# EXPERIENCES WITH THE COMPILER WRITING SYSTEM HLP

Kari-Jouko Räihä

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

ABSTRACT

Compilers for languages of widely varying nature have been constructed using the compiler writing system HLP. Characteristics of the language descriptions and of the compilers produced are reported. Opinions of the users are discussed, and various ways to improve HLP are suggested. The experiences gained are relevant for similar projects aiming at the construction of compiler writing tools.

## 1. INTRODUCTION

HLP (Helsinki Language Processor) is a compiler writing system designed and implemented at the University of Helsinki. The development work was begun in 1975, when funds for four research workers were obtained from the Academy of Finland. The main parts of the system were frozen in March 1978, when a user manual [27] was published. After that most of the development work has concentrated on increasing the user-friendliness, efficiency, and flexibility of the system. The main financial support for the project stopped in 1978, and since then the work has been carried out as normal university research and partly as student assignments.

The system runs on the Burroughs B6700 computer. It is written in B6700 Extended Algol and produces compilers written in the same language. The size of the present version of HLP is roughly 35.000 program lines, of which about 25% are comments. See [36] for more details on the project.

The source language is described to the system using two metalanguages: one for the lexical structure of the source language, and the other for everything else. Furthermore, a specially designed job control language [26] is used for controlling the execution of the various parts of HLP.

The lexical metalanguage offers simple set operations for describing character sets, regular expressions (with simple transformations) for describing token classes, and action blocks for describing the actual scanning and screening.

The main metalanguage combines features for defining the syntax and semantics of the source language. Syntax is defined using a version of BNF. Static semantics is described by semantic attributes, with complicated semantic actions expressed as procedures in B6700 Extended Algol. Code generation is defined by a translation scheme, where translation actions are again expressed as Algol procedures.

A detailed description of the metalanguages is given in [27].

The compilers produced by HLP employ the LALR(1) parsing method. The parse tree is constructed explicitly, and some semantic attributes 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 [9,24]. A final pass over the parse tree performs the code generation on the basis of the translation schemes.

In this paper we discuss the experiences gained in using HLP. In Section 2 we list various properties of the grammars used to describe different source languages. The quality of the compilers produced from these grammars is discussed in Section 3. Opinions of the users of HLP are given in Section 4. We conclude in Section 5 by summarizing the main implications of the experiences gained.

## 2. APPLICATIONS OF HLP

HLP is presently being used in student assignments on an introductory course on compiler construction. Moreover, a wide variety of real programming languages has been at least partially implemented with the aid of HLP. Owing to our limited resources, all these implementation tasks have been carried out by graduate students. For the same reason it has not always been possible to polish the code generation phase of the compilers produced. The emphasis has been on areas where HLP has more support to offer: lexical structure, syntactic analysis, and static semantics.

We have tested HLP with languages of very different nature in order to find out the suitability of the description tools in various situations. Thus HLP has been used to produce assemblers, precompilers, compilers for machine-oriented high-level languages, and compilers for general high-level languages. It has also been applied in implementing parts of its own job control language [21] and in describing a grammatical data base model [14,17].

Table 1 contains some properties of the language descriptions. The following abbreviations are used:

$|N|$ = number of nonterminals in the grammar

$|T|$ = number of terminals in the grammar

$|P|$ = number of productions in the grammar

$|G|$ = size of the grammar = $\sum_{X \to \alpha \in P} |X\alpha|$

$|A|$ = number of attributes in the grammar

ASE = number of evaluation passes in an alternating semantic evaluator

The syntactic information in Table 1 was obtained mainly from [20].

The figures in Table 1 do not always directly reflect properties of the language; instead, they may be strongly influenced by the nature of the implementation project. For instance, the attribute grammar for Euclid was produced from scratch, whereas in

