COMPILER CONSTRUCTION USING ATTRIBUTE GRAMMARS

Kai Koskimies+, Kari-Jouko Räihä, Matti Sarjakoski*

Department of Computer Science, University of Helsinki
Tukholmankatu 2, SF-00250 Helsinki 25, Finland

Abstract

The adequacy of attribute grammars as a compiler writing tool is studied on the basis of the experiences on attribute grammars for Pascal and a subset of Euclid. A qualitative assessment of the grammars shows that the compiler oriented view in the design of an attribute grammar tends to make the grammar hard to understand. A design discipline is proposed to improve the comprehensibility of the grammar. Quantitative measurements of the automatically generated compilers suggest that an efficient compiler can be produced from an attribute grammar. To achieve this, a carefully optimized implementation of the compiler-compiler is required.

1. Introduction

Attribute grammars were initially developed as a means of defining the semantics of programming languages. During the last decade they have been increasingly used as a specification language of compilers. Recent projects aiming at the construction of compilers for Ada (e.g. [6,18]) indicate that automatic generators of semantic evaluators are becoming part of the compiler writer's toolbox, which until recently has not contained much more than a parser generator.

The purpose of the present paper is to study the usability of attribute grammars as a practical tool for compiler construction. The study is based on attribute grammars written for Pascal [7] and C-Euclid [10,11]. C-Euclid (C for core) is a subset of Euclid [13], a derivative of Pascal designed for expressing verifiable systems programs. It should not be confused with Euclid-C [3].

+) The work of this author was supported by the Emil Aaltonen Foundation.

*) The work of this author was supported by the Academy of Finland.

In our study we shall regard attribute grammars as a special purpose language for writing compilers. Like any programming language, attribute grammars can be viewed from two opposite standpoints. The user appreciates properties that make the language a natural tool for writing the compiler, while the implementor of the grammar is interested in translating the grammar into a conventional, efficient program. In this paper, both of these aspects will be studied from a practical point of view.

The literature contains several examples of how various features of programming languages can be described using attribute grammars. However, the author of a complete grammar for a substantial language cannot find much advice on how to proceed. The general design principles of the attribute grammars for Pascal and C-Euclid are discussed in Section 2. Moreover, based on these experiences we propose an object oriented approach to writing attribute grammars in order to improve their readability.

Similarly, the implementation techniques of attribute grammars have been studied extensively, but little is known of the efficiency of actual compilers based on attribute grammars. The situation where the resulting compiler is produced manually on the basis of the attribute grammar is discussed in [5,18]. Some measurements of automatically generated semantic evaluators are given in [2,8].

Section 3 contains an analysis of the compilers for Pascal and C-Euclid generated using the compiler writing system HLP [17]. The compilers are compared with a conventional hand-written Pascal compiler.

HLP provides a separate metalanguage for describing the code generation phase of the compiler. Since our objective is to evaluate the applicability of attribute grammars for compiler construction, we have excluded the code generation phase from the comparisons. Thus our notion of a "compiler" is closer to a "compiler front-end": it means a processor that performs lexical analysis, parsing, and semantic analysis.

We conclude by summarizing the main results.

2. The design of attribute grammars

The attribute grammars for Pascal and C-Euclid

The grammars for Pascal and C-Euclid follow the same design principles and style. Global attributes are not used for storing space-critical information. The grammars are pure attribute grammars based on local inherited and synthesized attributes.

Both Pascal and (C-)Euclid are designed to be amenable for one-pass compilation. Accordingly, the attribute grammars are written so that they need only one evaluation pass. In general, the inherited attributes of nonterminal N describe the information that depends on the left context of N and that is needed in processing the string generated by N. Correspondingly, the synthesized attributes of N describe the information that is collected from the string generated by N and that is needed in processing the right context of N. Hence the information flows strictly from left to right. The resulting grammar is L-attributed [14] and evaluable in a single pass from left to right.

Note that this approach has a connection with the recursive descent method in which the evaluation of every nonterminal is written as a procedure. The inherited attributes represent the input parameters of the procedure for the corresponding nonterminal; the synthesized attributes represent the output parameters. The occurrence of a nonterminal in the derivation tree represents the activation of the corresponding procedure.

