Are Attribute Grammars Used in Industry?

Josef Grosch

CoCoLab - Datenverarbeitung
Hagsfelder Allee 16, 76131 Karlsruhe, Germany
grosch@cocolab.de

Abstract

This paper answers two questions: Are attribute grammars used in industry? Are attribute grammars just an academic playground? The answers are given from the author's personal experience and perspective. A case study demonstrates the application of attribute grammars for real world projects such as a PL/I parser using the Cocktail Toolbox for compiler construction.

1. Introduction

Are attribute grammars used in industrial applications? Or are attribute grammars just an academic playground? I would like to answer these two questions based on my personal experience. I have been working with attribute grammars for around 17 years now. Around 10 years ago I started creating the Cocktail Toolbox [GE90] which contains among other tools for compiler construction the attribute grammar tool *ag*. Five years ago I founded a company named *CoCoLab* which stands for compiler compiler laboratory. The company develops and markets the Cocktail Toolbox as well as parsers generated with Cocktail. We also do project work in the area of compiler construction, program analysis, and programming languages.

My first and very spontaneous answers to the above two questions are: Yes, in both cases. Attribute grammars are used in industry and at the same time they can be regarded as academic playground. This does seem contradictory, doesn't it? Therefore let me explain in more detail the reasons for giving the above answers.

1.1. Why Attribute Grammars are not Used

I am observing a lack of education and knowledge about compiler construction in industry. When I am asking the participants of our trainings or the employees we meet in our projects then only few people have learned about compiler construction during their education. For many of them compiler construction has a bad reputation because of what and how they have learned about this topic. Even fewer people have a usable knowledge about compilers. Even fewer people know about the theory of attribute grammars. And even fewer people know how to use attribute grammars for solving practical problems.

Nevertheless, attribute grammars are used in industry. However, in many cases the people in industry do not know about this fact. They are running prefabricated subsystems constructed by external companies such as ours. These subsystems are for example parsers which use attribute grammar technology.

1.2. Attribute Grammars are Used in Industry

On the other hand, attribute grammars are used in industry because at least our company uses attribute grammar technology for solving compiler problems. Let me describe where and how we are using attribute grammars.

Our main area of business are parsers or front-ends for all languages such as for example COBOL, PL/I, C, C++, Java, and so on. Currently, these parsers are used for program analysis in re-engineering projects (year 2000, EURO) and in programming environments (e. g. SNiFF, Source Navigator).

In these parsers, attribute grammars are used for mainly two tasks: First, for tree construction and similar tasks performed during parsing. Second, for name analysis within semantic analysis which is performed on the abstract syntax tree.

Attribute grammars during parsing used for tree construction might seem rather trivial. In fact it is a simple technology. Nevertheless it is of great benefit. In the Cocktail Toolbox we use the same notation for attribute grammars evaluated during parsing and those evaluated based on syntax trees. The attributes have to be declared and they have names (as opposed to the notation `$1`, `$2`, `$3` known from Yacc). Therefore, the tools can check the computation rules for completeness which is of high value. It decreases development time and it increases the reliability of the products. This is more a matter of experience (look and feel) than something to argue about.

2. The Cocktail Toolbox

We make heavy use of attribute grammars in our parsers which are used in industry all over the world. We consider this technology to be very valuable and we use it frequently. Former problems such as lack of run time efficiency or insufficient memory for attribute storage do not exist any more, because today we have hardware with fast processors and plenty of main memory. Also, the time spent in parsing and attribute evaluation is often negligible compared to the time spent in database access.

Our approach for compiler construction is of course the use of the Cocktail Toolbox [GE90] which contains the following tools:

Rex	generator for lexical analyzers
Lark	LR(1) and LALR(2) parser generator with backtracking and predicates
Ell	LL(1) parser generator
Ast	tree management tool
Ag	generator for attribute evaluators
Puma	tree processor tool (transformation, pattern matching)
Reuse	library of reusable modules

The attribute grammar tool *ag* handles the following classes of attribute grammars:

OAG	ordered attribute grammars [Kas80]
WAG	well-formed attribute grammars [Gro92a]
HAG	higher order attribute grammars [VSK89, Vog93]

It supports tree-valued attributes and object-oriented attribute grammars [Gro90]. The latter mechanism provides single inheritance as well as multiple inheritance for attributes and for attribute computations. Also, remote attributes referenced by a tree-valued attribute can be computed and it is possible to evaluate attributes of graphs (with some restrictions). The generated evaluators are very run-time efficient because they are directly coded using recursive procedures. While the Cocktail Toolbox supports the generation of compilers in C, C++, and Modula-2 we use C as implementation language in most of the cases because of portability reasons.

