: the OLGA compiler translates modules (the evaluator generator is not invoked) about four times slower than Sun's `cc` compiler operates on files of the same length and uses much more memory (although still reasonably) [JPJ90]. This is not intolerable but we're nevertheless working on improving this, in particular by adding a garbage collector.

We would like to point out here that AGs are, to the best of our knowledge, the only programming method which really supports incremental development: you may freely add new attributes and semantic rules, and then test your AG without having to completely specify it. This is made easier by the great expressive power of FNC-2, i.e. the SNC class: many AGs we have written were, at some time during their development, not ordered and not even *l*-ordered (and our bigger AG still isn't). Of course, with more or less work, it would be possible to make them actually ordered, but using FNC-2 gives us more freedom in our development. Also, a great expressive power is quite useful when AGs are automatically produced by other systems, such as PAGODE: designers of these systems can ignore "mundane" issues such as evaluation order.

But the single most beneficial feature of FNC-2 and OLGA is their support for modularity: as of mid-1990 there were about 30,000 lines of OLGA, *asx* and *atc* code in the system, summarized in Table 2 (these exclude the *ppat* subsystem, which is still under development, the files processed by SYNTAX, and the C files). As can be seen, the files are rather small and hence very readable. Even the 3,212-lines AG (an earlier version of the OLGA static semantics checker) is easy to understand because it is split in several phases (name analysis, type checking, check for well-definedness of the AG and generation of missing semantic rules, generation of the abstract AG) which address only one problem. We would have had much greater problems if we had had to specify the FNC-2 system in a single AG. Furthermore, separate compilation comes with (true) modularity, and this saves much time in the development process. As said above, another benefit is that it is easy to add new components in the system. Yet other benefits are that this makes teamwork easier, and that it is easy to reuse modules in different applications; for instance, the two instantiations of *atc* share most of their stuff.