| Source language | | Target language | Grammar | | | | | | | LALR(1) parsing automaton | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Ref. | | Ref. | \|N\| | \|T\| | \|P\| | \|G\| | \|A\| | ASE | Lines | States | Transitions |
| Cobol | [37] | - (only syntax) | [18] | 418 | 290 | 930 | 3113 | - | - | 2200 | 1839 | 8525 |
| PAL | [42] | - (only syntax) | [42] | 31 | 47 | 79 | 259 | - | - | 100 | 155 | 1235 |
| B6700 Extended Algol | [ 2] | - (only syntax) | [35] | 268 | 145 | 715 | 2350 | - | - | 1700 | 1389 | 10840 |
| Ada | [23] | - (only syntax) | [ 4] | 170 | 94 | 353 | 1065 | - | - | 500 | 619 | 3649 |
| A grammatical data base model | [14] | - (no code generation) | [17] | 16 | 10 | 22 | 72 | 12 | 2 | 200 | 48 | 87 |
| Simula | [ 5] | - (no code generation) | [39] [31] | 147 | 77 | 283 | 795 | 36 | 4 | 3200 | 423 | 3533 |
| Euclid | [15] | - (no code generation) | [12] | 158 | 96 | 342 | 1061 | 47 | 1 | 9400 | 628 | 3664 |
| MIXAL | [10] | MIX 1009 machine language | [30] | 27 | 17 | 52 | 132 | 17 | 2 | 1300 | 75 | 247 |
| PL360 | [40] | IBM 360 assembly code | [ 6] [ 8] | 60 | 65 | 151 | 490 | 49 | 2 | 3300 | 255 | 883 |
| PL/Mikko3 | [19] | Intermediate code | [16] | 102 | 135 | 353 | 1119 | 39 | 3 | 3500 | 595 | 4210 |
| S-Fortran (two versions) | [ 3] | a) Fortran | [28] | 223 | 106 | 449 | 1317 | 49 | 2 | 4200 | 781 | 2340 |
| | | b) Fortran | [28] | 74 | 46 | 130 | 407 | 48 | 2 | 2500 | 253 | 593 |
| Pascal | [41] | B6700 Extended Algol | [ 1] [13] | 138 | 68 | 254 | 721 | 72 | 5 | 6000 | 410 | 2290 |

Table 1.

the implementation of Pascal we took a "quick and dirty"-approach by trying to make use of an already existing Pascal compiler [7]. It turned out that this decision greatly obscured the attribute grammar, which partly explains the large number of attributes for Pascal. Another reason is that the Pascal compiler contains code generation, which is presently missing from the Euclid compiler. The addition of code generation typically causes an increase in the number of inherited attributes in the grammar.

The grammars for Simula and PL360 were originally produced elsewhere. They were fairly easily adapted to the form required by HLP. Moreover, grammars developed for HLP have been used in another system [22]. This shows one of the advantages of using a compiler writing system for producing a compiler: although the *compilers* generated by HLP are not portable, the *grammars* are. The set of languages which have been described by an attribute grammar is gradually growing. For instance, if we were to begin the implementation of a Pascal compiler today without our own grammar, we would undoubtedly adapt the grammar in [38] for HLP.

S-Fortran extends standard Fortran mainly with some new control structures. Two grammars have been written for S-Fortran: version (a) contains the complete syntax, whereas version (b) only checks the correctness of those parts of the program which contain features not in standard Fortran, leaving the rest to the Fortran compilation which follows the preprocessing phase. As a consequence, the size of the syntax in version (b) is less than a third of version (a). This is achieved by an increase in the size of the lexical description, which is about 80 lines for version (a) and 175 lines for version (b). Still, the savings in manual work are considerable. The description of the preprocessor would not be possible if the lexical metalanguage did not allow the use of several action blocks. Now the 'mode' of scanning can be changed so that standard Fortran is effectively skipped, and more detailed scanning is resumed when some catchword is encountered.

## 3. QUALITY OF THE PRODUCED COMPILERS

The emphasis in the implementation projects has been in describing the source language, not in producing an efficient compiler. The testing of the generated compilers has not been very comprehensive. However, some information on the processor time required by various compilers has been collected in Figure 1. The size of the test program is measured in lexical tokens. This is a more accurate measure of program size than the number of program lines, which can vary a lot depending on programming style and on the programming language. For the test programs used in Figure 1 the average number of lexical tokens per program line ranged from 3.4 (for MIXAL) to 8.2 (for PL360).