The tool *ag* supports so-called copy rules which is a very helpful feature in practical applications.

A copy rule copies the value of an attribute to another attribute with the same name. These rules do not need to be written down because they are inserted automatically by the tool. Only at a few rules you assign values to an inherited attribute and perhaps hundreds of copy rules are generated. This saves writing down a huge number of rules and it leads to concise notation which concentrates on the essential computations.

3. Overview of Using Attribute Grammars

The following sections give an overview of our use of attribute grammars.

3.1. Tree Construction

In all our parsers we use an attribute grammar for tree construction. This attribute grammar is evaluated during parsing. In simple cases there is only one synthesized attribute per symbol referring to the subtree corresponding to the symbol. Sometimes additional attributes are used if some constructs are not mapped to tree nodes but to attributes of tree nodes or for additional tasks. Examples are parsers for Ada, C, C++, Fortran 90, JCL, HTML, XML, Delphi, Powerbuilder, and so on. The development of these attribute grammars takes between one day and one or two months. The time depends on the size and the complexity of the concrete grammar. Our largest grammar for example has more than 3000 grammar rules.

Especially valuable proved this technique lately during the development of our ANSI C++ parser in particular because of the complexity of the grammar. Then it is advantageous to use a tool that reports missing computations as well as superfluous computations. In other words, such a technique keeps the code clean and increases its reliability.

```
postfix_expression = <
        = primary_expression .

        = p:postfix_expression '[' e:expression ']'
        { type := type_op (p:type, karray);
          tree := msubscript_expr ('[':Position, p:tree, e:tree); } .

        = postfix_expression '(' expression_list ')'
        { type := type_op (postfix_expression:type, kfunction);
          tree := mcall_expr ('(':Position, postfix_expression:tree,
                ReverseTree (expression_list:tree)); } .

        = s:simple_type_specifier '(' expression_list ')'
        { type := get_type (s:tree);
          tree := (s:tree->specifier.next = mnospecifier (),
                mconstruct_expr ('(':Position, s:tree,
                ReverseTree (expression_list:tree))); } .

        = ...
```

Figure 1: Excerpt from C++ Grammar

The example in Fig. 1 presents a small excerpt from our C++ grammar. Two attributes are computed in these rules: The attribute *tree* is used for tree construction. The attribute *type* determines

the type of every expression. This has to be done during parsing because at a class member access (`e.m`, `e->m`) the name of the member `m` has to be looked up in the class of the expression `e`. Attribute computation rules are not given for the first rule. The missing copy rules are inserted automatically.

3.2. Name Analysis

In some of our parser we do name analysis with an attribute grammar. Examples are COBOL, OO-COBOL, PL/I, Java, Tcl/Tk, CICS, SQL. Most of the time we use a combination of techniques as will be described in more detail below. Typically, the implementation of name analysis takes one day even for huge languages such as COBOL or PL/I.

We observe that we often use attribute grammars for the distribution of context information in the syntax tree. Here, the copy rules are a big advantage. Examples are:

- the scope valid at the use of a name
- the kind of access of a variable (read or write)
- the section name for every paragraph (in COBOL).

So far we have described conventional applications of attribute grammars in the area of compiler construction. Some unconventional applications are the following:

3.3. Tree Transformation in PL/I

For PL/I the so-called `F(X)` problem has to solved. The notation `F(X)` can denote a function call or an array subscription depending on the declaration of `F`. PL/I allows "use before declare" and therefore the parser might not know the declaration of `F` while building the tree node for `F(X)`. Therefore, the parser maps this syntactic construct to a certain node type in the first place. Later, a tree transformation is applied which uses the results of name analysis and which transforms the concerned tree nodes. This transformation has been implemented with an attribute grammar in combination with a few routines written for the tree processor tool *puma* [Gro92b].

3.4. Validation of XML Documents

Another example is the validation of XML documents according to the so-called document type definition (DTD). The DTD is a restricted form of a context free grammar. The validation is based on the syntax tree which results from parsing. We compute the FIRST sets or director sets of the grammar rules using an attribute grammar. The validation of the document is again performed by a combination of an attribute grammar and a tree processor tool (*puma*).