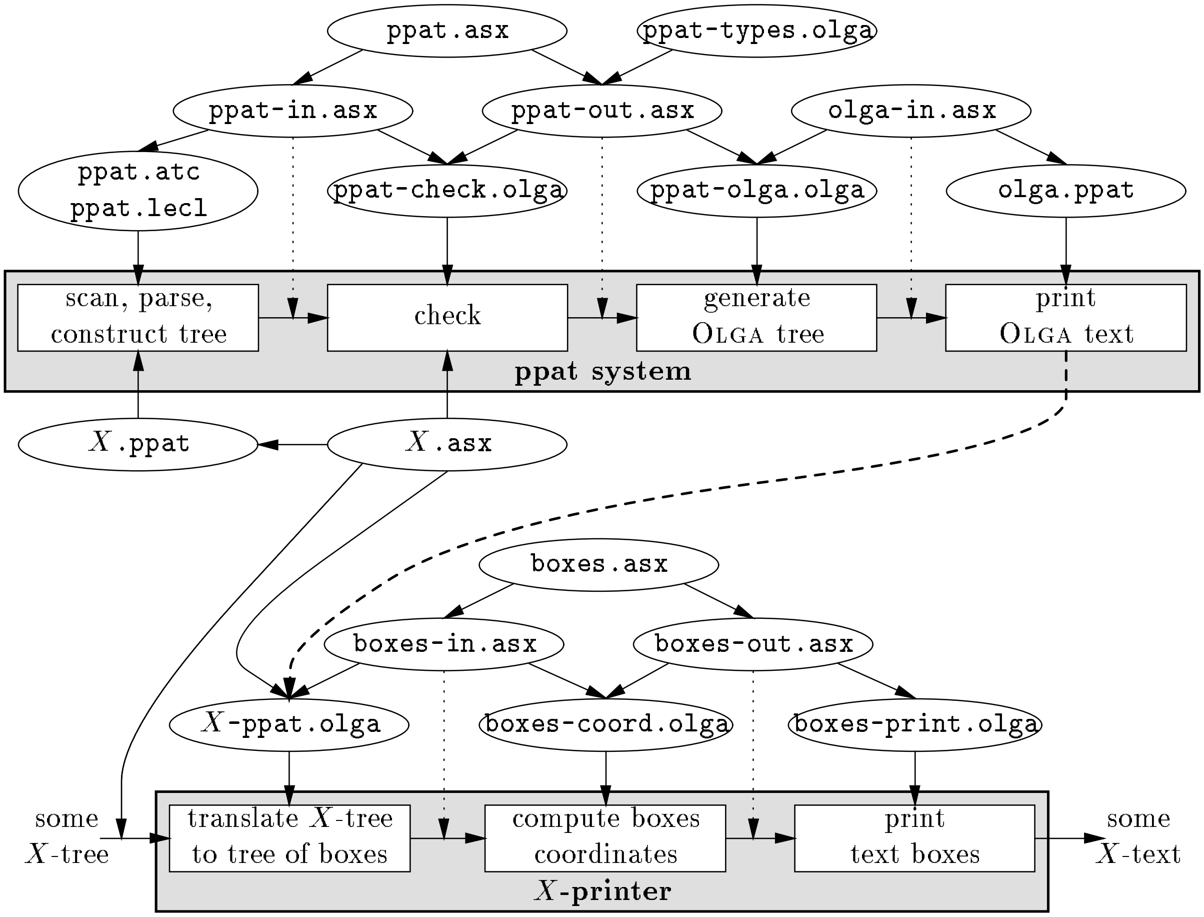


Figure 8: Generation and use of the *ppat* subsystem

As we gained experience, we learnt to push the usage of AGs and modularity very far. For instance, Fig. 8 depicts the organization of the *ppat* subsystem and the pretty-printers it generates, with all the files involved in the process. Files X.asx and X.ppat are written by the user; they respectively specify the input attributed abstract trees and their textual representation. File X-ppat.olga is generated from those by *ppat* and then combined with files boxes* to form the actual unparser. Apart from these X-files, all the files need be written only once. Each one of them is rather small and easy to read. Furthermore, it is easy to reuse them for other purposes (e.g. olga.asx is the AAS for OLGA, which is borrowed from FNC-2). This figure also illustrates our bootstrap philosophy (file olga.ppat).

It is quite hard to measure the efficacy of a programming language, i.e. the relative productivity of a programmer using this language. Let us however quote a single figure regarding OLGA: the above-described 30,000 lines, together with other support files, were written and tested by one man in not much more than one year (the development team grew quite a bit last year, so our experience with teamwork is real).

4 Conclusion and Future Work

We have presented the Fnc-2 attribute grammar system, which aims at production-quality by providing efficiency, expressive power, ease of use and versatility. Both the "engine" (the evaluator generator and the evaluation methods it implements) and the "body" (programming paradigm, input language Olga) were described at length. We have put the emphasis on how Fnc-2 and Olga can be used to develop large applications. Of paramount importance for the efficacy of the development is Fnc-2's support for modularity. Our experience is not very deep yet but it already shows that we now have a really usable system.

We still have quite a lot to do, though, before reaching production quality. One of the first items on the list is to build a clean and stable version of the system and distribute it to willing "customers." We're also in the process of integrating Fnc-2, Centaur [BCD88] and GiGAS [Fra89] into a powerful and attractive competitor to the Synthesizer Generator; in addition, this will serve as a testbed for our incremental evaluation algorithm and for application-specific propagation termination conditions. Lastly, work is being done to improve the translation of Olga to C and implement descriptional composition.

Future work will address parallel evaluation (perform more experiments, restrict parallelism to "still more useful" one by static analysis of the AG, devise specific space optimization techniques), the definition of Olga (unification of syntactic and semantic domains, generic AGs, etc.), source-level optimizations of AGs resulting from descriptional composition, the exploration of the greater expressive power we can achieve when we can store all the attributes out of the tree (conditional AGs), and more.

Acknowledgements Apart from the authors, the following people have contributed to the development of Fnc-2: Joël Bonnet, Christine Ayrault, Catherine Julié, Aziz Souah, Souad Taouil, Olivier Durin, Carole Le Bellec, Étienne Planes, Catherine Zylberman, Bruno Marmol, David Devillard, Gilles Roussel, Roberto Gomez and Jean-Philippe Jouve.

The work described here was supported in part by the CEC under ESPRIT Project # 5399 Compare and will be pursued in this framework. The goal of Compare is to build a complete, highly usable, production-quality compiler-construction workbench.

References

[**ASU86**] A. V. Aho, R. Sethi & J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1986.

[**Alb91a**] H. Alblas, "Incremental Attribute Evaluation," in this volume, 1991.

[**Alb91b**] ———, "Attribute Evaluation Methods," in this volume, 1991.

[**Bar84**] K. Barbar, "Classification des grammaires d'attributs ordonnées," Univ. de Bordeaux I, Rapport 8412, Apr. 1984.

[**BCD88**] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual, "Centaur: the System," *SIGSOFT Software Eng. Notes* **13** (Nov. 1988), 14–24, Joint issue with *ACM SIGPLAN Notices 24*, 2 (Feb. 1989).

[CoF82] B. Courcelle & P. Franchi-Zannettacci, "Attribute Grammars and Recursive Program Schemes," *Theoret. Comput. Sci.* **17** (1982), 163–191 and 235–257.

[DJL88] P. Deransart, M. Jourdan & B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. #**323**, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[DMR90] A. Despland, M. Mazaud & R. Rakotozafy, "Pagode: A Back-end Generator using Attributed Abstract Syntaxes and Term Rewritings," in *Compiler Compilers '90*, Dieter Hammer, ed., Lect. Notes in Comp. Sci. #**477**, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1990, 86–105.

[Eng84] J. Engelfriet, "Attribute Grammars: Attribute Evaluation Methods," in *Methods and Tools for Compiler Construction*, Bernard Lorho, ed., Cambridge Univ. Press, New York, NY, 1984, 103–138.