The problem of describing a traditional global symbol table with local attributes is solved by storing the possible states of the symbol table in local attributes env_i (inherited) and env_s (synthesized). Each instance of env_i gives the state of the symbol table in a one-pass compiler before processing the string generated by the associated nonterminal. Correspondingly, each instance of env_s gives the state of the symbol table after processing the string generated by the associated nonterminal.

One may imagine that there is a global symbol table whose successive states can be found travelling through the entire derivation tree in a top-down, left-to-right order:

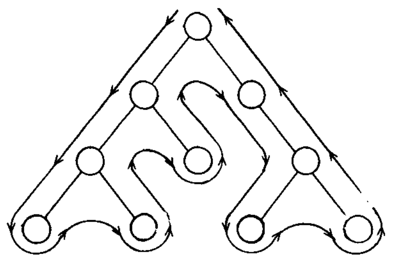

Figure 1. The symbol table attributes.

At each node, an arrow head pointing downwards represents an instance of env_i while an arrow head pointing upwards represents an instance of env_s.

This general idea of using local symbol table attributes is in accordance with the space-efficicient, one-pass subclass of attribute grammars formulated in [12]. Consequently, the evaluator is able to use the same global space for all the instances of env_i and env_s, instead of allocating a separate space for different states of the symbol table.

The symbol table attributes have an internal structure similar to that of a symbol table in a hand-written one-pass compiler. However, this internal structure cannot be seen at the attribute level, where the symbol table attributes are always manipulated by operations affecting the attributes as a whole. This kind of data abstraction improves the comprehensibility of the grammar: it separates the abstract concept of an identifier environment (used in semantic rules, at the attribute level) from the complex structure of the symbol table (used in the implementation of the semantic functions). The usefulness of data abstractions in attribute grammars is also pointed out in [5].

There is a special class of attributes whose values are pointers to the entries in the symbol table; the attributes map various objects of the source program to the symbol table. The pointer attributes appear either in trivial copy rules or as parameters of the symbol table operations. Hence these attributes do not destroy the symbol table abstraction.

Statistics

Some characteristic measurements of the attribute grammars for Pascal and C-Euclid are given in Table 1.

	Pascal	C-Euclid
productions	286	227
nonterminals	137	108
semantic rules	1012	1081
copy rules	765	718
semantic functions	67	68
inherited attributes	10	18
synthesized attributes	19	37
attributes per nonterminal (average)	3.8	5.1

Table 1. Statistics of the attribute grammars.

The relatively small average number of attributes per nonterminal is explained partly by the heavy use of the symbol table attributes. Often a large amount of information is represented indirectly by a single attribute whose value is a pointer to an entry in the symbol table. This feature obviously reduces the space requirements of the evaluator.

Discussion

Our exprerience with the grammars for Pascal and C-Euclid indicate that the attribute formalism is, in general, an adequate language for compiler writing. The one-pass design principles outlined above allowed a straightforward development of the semantic analysis phases.

The most unsatisfactory point of the grammars is that they are hard to read and understand as such: they are far from being self-documenting. This unfortunate property seems to be shared by many other attribute grammars, too.

There may be several reasons for the poor readability of an attribute grammar. One is probably the simple fact that attribute grammars have a graph-like structure that seems difficult to grasp from a linear form. However, we argue that the main reason is the compiler oriented approach that encourages a functional view in the design: semantic rules are written in the (supposed) order of evaluation, information is transferred in the derivation tree to the "computation points", attributes are introduced to support some compiler actions. This kind of view is not in harmony with the basic declarative nature of an attribute grammar. The description will be hard to understand because it is neither a conventional compiler nor a declarative definition of semantics, but a strange mixture of both.

An indication of the compiler oriented approach is the central role of productions and semantic rules in a grammar. This central role is reflected in many features of a grammar. In the Pascal and C-Euclid grammars semantic functions often include complex combined operations. Such functions do not describe relations between attributes associated with nonterminals; rather, they describe actions associated with productions. Another production oriented feature is the use of conditions (e.g. [15]) to specify the static semantics: these conditions do not define the values of attributes but values that are associated with productions.