3.5. Layout Algorithm

For a last example let me mention the use of an attribute grammar for the development of a layout algorithm. The tree management tool *ast* [Gro91] of the Cocktail Toolbox offers a graphical browser for trees and graphs. This browser determines a layout for the tree and displays nodes and edges as everybody would expect. The contents of the nodes consisting of its attributes can be inspected as well. It offers zooming and scrolling and so on. And it can efficiently handle huge trees with thousands of tree nodes. While the layout algorithm is generated specifically for every tree definition by the tool *ast* the graphical user interface is fixed. The graphical user interface is implemented with the package Tcl/Tk and therefore the browser is portable and available under Unix and Windows.

```
RULE

root    = tree .

tree    = <
   node = child1:tree   child2:tree child3:tree .   /* node with 3 children */
   list = next:tree REV child2:tree child3:tree .   /* list node */
   leaf = .                                         /* leaf node */
> .

MODULE compute_layout

DECLARE tree = [xin INHERITED]   [yin INHERITED]
               [x   OUTPUT]      [y   OUTPUT]
               [w   SYNTHESIZED] [h   SYNTHESIZED] .   /* width and height */

RULE

root    = { tree:xin := 0; tree:yin := 0; } .

tree    = { x := xin; y := yin; w := 0; h := 0; } .

leaf    = { x := xin; y := yin; w := 1; h := 1; } .

node    = { x := (child1:x + child3:x) / 2; y := yin;
            w := child1:w + child2:w + child3:w;
            h := Max (Max (child1:h, child2:h), child3:h) + 1;
            child1:yin := yin + 1; child1:xin := xin;
            child2:yin := yin + 1; child2:xin := xin + child1:w;
            child3:yin := yin + 1; child3:xin := xin + child1:w + child2:w;
          } .

list    = { x := xin; y := yin;
            w := Max (1 + child2:w + child3:w, next:w);
            h := Max (child2:h, child3:h) + next:h;
            next:yin := yin + Max (child2:h, child3:h); next:xin := xin;
            child2:yin := yin + 1; child2:xin := xin + 1;
            child3:yin := yin + 1; child3:xin := xin + 1 + child2:w;
          } .

END compute_layout
```

Figure 2: Attribute Grammar for Layout Algorithm

We used an attribute grammar for the specification and for prototyping of the layout algorithm of the browser (Fig. 2). The layout algorithm solves a simplified version of the problem. It is oriented towards syntax trees and therefore rather simple. Only trees with a fixed number of children have been considered: The node type `node` has three children called `child1`, `child2`, and `child3`. The node type `list` has three children, too. One of its children called `next` has the property `REVERSE` (`REV`) specifying that this node is element of a list. The elements of lists shall be displayed one underneath the other while nodes of type `node` are centered above their subtrees. The attribute grammar uses six attributes: The width (`w`) and height (`h`) of a subtree and its `x` and `y` coordinates. The attributes `xin` and `yin` are the coordinates which are propagated from the parent to the child nodes. This grammar has been checked by a tool for completeness and wellformedness. Once we did gain trust in it it was used as the base of the real implementation. In the real world a similar attribute computation is performed by an attribute evaluator generated specifically for every tree definition. Also, the algorithm has been extended to handle an arbitrary number of node types with arbitrary many children as well as graphs, even cyclic ones.

3.6. We Should Have Used an Attribute Grammar

We can also report a case were we regret not having used an attribute grammar. The project implemented a translator (to C) and an interpreter for a customer specific language. With the intention to gain experience in the implementation of semantic analysis without attribute grammars and to save the customer from learning about attribute grammars we used just the tree processor tool *puma* for semantic analysis. Looking back we have to say this decision was wrong. The use of an attribute grammar would have been better. First, it clearly structures the computations. Every attribute is a request for a computation and indicates its location. Second, the tool checks for completeness and the absence of cycles. This would have led to a clearer implementation as well as to a shorter development time.

4. Case Study

The following case study is taken from our PL/I system. The next sections will discuss our approach to name analysis in some more detail.

4.1. Name Analysis in PL/I

During many projects we have developed some kind of a standardized approach for name analysis. Name analysis uses a symbol table which is an ordered set of scopes. A scope describes a set of declared objects. Most of the time we implement the symbol table with the tree manager tool *ast* [Gro91]. In simple cases sets of objects are implemented by linked lists. If efficiency is necessary then the linked lists can be overlaid by hash-tables. The symbol table is constructed using the tree processor tool *puma* [Gro92b]. The scope information is distributed in the syntax tree by an attribute grammar. The attribute grammar also triggers the calls of the functions `Identify...` which are written in C and which perform the lookup in the symbol table.