The times in Figure 1 are enormous and far from production quality; perhaps only the time required by the S-Fortran preprocessor could be tolerated in a production

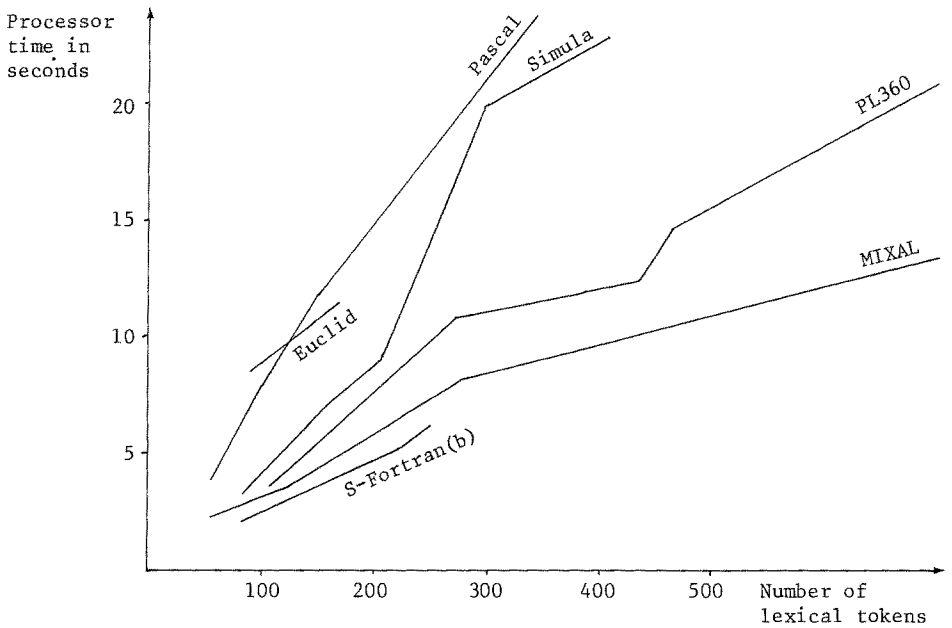


Figure 1.

environment. What is the reason for this excessive usage of time? A comparison of the various compilers shows that almost independently of the source language, the time required for lexical analysis, parsing, and construction of the parse tree (with space allocation for attributes) is about 40% of the entire time, while attribute evaluation takes 40-45% and code generation the rest 15-20%.

Lexical analysis [29] is certainly very fast. A comparison of the original lexical analyzer in the hand-written Pascal compiler [7] to the one produced by HLP showed that the latter was considerably faster, resulting in a reprogramming of the hand-written analyzer! This emphasizes one of the advantages of using a compiler writing system: it is possible to analyze the efficiency of various solutions and language features in more detail than in a normal compiler project. When the compiler is generated automatically, the best solutions found are made available to users having little or no idea of the characteristics of the specific computer.

In spite of the speed of the lexical analyzer, a further analysis of the Pascal compilers showed that the first phase (lexical analysis, parsing, construction of the parse tree, linking of attributes) of the compiler produced by HLP alone took much more time than the entire compilation with the hand-written compiler. We can see two reasons for this situation. First, the other parts of the compilers produced by HLP have not been tuned as carefully as the lexical analyzer; a wise choice between several possible statement structures reduced the time for lexical analysis by a factor of ten. And secondly, HLP generates multi-pass compilers, whereas the hand-written

Pascal compiler is a one-pass compiler. Although we use one of the simplest techniques for attribute evaluation in a multi-pass compiler, the construction of the parse tree and the linking of attributes take a lot of time which is not required in a hand-written compiler. It seems that the price for ease of language and compiler description (i.e. the allowance of multi-pass evaluation) is that the resulting compiler will be slower than a hand-written one.

Although the time requirements are large, the usage of memory is the real hindrance for testing the compilers with large programs. For instance, the largest Euclid program shown in Figure 1 consists of 165 lexical tokens (42 program lines), and yet its average core usage is more than 90K words of B6700 memory. We are presently implementing a new dynamic storage management scheme for semantic attributes [25]. This was anticipated in the attribute grammar for Euclid, where the emphasis was on elegance of language description rather than efficient evaluation of attributes. Consequently, under the present static space allocation scheme the Euclid compiler has symbol tables hanging all over the parse tree, which accounts for the large space requirement.

In other grammars, the users have been to some extent responsible for the space allocation themselves: critical attributes have been implemented as global attributes, which have fewer (yet more than one) instances than the attributes in a pure attribute grammar. Even then the space requirements are considerable; for instance, the average core usage for the data structures required in the compilation of a Pascal program with 500 lexical tokens is almost 20K words. The space requirement seems to grow linearly with the size of the program, making the compilation of very large programs impossible.