To achieve a descriptive, declarative grammar the designer should lay emphasis on nonterminals and attributes, rather than on productions and semantic rules. That is, the grammar designer should start by considering the objects represented by nonterminals, not by considering actions associated with productions. Hence the primary aim should be to describe the nonterminals with attributes.

An example

Let us illustrate this point with the following small example taken from the grammar for C-Euclid. Consider the productions

(1) TYPE = ARRAY_TYPE

(2) ARRAY_TYPE = array INDEX_TYPE of COMPONENT_TYPE

for which the semantic rules are given graphically in Figure 2. For each nonterminal box, the inherited attributes are on the left and the synthesized attributes are on the right. The function "allocateUnnamedType" allocates a symbol table record for an unnamed type and returns a pointer to the allocated record ($pntr_i$ of ARRAY_TYPE). If TYPE represents a named type ($context_i$ = type declaration) the descriptor of the type is already in env_i of TYPE, and the function simply passes down the pointer to the descriptor ($pntr_i$ of TYPE). Hence, in any case $pntr_i$ of ARRAY_TYPE points to the descriptor of the corresponding array type. The purpose of the function "completeArrayType" is to mark the descriptor completed. The information of the index type and of the component type will be

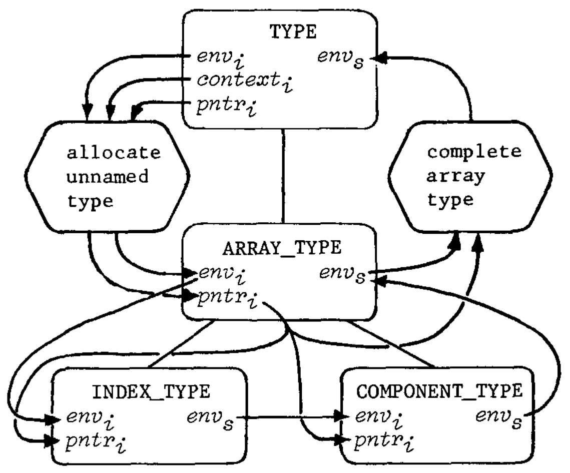

Figure 2. A fragment of the C-Euclid grammar.

stored in the descriptor while processing the subtrees of INDEX_TYPE and COMPONENT_TYPE, respectively, using the pointer $pntr_i$.

The role of the attributes in this example is best characterized as input and output information for the processing of the corresponding nonterminal. The descriptive value such attributes may possess is purely accidental.

Let us redesign this fragment of the grammar by considering first the nonterminals. (For the sake of clarity, we ignore the symbol table attributes.) The nonterminal TYPE may generate any type, hence we introduce an attribute, say *type*, whose value is the description of a type (or a pointer to such a description). The nonterminal ARRAY_TYPE always produces a type that is completely determined by two types, the index type and the component type. Correspondingly, this nonterminal should be associated with two attributes, say *indexType* and *componentType*, both describing a type. The nonterminals INDEX_TYPE and COMPONENT_TYPE are similar to the nonterminal TYPE in that we do not know which types the nonterminals produce. Hence the attribute *type* is associated with these nonterminals, too.

The semantic rules should be now imposed by these attributes. First we observe that in production (2) the value of *indexType* at ARRAY_TYPE is the same as the value of *type* at INDEX_TYPE. Similarly, the value of *componentType* at ARRAY_TYPE is the same as the value of *type* at COMPONENT_TYPE. Further, we note that in production (1) the value of *type* at TYPE is a description of an array type composed of the types described by *indexType* and *componentType* of ARRAY_TYPE. Hence we introduce a function, say "arrayType", that reduces (produces) the descriptions of the index type and the component type to (from) the description of the array type. We are not interested in the direction of the evaluation, any more than the writer of a context-free grammar is interested in the direction of parsing.