The approach uses a combination of attribute grammars and other techniques. We often combine the following tools in order to solve the various subtasks of name analysis:

#	subtask	tool	name
1	define and manage data structure for symbol table	tree manager	*ast*
2	build symbol table	tree processor	*puma*
3	distribute scope information in the syntax tree	attribute grammar	*ag*
4	lookup in symbol table	search algorithm	C

Note, the subtasks 2 and 4 are controlled or triggered by the attribute grammar, too. However, the real computations are performed outside the attribute grammar.

The question of concern is: How do we know which tool or technique to use for which subtask and how to find out about the right combination?

4.2. Source Program

The language PL/I has nested scopes as shown by the example program in Fig. 3. The procedure `P` establishes a block and declares the variables `VAR1`, `VAR2`, and `VAR3`. The variable `VAR3` is a record variable with the fields `FIELD1`, `FIELD2`, and `FIELD3`. The procedure `P` contains a `BEGIN-END` block which forms a nested scope. This nested scope declares the variables `VAR2`, `VAR3`, and `VAR4` thus hiding some declarations from the surrounding scope.

```
 1   P: PROC;
 2
 3      DCL VAR1 PIC '99', VAR2 PIC CHAR(2);
 4      DCL 1 VAR3(60)EXT,
 5        2 FIELD1 FIXED DEC(9,2),
 6        2 FIELD2 FIXED DEC(9,2) INIT (0),
 7        2 FIELD3 FIXED DEC(5,3);
 8
 9      BEGIN;
10         DCL (VAR2, VAR3, VAR4) PIC '99';
11
12         VAR1 = VAR3 + VAR4;
13      END;
14   END;
```

Figure 3: Sample PL/I Program

4.3. Syntax Tree

The parser of the PL/I system translates the source program to an abstract syntax tree. The graphical representation of syntax trees, even for small examples, is always larger than a sheet of paper or a computer screen. Therefore Fig. 4 displays the tree for the `BEGIN-END` block, only. The nodes are labelled with the names of the node types truncated to fit into the rectangle. The nodes at the left-hand side represent the statements `DCL`, assignment, and `END`. These are displayed underneath each other because they are elements of a (statement-) list. The sub-window shows the attributes of the highlighted node. The first line contains the type of the node. The remaining lines describe the children and attributes of the node. Each line gives the name of an attribute or child and its value. Values marked with an asterisk denote children where the value is a reference to a subtree.

4.4. Symbol Table

A symbol table stores information about all objects declared in a program. The relevant aspects are a data structure and algorithms for building and accessing it. We prefer to use the tree manager tool *ast* for the description and administration of the data structure. The example in Fig. 5 specifies the symbol table for PL/I.