[EnF82] J. Engelfriet & G. Filè, "Simple Multi-Visit Attribute Grammars," *J. Comput. System Sci.* **24** (June 1982), 283–314.

[Fil87] G. Filè, "Classical and Incremental Attribute Evaluation by Means of Recursive Procedures," *Theoret. Comput. Sci.* **53** (Jan. 1987), 25–65.

[Fra89] P. Franchi-Zannettacci, "Attribute Specifications for Graphical Interface Generation," in *Information Processing '89*, G. X. Ritter, ed., North-Holland, Amsterdam, Aug. 1989, 149–155.

[GaG84] H. Ganzinger & R. Giegerich, "Attribute Coupled Grammars," *ACM SIGPLAN Notices* **19** (June 1984), 157–170.

[GJR87] J. Garcia, M. Jourdan & A. Rizk, "An Implementation of PARLOG Using High-Level Tools," in *ESPRIT '87: Achievements and Impact*, Commission of the European Communities—DG XIII, ed., North-Holland, Amsterdam, Sept. 1987, 1265–1275.

[Jou91] M. Jourdan, "A Survey of Parallel Attribute Evaluation Methods," in this volume, 1991.

[JLP90] M. Jourdan, C. Le Bellec & D. Parigot, "The Olga Attribute Grammar Description Language: Design, Implementation and Evaluation," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 222–237.

[JoP89] M. Jourdan & D. Parigot, *The FNC-2 System User's Guide and Reference Manual* release 0.4, INRIA, Rocquencourt, Feb. 1989, This manual is periodically updated.

[JoP90] ———, "Techniques for Improving Grammar Flow Analysis," in *European Symp. on Programming (ESOP '90)*, Neil Jones, ed., Lect. Notes in Comp. Sci. #**432**, Springer-Verlag, New York–Heidelberg–Berlin, May 1990, 240–255.

[JPJ90] M. Jourdan, D. Parigot, C. Julié, O. Durin & C. Le Bellec, "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System," *ACM SIGPLAN Notices* **25** (June 1990), 209–222.

[JuP90] C. Julié & D. Parigot, "Space Optimization in the FNC-2 Attribute Grammar System," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 29–45.

[KLM83] G. Kahn, B. Lang, B. Mélèse & É. Marcos, "Metal: a Formalism to Specify Formalisms," *Sci. Comput. Programming* **3** (1983), 151–188.

[Kas80] U. Kastens, "Ordered Attribute Grammars," *Acta Inform.* **13** (1980), 229–256.

[Kas91] ———, "Implementation of Visit-Oriented Attribute Evaluators," in this volume, 1991.

[KeW76] K. Kennedy & S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars," in *3rd ACM Symp. on Principles of Progr. Languages*, Jan. 1976, 32–49.

[Knu68] D. E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2** (June 1968), 127–145.

[LoP75] B. Lorho & C. Pair, "Algorithms for Checking Consistency of Attribute Grammars," in *Proving and Improving Programs*, Gérard Huet & Gilles Kahn, eds., INRIA, Rocquencourt, July 1975, 29–54.

[Mar90] B. Marmol, "Évaluateurs d'attributs parallèles sur multi-processeurs à mémoire partagée," Univ. d'Orléans, rapport de DEA, Sept. 1990.

[MöW91] U. Möncke & R. Wilhelm, "Grammar Flow Analysis," in this volume, 1991.

[Par88] D. Parigot, "Transformation, évaluation incrémentale et optimisations des grammaires attribuées: le système FNC-2," Univ. de Paris-Sud, thèse, Orsay, May 1988.

[Par85] ———, "Un système interactif de trace des circularités dans une grammaire attribuée et optimisation du test de circularité," Univ. de Paris-Sud, rapport de DEA, Orsay, Sept. 1985.

[Rep84] T. Reps, *Generating Language-based Environments*, MIT Press, Cambridge, MA, 1984.

[Rii83] H. Riis-Nielson, "Computation Sequences: A Way to Characterize Subclasses of Attribute Grammars," *Acta Inform.* **19** (1983), 255–268.

[SwV91] S. D. Swierstra & H. H. Vogt, "Higher Order Attribute Grammars," in this volume, 1991.

[UDP82] J. Uhl, S. Drossopoulos, G. Persch, G. Goos, M. Daussmann, G. Winterstein & W. Kirchgäßner, *An Attributed Grammar for the Semantic Analysis of ADA*, Lect. Notes in Comp. Sci. #**139**, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[VoM82] A. O. Vooglaid & M. B. Méristé, "Abstract Attribute Grammars," *Progr. and Computer Software* **8** (Sept. 1982), 242–251.

[Yeh83] D. Yeh, "On Incremental Evaluation of Ordered Attributed Grammars," *BIT* **23** (1983), 308–320.