The relations of the attributes are summarized in Figure 3.

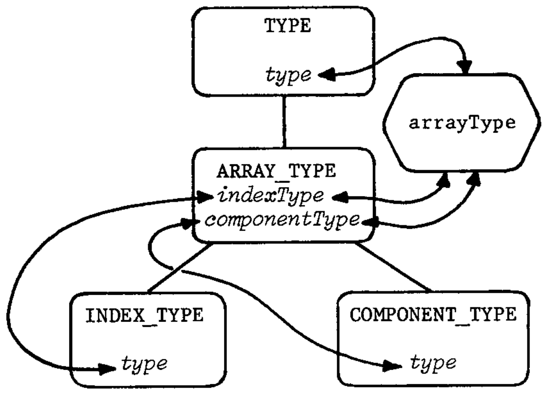

Figure 3. Relations of attributes.

The arrows are double-headed because so far we assume nothing of the evaluation. However, since the information flows bottom-up these attributes will all be synthesized in the actual attribute grammar.

The revised grammar differs notably from the original. There is no separate function for the unnamed case, the type descriptor is always created by "arrayType". Further, there are no inherited attributes (except for the symbol table): type definitions can be processed using only synthesized information and the symbol table, very much like expressions. The important point, however, is that the attributes describe exactly those nonterminals they are associated with. Yet, from a practical point of view the revised form is as adequate as the original one.

A design discipline

A serious drawback of attribute grammars is the lack of practical design methodologies. The small case study presented above demonstrates the advantages of a nonterminal oriented design discipline that can be a first step towards such a methodology.

To be more precise, we suggest that the design should be divided into three successive phases that we call nonterminal analysis, production analysis, and evaluation analysis. The nonterminal analysis is the most important phase that fixes the nature of the grammar. In this phase each nonterminal is considered, and the properties describing the nonterminal are determined. These properties are represented as attributes.

In the production analysis each production is considered separately, and the relations between the values of the attributes associated with the nonterminals are specified. The nonterminal analysis and the production analysis are independent of the evaluation method.

The evaluation analysis is a technical phase that transforms the "pseudogrammar" developed in the first two phases into a concrete, evaluable grammar. Hence this phase corresponds to the coding phase of conventional programming. If necessary, the domains of attributes are specified and new technical attributes are introduced. This phase obviously depends on the available evaluation method. Attributes may be classified in different ways (e.g. synthesized, inherited) to make the evaluation analysis easier.

3. Efficiency of the compilers

The compilers produced by HLP use the LALR(1) parsing method. The parse tree is constructed explicitly, and as many synthesized attributes as possible are evaluated during parsing. The rest of the attributes are evaluated in several depth-first traversals through the parse tree made by an alternating semantic evaluator [9].

It should be emphasized that the primary motivation in the construction of the attribute grammars for Pascal and C-Euclid was to study the static semantics of the languages. Consequently, clarity and simplicity were the main design principles in coding the semantic functions; efficiency played no role. As an example, in both grammars symbol table access is based on linear search. This reason alone puts the automatically generated compilers into an unfavorable position in comparison with the conventional, hand-written, carefully tuned compilers.

Time

The most common estimate of the efficiency of a compiler is its compilation speed. A medium-sized program with 1040 lines and 4720 lexical tokens takes 10.8 seconds to analyze with the compiler front-end generated by HLP. This means a speed of 5800 lines per minute on the Burroughs B7800.

However, such figures are not very informative as such. It is more interesting to compare the performance of automatically generated compilers with conventional compilers. The relative speeds of the Pascal and C-Euclid compilers are shown in Figure 4 as a function of the number of lexical tokens in the program. Figure 4 also contains measurements of the speed of a hand-written Pascal compiler for the B7800, called Pascal/HB [4]. To make the results comparable, we have excluded code generation from the Pascal/HB compiler, too.