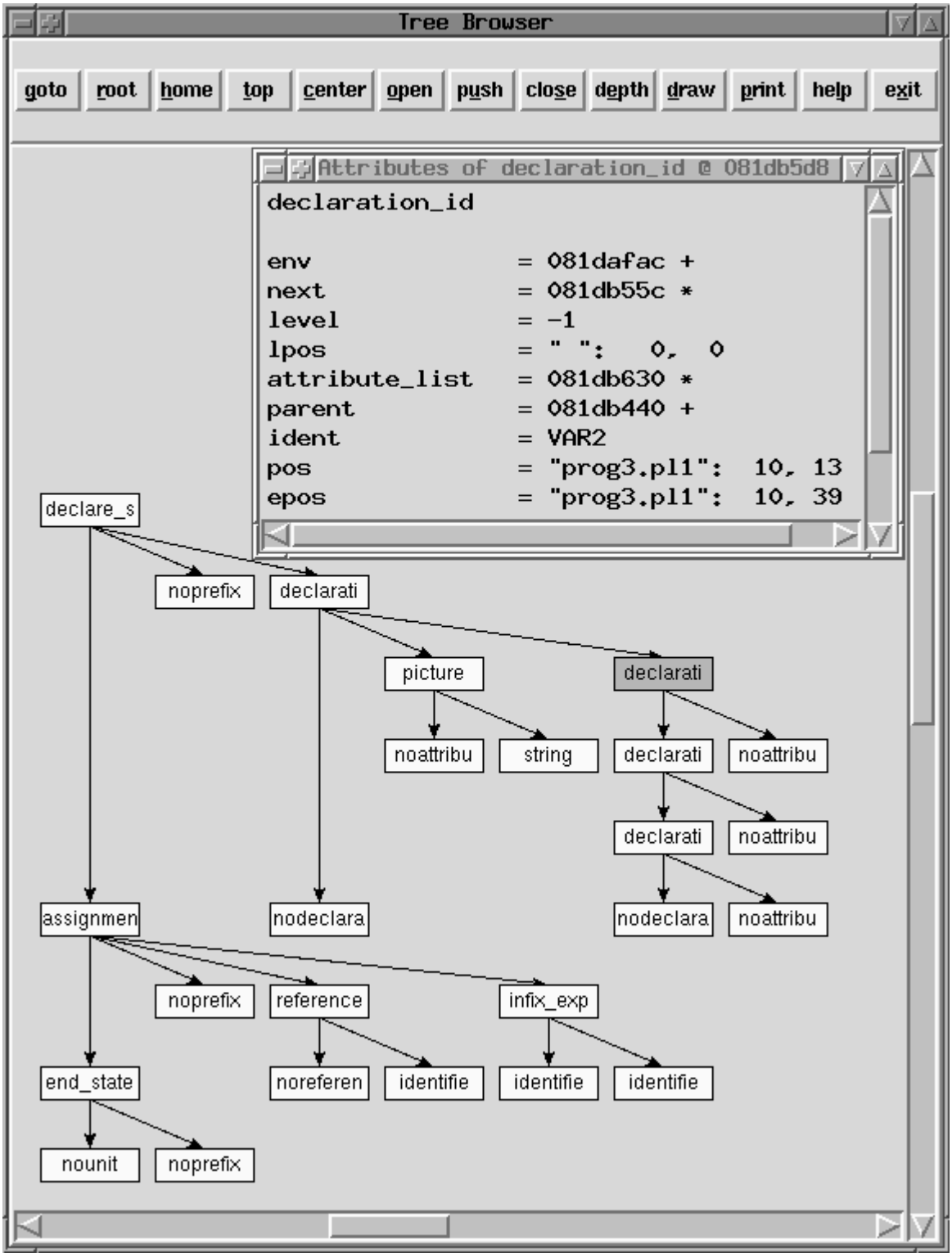

Figure 4: Syntax Tree - Excerpt

```
MODULE symbol_table

DECLARE         /* the attribute 'scope' gives access to the symbol table       */

allocation_list argument_list ... unit_list when_list
                = [scope : scope   INH]
                  .
declaration     = [parent: declaration] /* pointer to "parent declaration"      */
                  .                     /* for access of factored out attributes*/
declaration_id  = [attributes: tset  ]  /* bit_set describing all explicitly    */
                                        /* associated attributes,               */
                  [fields    : scope ]  /* list of fields of structures (scope) */
                  [lower     : declaration]/* pointer to lower (enclosing) level*/
                  .
RULE                                    /* node types for the symbol table      */

                                        /* a scope (or block) contains:         */
scope           = objects          IN   /* a set (list) of objects              */
                  scope            IN   /* a reference to a surrounding scope   */
                  [object   : Tree IN]  /* a node in the abstract syntax tree   */
                  [HashTable: tHashTable IN] /* pointer to hash-table           */
                  [HashSize : int  IN]  /* size of hash-table                   */
                  [Set      : tSet   ]
                  .
objects         = <                     /* an object is described by:           */
   object       = [object: Tree    IN]  /* a node in the abstract syntax tree   */
                  [ident : tIdent  IN]  /* an identifier                        */
                  [is_formal:rbool IN]  /* whether it is a formal parameter     */
                  next   : objects IN REV /* a next object                      */
                  [collision: object ]  /* collision chain for hash-table       */
                  <
      object_implicit   =               /* for implicitly declared objects      */
                  [Attribute: tkeyword] /* implicit attribute                   */
                  .
   > .
   noobject     = .                     /* the end of a list of objects         */
> .
                       /* attribute grammar to construct the symbol table      */

root            = { procedure:scope := mScope (get_objects (procedure, nnoobject),
                                      NoTree, SELF); } .
procedure       = { unit_list:scope := mScope (get_objects (unit_list, nnoobject),
                                      scope, SELF);
                    procedure_statement:scope := unit_list:scope; } .
begin_block     = { unit_list:scope := mScope (get_objects (unit_list, nnoobject),
                                      scope, SELF); } .

END symbol_table
```

Figure 5: Specification of Symbol Table

For every scope a node of type `scope` is created. Its three most relevant components are `objects`, `scope`, and `object`. The child `objects` points to a list of objects declared in this scope. The "child" `scope` contains a reference to a node of type `scope` describing the surrounding scope. The attribute `object` refers to the node in the abstract syntax tree which represents the scope (e. g. block or procedure). The other components implement a hash-table which is used for fast access in the list of objects instead of linear search.