The main reason for the vast amount of storage required is the number of attribute instances, which tends to explode with the size of the program: the attributed parse tree which corresponds to a PL360 program with 134 lines has more than 15.000 attribute instances! Although we hope that the new strategy for memory management will somewhat diminish the usage of space, it seems again obvious that the space requirements of the compilers produced automatically using the present techniques are much larger than those of hand-written compilers.

Besides the data structures, the code of the generated compilers takes a lot of space, too. One reason for this may be that all parts of the compiler (lexical analyzer, parser, error recovery procedure, semantic analyzer, code generator) are generated individually for each source language, instead of being table-driven. In the case of the lexical analyzer its speed justifies this decision, especially as the lexical analyzers are reasonably short (around 1000 program lines). However, for the other parts the wiseness of the technique can be questioned. Although the encoding of the tables into program form makes the text of the compiler fairly readable, it is still probable that in further revisions of HLP we will at least offer the generation of table-driven compilers as an alternative to the present approach.

We have seen that the compilers produced by HLP are inefficient. However, although easiest to measure, efficiency is not the only factor to be considered in estimating the quality of a compiler. In other respects the compilers produced by HLP do better, because they are described using high-level metalanguages. Thus such properties as portability, reliability and modifiability are properties of the description rather than properties of the compiler. Moreover, in most cases the students in charge of the implementations had only little knowledge of HLP and of the source language before the implementation, and yet the time required for completing the assignments varied from three to twelve man months; note however that usually the result was not a complete compiler with code generation.

## 4. OPINIONS OF THE USERS

The users quite understandably like best those parts of the system which are most automatic. In particular, the error recovery part [32] requires no input from the user besides the context-free syntax. Consequently, the first automatically generated error messages encountered in test runs are often considered by students as small miracles. The compactness of the lexical metalanguage has been quite appealing, too. The only major complaint has come from users describing languages with a fixed-column format (MIXAL, Cobol, S-Fortran); the description of column dependencies with regular expressions is, although possible, rather awkward.

The syntactic and semantic metalanguages were generally considered reasonably easy to read. However, the creation of syntactic and semantic descriptions which satisfy the requirements of the system was more problematic. In the syntactic part, the main problem was the LALR(1) condition (or more exactly, the LR(1) condition: all of the conflicts in the initial attempt at an Euclid grammar were typically LR(1) conflicts, not specifically LALR(1) conflicts). In particular, the structure of expressions of various type is often severely distorted when the syntax is forced into unambiguous LALR(1) form. Another source of conflicts is formed by optional features in the source language. In Cobol, even some declarations and statements are optional, not to mention a whole bunch of optional keywords. In Euclid, semicolons are often optional. Such options are generally described by $\varepsilon$-productions, possibly causing LALR(1) conflicts which are difficult to solve. Figure 2 (taken from [11]) shows how the number of LALR(1) conflicts for the Euclid grammar developed in consecutive runs.

The Euclid syntax was written by a person who was thoroughly familiar with LALR parsing, so that the initial grammar (with almost 30 conflicts) was not just 'any' grammar: it was written with LALR parsing in mind. For a typical user (with less knowledge of LALR parsing) and for a grammar of the same size, both the number of conflicts in the initial grammar and the number of runs required to remove the conflicts were roughly five times larger. One reason for this is that the conflicts are reported in terms of the automaton, not in terms of the syntax; this may sometimes

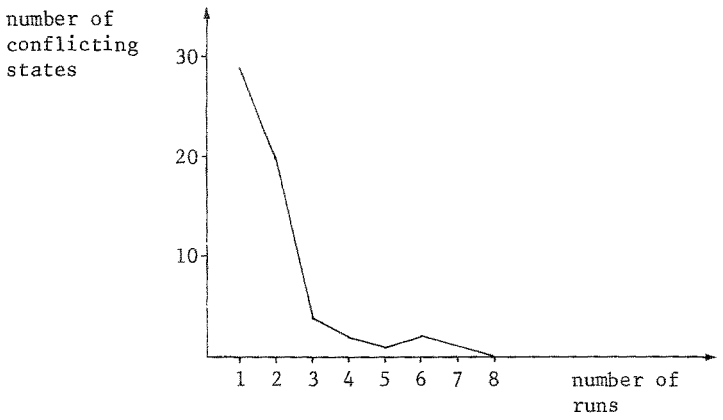


Figure 2.

lead to hasty and unsuccessful correction attempts. A modification of HLP to remedy this situation is in preparation [33]. Many conflicts could be resolved by the use of semantic information; such extensions to HLP are presently being planned [34]. This feature seems to become more and more important with new languages which apply the principle of "uniform reference", i.e. where semantically different constructs have the same syntactic outlook.