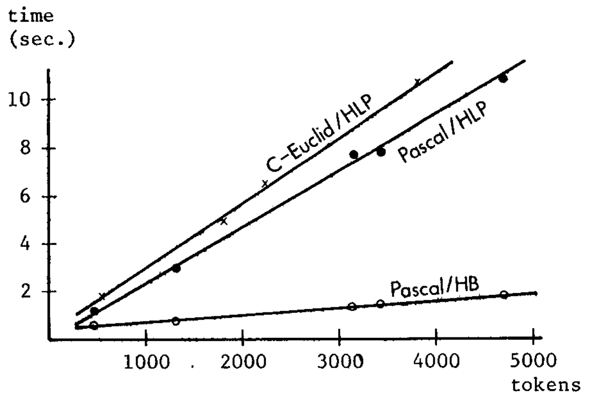

Figure 4. Time requirements.

The degree of efficiency that one requires from a compiler depends on its intended use. For a language designer who writes the attribute grammar to be able to test various constructs in practice, the speed of the automatically generated compilers is more than adequate. In a production environment, the speed of the Pascal/HLP and C-Euclid/HLP compilers seems sufficient at least for small and medium-sized programs. Thus the present trend towards small, separately compilable modules should increase the usability of automatically generated compilers.

The fact that Pascal/HB is faster than Pascal/HLP came as no surprise. Perhaps the main reason is that although the Pascal grammar needs only one evaluation pass, the resulting compiler is essentially a two-pass compiler. During the first pass the program is parsed and an internal representation, the parse tree, is constructed. The second pass consists of a traversal of the parse tree. Both the construction and the traversal of the tree are features that can be omitted in the hand-written, pure one-pass compiler.

Thus we expect that the speed of automatically generated compilers would be much closer to the speed of hand-written compilers for languages that require multi-pass compilation, or if the compilers perform advanced optimizations. In the one-pass case, the efficiency difference could be cut down by treating one-pass compilers as a special case in the compiler writing system: when the system finds out that all the attributes can be evaluated during parsing, it can omit the construction of the parse tree. To be fully useful, this feature probably requires the use of a top-down parsing technique, since the possibilities to evaluate inherited attributes during bottom-up parsing are limited.

To summarize, we were quite satisfied with the speed of the generated compilers. Considering the advantages of automatic compiler generation, even a slower speed could have been satisfactory.

Space

Skeptiks often argue that compilers based on attribute grammars cannot be practical since they require too much space, e.g. for storing the parse tree. On the other hand, advocates of attribute grammars sometimes claim that space is no problem in a virtual storage environment. Our results support neither of these extreme views.

In the first implementation of HLP [17] the space problem was left to the user by encouraging the use of global attributes. For the Pascal and C-Euclid grammars which do not use global attributes it soon became evident that the virtual storage of B7800 could not handle the problem by itself. For instance, it turned out to be impossible to use the C-Euclid compiler for programs that were longer than 40 lines without blocking almost all other use of the system.

To overcome this problem, a new space allocation technique for attributes was implemented in HLP [16,19]. The idea is to keep track of the liveness of attribute instances and to reuse space that holds a dead value. The method is dynamic; thus we expected a slower compilation speed as the price of the decreased demand of space. However, the need for the storage management operations performed by the underlying operating system decreased so drastically that the speed of the compilers was actually improved!

Tables 2 and 3 show the maximum amount of workspace required for storing the parse tree and the attribute values in the Pascal/HLP and C-Euclid/HLP compilers.

Tokens in input	Lines in input	Nodes in parse tree	Attribute instances	Workspace (Kbytes)
269	51	451	1414	20
1283	308	1954	6733	70
3433	730	5439	19063	185
4716	1038	7394	25798	251

Table 2. Space requirements of Pascal/HLP.

Tokens in input	Lines in input	Nodes in parse tree	Attribute instances	Workspace (Kbytes)
271	55	506	2101	25
1771	319	3241	13668	124
3845	713	7058	29663	262
7691	1427	14115	59326	519

Table 3. Space requirements of C-Euclid/HLP.

The parse trees constructed by HLP do not contain all the nodes used in the derivation of the source string. Nodes created by unit reductions with trivial (copying) semantics are automatically eliminated. As a result, the size of the parse tree is decreased by 20 - 25 percents. Similar results are reported in [2].