For every declared object a node of type `object` is created and linked to its scope node. Its three most relevant components are `object`, `ident`, and `next`. The attribute `object` refers to the node in the abstract syntax tree which represents the declared object. The attribute `ident` specifies the name of the declared object. The child `next` refers to the next declared object in the list.

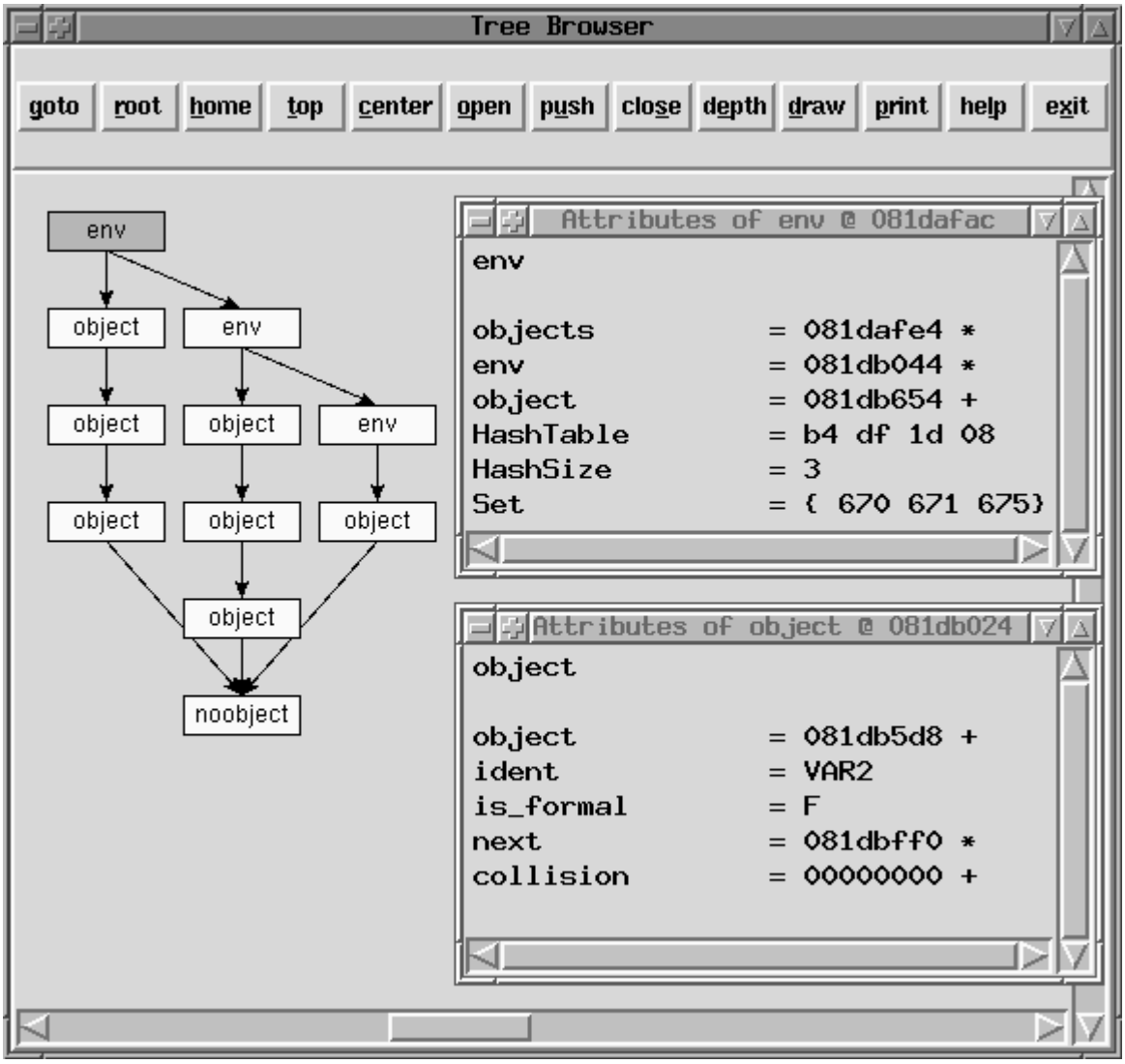

Figure 6: Symbol Table

Fig. 6 displays the symbol table for the example source program. The top-most node of type `scope` describes the inner-most scope of the `BEGIN-END` block. The next node of type `scope` describes the procedure scope. The last node of type `scope` describes the global scope containing the declaration