Semantic attributes have been generally well received. The parse tree gives a nice underlying structure for semantic processing. The compilation of control structures is particularly easy; this made the writing of the S-Fortran preprocessor very convenient. Moreover, the description can be built both in small syntactic units (production after production) and in small semantic units (groups of attributes), making the description easily manageable. Quoting from [31], "how else could you manage to develop a 3000-line program without any algorithms, flowcharts, or other documents?"

Perhaps the most annoying feature with the semantic metalanguage has been the necessity to be aware of the space constraints, i.e. the necessity to use global attributes. Global attributes are dangerous because they make the meaning of the semantic description depend on the order of attribute evaluation. We tried to impose a discipline on the use of global attributes by dividing them into inherited and synthesized on the basis of some experiments with symbol tables. However, it turned out that this classification did not suit equally well for other situations where global attributes were necessary. Consequently, to meet the restrictions imposed by HLP and to achieve efficiency, users were tempted to resort to the semantic procedures for using global attributes in a way that would otherwise have been prevented by HLP (the system does not check how semantic procedures use their parameters). This is clearly undesirable. Even though some users have in the long run grown to like global attributes (for the same reasons why global variables are sometimes more convenient than local variables or parameters), we are still planning to abandon global attributes altogether if the

new memory management strategy meets our expectations.

Users have also disliked the fact that a great deal of the semantic processing is performed in semantic procedures instead of semantic rules. Such scattering of semantic information is a consequence of the design principle of the semantic metalanguage: we tried to keep the metalanguage small and resort to Extended Algol for any complicated processing. In fact, the semantic rules are actually compiled into Extended Algol in a preprocessing fashion. The decision was motivated by ease of implementation, but it had several undesirable consequences: scattering of semantic information into places wide apart from each other; unsatisfactory data types (those of Algol) for semantic attributes; and unsafe procedures, highlighted by the tricks used to access global attributes illegally. It would have been better to design a completely new and more powerful metalanguage for the semantic rules. Another way to diminish the user's programming work would be to offer some built-in semantic actions for frequently occurring tasks, e.g. symbol table management.

None of the users complained of the restrictions imposed by the alternating semantic evaluator as being very difficult to satisfy. On the contrary, in the grammar for PL360 the checking of the grammar at metacompile time revealed circularities in the original grammar [6]. This supports our belief that contrary to what has sometimes been claimed, the problem of circular definitions is not merely academic. The well-formedness of every attribute grammar should be checked at metacompile time, be it with a full circularity test or some more restrictive algorithm.

The satisfaction with the tool for code generation, i.e. tree-walking translation schemes, seems to depend on how closely the target language is related to the source language. In the case of S-Fortran the translations were quite adequate, but for Pascal (with Extended Algol as target language) a more powerful tool would have been useful. The compiler for Simula does not contain code generation, but the author of the grammar expected that writing the code generation part with the translation schemes would be difficult.

Although the conceptual tools have their flaws, the main reasons for frustration among the users have been of a pragmatic nature and often typical for large software projects in general. Many of the student assignments were begun when the system was still under development, so that many users were faced with a malfunctioning of HLP. Of course, debugging the system was one of the main goals in the test implementations, but it is understandable that students were not pleased with delays caused by system corrections, not to mention the time they had in vain used in trying to find an error in their own work.

The job control language for HLP was not released before the system had been used for quite a while. Before it, the users were forced to handle the management of a large number of files used and produced by HLP themselves, an annoying and error-prone task. The satisfaction with the job control language has convinced us that every similar

system with several interacting modules should have a high-level control language of its own, even when the general job control language of the computer is as pleasent as that of B6700. Such a control language could also be used for tying together tools designed originally for independent use. One of the short-comings of HLP is that it is too monolithic: it is somewhat difficult to apply only some part of the system, since it has been anticipated that all parts of the compiler are produced by HLP; then the couplings between different modules in the compiler are unnecessarily strong. Moreover, the compilers produced by HLP have a too fixed form: they do not offer options for users as hand-written compilers do.