Even after this optimization the parse tree grows very fast. Thus it is important that the nodes are stored compactly. For the Pascal and C-Euclid grammars a simple left-to-right evaluation pass is sufficient. Since one-pass evaluation is not treated as a special case by HLP, the parse tree will in this case contain redundant information that is not used by the compilers. For instance, each node is now linked both to its right and left neighbours to facilitate alternating passes over the parse tree. Nodes also contain information on the corresponding location in the source program to make possible error messages more accurate. By allowing a small increase in the processing times for erroneous programs, it would be sufficient to store this information only in the leaves.

Our results are remarkably similar with those reported in [2]. To analyze a program with 9660 tokens, their Pascal front-end needs 475 Kbytes for storing the parse tree and attribute values. Our measurements show that for a program of that size, the Pascal/HLP compiler requires about 535 Kbytes without the compactifying optimizations suggested above.

Workspace is not the only space consumer in a compiler: the code has to be stored, too. The sizes of the C-Euclid/HLP, Pascal/HLP and Pascal/HB compilers are 310, 300 and 100 Kbytes, respectively. One of the reasons for the larger size of

HLP-compilers is a design principle of HLP: it should generate readable compilers. This tends to make the compilers quite large, since every semantic rule is directly encoded in B7800 Extended Algol. In particular, almost 50 percents of the code in the C-Euclid compiler is used to control semantic evaluation. This suggests that table-driven techniques commonly used for implementing parsers might be a better alternative for the semantic analyzer, too.

We have given in Tables 2 and 3 the maximal space requirements, which are independent of the computer and its operating system. However, in a virtual storage environment (like B7800) all of the workspace does not have to be in core simultaneously. Likewise, only part of the program code is needed at any time. Therefore average core usage or memory integral could be a more realistic, albeit implementation dependent, measure when virtual storage is used.

As an example, for a program with 4716 tokens the average core usage (both for code and data) of the Pascal/HLP and Pascal/HB compilers is 306 and 90 Kbytes, respectively. For a program with 3845 tokens, the C-Euclid/HLP compiler uses 408 Kbytes on the average.

Again, the hand-written compiler is more efficient than the automatically generated compilers. This is quite natural, since HLP produces compilers that construct an internal representation of the source program (the parse tree) not required by the one-pass Pascal/HB compiler. Comparisons with multi-pass compilers (e.g. optimizing compilers) would shorten the efficiency gap. Moreover, if one-pass evaluators were treated separately by HLP, the construction of the parse tree could be omitted. However, even now the space required by the tree is not a hindrance to the use of the generated compilers.

Finally, note that the results were obtained without code generation. If code generation is performed in a separate pass after semantic analysis, all of the attributes used during the final pass should be retained after semantic analysis. In this case code generation should be designed so that it does not depend on large space-consuming attributes. Another way to circumvent the problem is to integrate code generation with semantic analysis.

4. Conclusions

Based on the experiences with the attribute grammars for Pascal and C-Euclid, we have studied attribute grammars as a practical tool for compiler production.

From the user's point of view, a serious drawback of attribute grammars is the lack of adequate design methodologies. When the goal is a practical compiler, the grammar designer easily adopts a functional view that is not in harmony with the declarative nature of attribute grammars. The design principles followed in the Pascal and C-Euclid grammars represent such a view; in fact, they closely correspond to the recursive descent method. A functional view that conflicts with the nature of the description tool tends to make an attribute grammar hard to understand.

A three-phase design discipline that emphasizes the role of nonterminals is suggested. This discipline is based on the idea that a readable attribute grammar is a collection of the descriptions of nonterminals. Although loosely formulated, this discipline can be the basis of a practical design methodology for attribute grammars.

Measurements of the compilers produced automatically from the attribute grammars indicate that the performance of the compilers is reasonable. The efficiency of a comparative hand-written compiler was not achieved (except for very small programs), but the results show that practical compilers can be produced automatically from attribute grammars.