of the procedure P. The upper attribute window shows the contents of the top-most node of type scope. The lower attribute window shows the contents of the node of type object describing the first declared object in the inner-most scope. The references from the symbol table to the syntax tree or vice versa are not displayed in the graphical browser because they are specified to be attributes instead of children. Nevertheless the corresponding information can be displayed in a separate window by clicking on the attributes marked with the character +.

In order to be able to access the symbol table the attribute scope is distributed to almost all nodes of the syntax. This attribute is declared in the first lines of the module symbol_table (Fig. 5). The last three lines of this module specify the computation of this attribute at the nodes that describe scopes. At all other nodes this attribute is copied by automatically inserted copy rules.

For PL/I we build the symbol table by traversing the relevant parts of the syntax tree with routines such as get_objects which are implemented using the tree processor tool *puma* [Gro92b]. During the traversal the nodes of type object are created and added to the lists of objects. We do not show these routines because this is outside of our topic. The function mScope creates a node of type scope and adds a hash-table to the linear list of objects.

4.5. Symbol Table Lookup

The lookup in the symbol table is accomplished by a few functions written in C. The function IdentifyVar shown in Fig. 7 is an example. It takes two arguments specifying a name (Ident) and a scope and returns either a pointer to a node of type object describing the object found or the value nnoobject in case of failure. The function uses hashing for fast lookup. In PL/I names of fields do not have to be fully qualified. This is taken into account by the function IdentifyUnqualified.

```
tTree IdentifyVar (register tIdent Ident, tscope Scope)
{
   while (Scope != NoTree) {
      register tTree Object =
         Scope->scope.HashTable [Ident % Scope->scope.HashSize];
      if (IsElement (Ident, & Scope->scope.Set)) {
         while (Object != NoTree)       /* pass 1: search on current level */
            if (Object->object.ident == Ident) return Object;
            else Object = Object->object.collision;
                                        /* pass 2: search on all levels */
         Object = IdentifyUnqualified (Ident, Scope);
         if (Object != nnoobject) return Object;
      }
      Scope = Scope->scope.scope;       /* search in surrounding block */
   }
   return nnoobject;
}
```

Figure 7: C Function for Symbol Table Lookup

4.6. Name Identification

The result of name analysis is the computation of an attribute called object at every node of type identifier, structure_qualification, or subscription. This attribute is declared at the

```
MODULE name_analysis

DECLARE
   value_reference = [object       : object] . /* pointer to symbol table      */
   value_reference = [dimension    : short ] . /* number of dimensions         */
   argument_list   = [dimension    : short ] . /* number of dimensions         */

RULE                                /* compute object and dimension attributes */

value_reference         = { object      := nnoobject;
                            dimension   := 0; } .

function_reference      = { object      := nnoobject; /* result is unknown */
                            dimension   := 0; } .

locator_qualification   = { object      := IdentifyVar (ident, scope);
                            dimension   := no_of_dimensions (object); } .

subscription            = { object              /* solve F(X) problem */
                            dimension   := {
                               if (value_reference:dimension == 0 &&
                                   is_entry (value_reference:object)) {
                                  object    = nnoobject; /* result is unknown */
                                  dimension = 0;
                                  trafo_to_function_ref (SELF);
                               } else if (value_reference->Kind == kidentifier &&
                                   value_reference:object->Kind == kobject &&
                                   has_builtin (value_reference:object)) {
                                  object = nnoobject; /* result is unknown */
                                  dimension = 0;
                                  trafo_to_builtin_ref (SELF);
                               } else if (value_reference:dimension > 0 &&
                                          subscript_list:dimension  > 0) {
                                  object    = value_reference:object;
                                  dimension = value_reference:dimension -
                                              subscript_list:dimension;
                               } else if (value_reference->Kind == kidentifier &&
                                  is_builtin (value_reference->identifier.ident)){
                                  object = nnoobject; /* result is unknown */
                                  dimension = 0;
                                  trafo_to_builtin_ref (SELF);
                                  value_reference:object = nnoobject;
                               } else {
                                  object    = value_reference:object;
                                  dimension = value_reference:dimension -
                                              subscript_list:dimension;
                               }
                            };
                          } .
```