Some of the early users also complained about the lack of adequate documentation. This need has been somewhat removed by the publication of the user manual [27], but even now the wish of a presentation more in a textbook style with examples of varying difficulty comes up every now and then. In particular, more information has been required by persons whose background has not been in compiler writing, but who have used HLP to implement small application-oriented languages; this is exactly the area where HLP and other compiler writing systems should be most useful. This indicates that for application-oriented users the metalanguages should be kept simple.

HLP produces a lot of useful listings and statistical information of the language descriptions. Yet many users have expressed the wish for one further listing: a cross-reference list of the semantic rules arranged by semantic attributes, not by productions. Although the grammar is easiest to develop when it is arranged by productions, the cross-reference list would be useful for documentation purposes.

## 5. CONCLUSIONS

HLP has been extensively used in student assignments. Various languages of different nature have been more or less completely implemented with the help of HLP. The compilers produced by the present system are inefficient. Several revisions are planned, but it seems obvious that the compilers generated by HLP will never reach the efficiency of hand-written compilers. However, the compiler descriptions have been fairly easy to write, easy to read, and easy to maintain.

The users have been reasonably satisfied with the description tools, the main difficulties being concerned with the LALR(1) condition and the excessive usage of space in the implementation of attribute grammars, which has forced the users to deviate from using pure attribute grammars. Yet the main complaints have been pragmatic: difficulties with the use of the system at the program level, lack of documentation, lack of personal assistance for uninitiated users, behaviour of the produced compilers. Although users can adapt themselves rather easily to short-comings in the metalanguages, they soon get frustrated if the system has bugs, if it is difficult to use, or if the product has to be manually modified. The amount of work required for such practical issues should not be underestimated in similar projects.

ACKNOWLEDGEMENTS

The HLP project is headed by Professor Martti Tienari. Additional members in the design and implementation team included Kai Koskimies, Paula Mäkelä, Mikko Saarinen, Seppo Sippu and Eljas Soisalon-Soininen. This particular paper could never have been written without the efforts of numerous students, who participated in the implementation and use of the system. The time spent by several persons for discussions on the topic and for reading a draft of the manuscript is also gratefully acknowledged.

This work was supported by the Academy of Finland.

REFERENCES

Unless otherwise stated, the reports listed below have appeared in the publication series of the Department of Computer Science, University of Helsinki.

1. Asp,J.: A description of the syntax and static semantics of Pascal using HLP (in Finnish). In preparation.
2. Burroughs Corp.: Burroughs B6700/B7700 Algol language reference manual. Burroughs Corporation, May 1974.
3. Caine, Farber, Gordon, Inc.: S-Fortran language reference guide. Caine, Farber, Gordon, Inc., 1974.
4. CII Honeywell Bull: LALR(1) grammar for Ada. Unpublished computer listing, CII Honeywell Bull, 1979.
5. Dahl,O.-J., Myhrhaug,B. & Nygaard,K.: Simula 67 Common Base Language. Norwegian Computing Center, Oslo, 1968.
6. Dreisbach,T.A.: A declarative semantic definition of PL360. Technical Report UCLA-ENG-7289, Computer Science Department, University of California, Los Angeles, California, October 1972.
7. Erkiö,H., Sajaniemi,J. & Salava,A.: An implementation of Pascal on the Burroughs B6700. Report A-1977-1, May 1977.
8. Ikonen,M.: An attribute grammar for PL360 (in Finnish). Internal Report C-1979-24, January 1979.
9. Jazayeri,M. & Walter,K.G.: Alternating semantic evaluator. Proceedings of the ACM 1975 Annual Conference, October 1975, 230-234.
10. Knuth,D.E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1967.
11. Koskimies,K.: LALR(1) syntax analysis for the programming language Euclid (in Finnish). Internal Report C-1978-41, April 1978.
12. Koskimies,K. & Juutinen,L.: An attribute grammar for the compile-time semantics of a subset of the programming language Euclid. Internal Report C-1979-130, December 1979.
13. Laaksonen,H.: Code generation in the Pascal compiler produced by HLP (in Finnish). Internal Report C-1978-65, July 1978.
14. Laine,H., Maanavilja,O. & Peltola,E.: Grammatical data base model. Information Systems 4, 4 (1979), 257-267.
15. Lampson,B.W., Horning,J.J., London,R.L., Mitchell,J.G. & Popek,G.J.: Report on the programming language Euclid. SIGPLAN Notices 12, 2 (February 1977).