Let us conclude with a citation from [1]: "... The automatic implementation of attribute and affix grammars can be slow ... The extent to which attribute grammars can be used in practical compiler-compilers is still to be determined." We fully agree with the first statement: straightforward automatic implementations of attribute grammars can indeed be inefficient. However, <u>if</u> attribute grammars are used in a disciplined manner, and <u>if</u> they are implemented carefully, our experiences suggest that they are a useful and efficient tool in compiler construction.

References

1. A.V.Aho, Translator Writing Systems: Where Do They Now Stand? Computer 13 (Aug. 1980), 9-14.
2. B.Asbrock, U.Kastens and E.Zimmermann, Generating an Efficient Compiler Front-End. Bericht Nr. 17/81, Fakultät für Informatik, Universität Karlsruhe, 1981.
3. J.R.Cordy and R.C.Holt, Specification of Concurrent Euclid (Preliminary Version). Technical Report CSRG-115, Computer Systems Research Group, University of Toronto, July 1980.
4. H.Erkiö, J.Sajaniemi and A.Salava, An Implementation of Pascal on the Burrougs B6700. Report A-1977-1, Department of Computer Science, University of Helsinki, May 1977.
5. R.Farrow, Experiences with an Attribute Grammar-Based Compiler. Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Jan. 1982, 95-107.
6. G.Goos and G.Winterstein, Problems in Compiling Ada. Trends in Information Processing Systems, A.J.W.Duijvestijn and P.C.Lockemann (eds.), Springer-Verlag, Berlin - Heidelberg - New York, 1981, 173-199.
7. K.Hiitola, An Analysis of the Static Semantics of Pascal (in Finnish). Department of Computer Science, University of Helsinki, in preparation.
8. M.Jazayeri and D.Pozefsky, Space-Efficient Storage Management in an Attribute Grammar Evaluator. ACM Transactions on Programming Languages and Systems 3, 4 (Oct. 1981), 388-404.
9. M.Jazayeri and K.G.Walter, Alternating Semantic Evaluator. Proceedings of the ACM 1975

Annual Conference, Oct. 1975, 230-234.

10. K.Koskimies, An Experience on Language Implementation Using Attribute Grammars. Report A-1982-2, Department of Computer Science, University of Helsinki, March 1982.

11. K.Koskimies, L.Juutinen and J.Paakki, An Attribute Grammar for C-Euclid. Computer Listing, Department of Computer Science, University of Helsinki, March 1982.

12. K.Koskimies and K.-J.Räihä, On the Use of Attribute Grammars for Describing One-Pass Translation. Manuscript, Department of Computer Science, University of Helsinki, Aug. 1981.

13. B.Lampson, J.J.Horning, R.L.London, J.G. Mitchell and G.J.Popek, Report on the Programming Language Euclid. SIGPLAN Notices 12, 2 (Feb. 1977).

14. P.M.Lewis, D.J.Rosenkrantz and R.E.Stearns, Attributed Translations. Journal of Computer and Systems Sciences 9 (Dec. 1974), 279-307.

15. M.Marcotty, H.F.Ledgard and G.Bochmann, A Sampler of Formal Definitions. Computing Surveys 8, 2 (June 1976), 191-276.

16. K.-J.Räihä, A Space Management Technique for Multi-Pass Attribute Evaluators. Ph.D. Thesis, Report A-1981-4, Department of Computer Science, University of Helsinki, Sept. 1981.

17. K.-J.Räihä, M.Saarinen, E.Soisalon-Soininen and M.Tienari, The Compiler Writing System HLP (Helsinki Language Processor). Report A-1978-2, Department of Computer Science, University of Helsinki, March 1978.

18. K.Ripken, Application of Meta-Compilation Methods in the Ada Test Translator Development. GI - 10. Jahrestagung, R.Wilhelm (ed.), Springer-Verlag, Berlin - Heidelberg - New York, 1980, 66-77.

19. M.Sarjakoski, Space Management for Attributes in the Compiler Writing System HLP (in Finnish). M.Sc. Thesis, Report C-1982-10, Department of Computer Science, University of Helsinki, Feb. 1982.