Figure 8: Attribute Grammar for Name Analysis - Part 1

```
structure_qualification = { object      := value_reference:object != nnoobject &&
                                           value_reference:object->\object.\object->
                                           Kind == kdeclaration_id &&
                                           value_reference:object->\object.\object->
                                           declaration_id.fields != NoTree ?
                                                   /* symbol table lookup */
                              IdentifyField (ident, value_reference:object->
                              \object.\object->declaration_id.fields) : nnoobject;

                            dimension   := no_of_dimensions (object);

                            => {                /* solve F(X) problem */
                               if (value_reference:dimension == 0 &&
                                     is_entry (value_reference:object))
                               value_reference = DEP (
                                  trafo_to_function_ref_of_no_args
                                     (value_reference), dimension); };
                          } .

identifier              = {                       /* symbol table lookup */
                            object      := IdentifyVar (ident, scope);
                            dimension   := no_of_dimensions (object);
                                                  /* handle implicit declaration */
                            CHECK object != nnoobject ||
                                  is_subscripted && is_builtin (ident) => {
                               tTree e = get_ext_procedure (scope);
                               object = mobject_implicit (SELF, ident, rfalse,
                                                          NoTree);
                               if (is_subscripted)
                                  object->object_implicit.Attribute = k_entry;
                               if (e) ExtendScope (object, e->procedure.\scope);
                            };
                          } .

END name_analysis
```

Figure 9: Attribute Grammar for Name Analysis - Part 2

beginning of the module `name_analysis` (Fig. 8) for the super class `value_reference` of the mentioned node types. In the simplest case the following computation suffices:

```
identifier              = { object      := IdentifyVar (ident, scope); } .
```

The language PL/I is not simple and therefore the real computations are more complicated as shown by the excerpt of the attribute grammar in Figures 8 and 9 which will not be explained in all details.

If a name is not declared at the node type `identifier` then it is declared implicitly using the functions `mobject_implicit` and `ExtendScope`. At the node types `subscription` and `structure_qualification` the `F(X)` problem is solved. Depending on the specific case the functions `trafo_to_function_ref` or `trafo_to_builtin_ref` are called which modify the tree by replacing nodes of the types `subscription` or `identifier` by nodes of type `function_reference` or `builtin_function_reference`.

5. Conclusion

We showed that attribute grammars are used in industry. However, based on our knowledge, attribute grammars are rarely used in industry. Often people from industry do not know that they are using attribute grammars because they are running prefabricated programs from external companies. At the same time we are observing a lack of knowledge and education about attribute grammars. This holds for the theory of attribute grammars as well as for tools that support the use of attribute grammars.

The question is how to improve the use of attribute grammars? What is needed in order to increase the dissemination of attribute grammars in industry? In my humble opinion the following could be improved:

- more education in compiler construction with emphasis on practical applications
- simple and practical tools for attribute grammars
- education about the practical use of attribute grammars
- tutorials, user's guides, and other documentation that present the matter in a good didactic style and which are easy to understand
- simple examples that support the style of learning by example
- knowledge about what attribute grammars can do and what they can not do
- knowledge about the right combination of attribute grammar tools and other tools

Bibliography

[GE90] Josef Grosch and Helmut Emmelmann. A tool box for compiler construction. In *Proceedings of the Third International Workshop on Compiler Compilers*, volume 477 of *Lecture Notes in Computer Science*, pages 106–115. Springer Verlag, Heidelberg, New York, 1990.

[Gro90] Josef Grosch. Object-oriented attribute grammars. In A. E. Harmanci and E. Gelenbe, editors, *Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, pages 807–816, October 1990. Cappadocia, Nevsehir, Turkey.

[Gro91] Josef Grosch. Tool support for data structures. *Structured Programming*, 12:31–38, 1991.

[Gro92a] Josef Grosch. Efficient evaluation of well-formed attribute grammars and beyond. Technical report, CoCoLab - Datenverarbeitung, Karlsruhe, 1992.

[Gro92b] Josef Grosch. Transformation of attributed trees using pattern matching. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, Heidelberg, New York, October 1992. Proceedings of the 4. International Conference, CC'92, Paderborn Germany.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[Vog93] H. H. Vogt. Higher order attribute grammars. Technical report, PhD Thesis, University of Utrecht, February 1993.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Notices*, 24(7):131–145, July 1989.