16. Luikku,M. & Riekkola,E.: Construction of a PL/Mikko compiler (in Finnish). Internal Report C-1979-73, April 1979.

17. Maanavilja,O.: The query language in a grammatical data base model (in Finnish). Internal Report C-1979-45, March 1979.

18. Niittyranta,T.: An LALR(1) syntax for CODASYL Cobol (in Finnish). Internal Report C-1978-62, May 1978.

19. Nokia Electronics: MIKKO3 RTX II PL programming language user's guide. Nokia Electronics, Helsinki, 1978.

20. Nurmi,O.: Testing the LR(k) and SLR(k) conditions in the compiler writing system HLP (in Finnish). Internal Report C-1979-82, June 1979.

21. Pietarinen,P.: Implementation of the job control language for HLP (in Finnish). Internal Report C-1978-49, April 1978.

22. Pozefsky,D.: Building efficient pass-oriented attribute grammar evaluators. Report TR 79-006, Department of Computer Science, University of North Carolina, Chapel Hill, N.C., 1979.

23. Preliminary Ada reference manual. SIGPLAN Notices 14, 6 (June 1979), Part A.

24. Räihä,K.-J.: On attribute grammars and their use in a compiler writing system. Report A-1977-4, August 1977.

25. Räihä,K.-J.: Dynamic allocation of space for attribute instances in multi-pass evaluators of attribute grammars. Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 14, 8 (August 1979), 26-38.

26. Räihä,K.-J. & Saarinen,M.: Operating instructions for HLP. Internal Report C-1979-68, May 1979.

27. Räihä,K.-J., Saarinen,M., Soisalon-Soininen,E. & Tienari,M.: The compiler writing system HLP (Helsinki Language Processor). Report A-1978-2, March 1978.

28. Rikkilä,J.: A precompiler for B6700 S-Fortran (in Finnish). Internal Report C-1978-61, 1978.

29. Saarinen,M.: The lexical part of the compiler writing system HLP (in Finnish). Internal Report C-1978-55, October 1978.

30. Siitari,H.: Application of HLP in the construction of a MIXAL compiler (in Finnish). Internal Report C-1978-113, October 1978.

31. Siitari,H.: An attribute grammar for the programming language Simula (in Finnish). Internal Report C-1979-90, May 1979.

32. Sippu,S. & Soisalon-Soininen,E.: On defining error recovery in context-free parsing. Automata, Languages and Programming, Fourth Colloquium, A.Salomaa & M.Steinby (eds.), Springer-Verlag, Berlin - Heidelberg - New York, July 1977, 492-503.

33. Suikkanen,J.: Conflicts in the LALR(1) parsing automaton (in Finnish). Internal Report C-1980-5, November 1979.

34. Tarhio,J.: Parsing ambiguous grammars in the compiler writing system HLP (in Finnish). Internal Report C-1979-93, May 1979.

35. Tarhio,J.: An LALR(1) syntax for Extended Algol (in Finnish). Internal Report (to appear).

36. Tienari,M.: Research on programming languages and compilers. Data 11/ -78 (1978), 75-78.

37. USASI: USA Standard Cobol. USAS X3.23-1968, USA Standards Institute, 1968.

38. Watt,D.: An extended attribute grammar for Pascal. SIGPLAN Notices 14, 2 (February 1979), 60-74.

39. Wilner,W.T.: Declarative semantic definition as illustrated by a definition of Simula 67. Ph.D. Thesis, Computer Science Department, Stanford University, Stanford, California, June 1971.

40. Wirth,N.: PL360, a programming language for the 360 computers. JACM 15, 1 (January 1968), 37-74.

41. Wirth,N.: The programming language Pascal (revised report). Bericht 5, Institut für Informatik, Eidgenössische Technische Hochschule, Zürich, 1972.

42. Wozencraft,J.M. & Evans,A. Jr.: Notes on programming linguistics. Department of Electrical Engineering, MIT, Cambridge, Mass., July 1979.