TOWARDS AN EXPERT SYSTEM FOR COMPILER DEVELOPMENT

by

Harvey Abramson

Technical Report 87-33

October 1987

Harvey Abramson
Department of Computer Science
The University of British Columbia
2075 Wesbrook Mall
Vancouver, B.C. Canada
V6T 1W5

Towards an Expert System for Compiler Development

Since the introduction of high-level programming languages in the late fifties and early sixties, a great deal of expertise has been developed in the area of compilation and interpretation of these languages. (A recent edition of an *introductory* textbook [Aho&Sethi&Ullman,1986] totals 796 pages! For other treatments of compiling see [Aho&Ullman,1973], [Aho&Ullman,1978], [Bornat,1979], and [Davie&Morrison,1981]) As this area of expertise developed, attempts were made to codify and systematize this knowledge in "Translator Writing Systems" and "Compiler Compilers". The trouble with such codifications, however, was, and continues to be, that very little of the knowledge that is current is embodied in those systems. If one considers these as expert systems, then they must be judged to be expert in only a subset of compiling technology, and also very rigid and limited experts at that. One might consider as a case in point the well known Unix utilities LEX (lexical scanner generator)) and YACC - "Yet Another Compiler Compiler". (The name YACC itself perhaps betrays some despair.) One can use YACC as a tool to create a parser using one specific technique (LALR(k) albeit augmented to deal with some ambiguities in expressions), and one can subsequently generate code by associating portions of C code with the grammar rules. There is very little support for helping out if the grammar is not in the appropriate class, and there is very little of an environment for developing the code generation, understanding how to modify the grammar's departure from the required class, etc. Other systems make use of further aspects of compiler technology, for example, use of attribute grammars to specify code generation, or some way of including code optimizations. However, all of these systems tend to be rather massive and hard to extend and modify. One serious problem with any of these systems, furthermore, is that they are not written in languages which are sufficiently "high" and powerful enough to manipulate programs and representations of programs: some of the rigidity mentioned above stems from this lack of a meta-level facility.

The area of knowledge we are concerned with is a rather formal one. Grammars can be used to specify all or portions of lexical and syntactic analysis. Attribute rules can be used to specify how functions or relations on derivation tree nodes are to be evaluated. The attributes can specify a formal representation of the source program in some standard intermediate code. Optimization algorithms working over such an intermediate code representation can be used to produce another version of the original program, in a more compact or efficient intermediate code representation. Somewhat more experimentally, hardware can be formally described, and the relation between the semantics of a programming language and the code of a target machine can be specified.

There are apparently several formalisms (lexical specification, syntactic specification, attribute rule definition, code generator specification, etc.) that would have to be tied together in order to unify compiling knowledge in an expert system. If this were so, then the design of the interfaces between the various formalisms might itself prove a fairly difficult problem. However, the formalisms involved can be uniformly represented in first-order logic, and the techniques developed for expert systems written in logic programming languages such as Prolog can be utilized to join together the various aspects of compiler specification. Logic programming, and its implementation in Prolog (see [Pereira,1982] and [Clocksin&Mellish,1981] for details of the Prolog programming language), provide a high level specification of *what* should be computed without the clutter of *how* it should be computed; the procedural interpretation of logic

programming, moreover, yields a means of executing these specifications. Logic programming also provides a grammatical notation which can be executed: if, in general, "the logical specification is the program", then in the case of the logic grammar notation, "the grammar is the interpreter". Logic programming manipulates trees directly, and with a little bit of extra work, graphs: these are the data structures which can be used to specify, quite generally, the optimization algorithms developed over the last two decades. A suite of logic programs can then, in principle, be developed to specify - and therefore, implement - efficient programming language processors.

Logic programming has been used to define languages and to implement compilers. See the work of [Colmerauer,1978] which introduced the first logic grammar notation, the basis for formal language applications of logic programming. Moss pioneered the use of logic programming to formally describe - and therefore to implement - programming languages: see [Moss,1979], [Moss,1981], and [Moss,1982]. Warren has pioneered the application of logic programming to compiler writing [Warren,1977]. [Maluszynski&Nilsson,1981,82] extend the notion of unification to grammatical processing.

What sort of expert system shell is required for compiler applications? We take the point of view that since much of compiling knowledge can be specified formally and with relatively little "fuzziness", a simply structured shell with explanation and "query-the-user" facilities will prove adequate - at least in the initial stages - for our purposes. We will model our shell on recent work at Imperial College in the field of expert system design which has concentrated on areas in which domain specific knowledge is clearly and almost completely spelled out. An example is the design of expert systems for handling social security benefit claims and queries regarding immigration laws. In such contexts, the construction of an expert system is relatively easy in that there is little in the way of uncertain knowledge which has to be dealt with. An expert system for compiler development provides an environment for developing compilers. It can be considered a sophisticated compiler writing system which encapsulates textbook knowledge about compiler construction in an expert system shell. Such a shell, based on APES (A Prolog Expert System Shell, developed at Imperial College by Hammond and Sergot: aee [Hammond,1982a,b] and [Sergot,1983]) provides input of specifications, evaluation of queries, and explanation of answers in a domain independent fashion. The shell is then tailored to a domain of knowledge relevant to the specification of programming language processsors. Hence, the system must know about grammatical specifications of lexical and syntactic structure. Explanations and dialogue must be in grammatical terms, not in terms of the generated Prolog clauses (see [Salim,1985]).

In the rest of this paper we shall examine what a simple logical expert system shell would have to provide in order to advance towards an expert system for compiler development. (We will assume some familiarity with logic programming, Prolog, and with Definite Clause Grammars. See the above mentioned references in connection with Prolog, and [Pereira&Warren,1981] about Definite Clause Grammars.) We will describe how grammatical knowledge can be represented in logic and how various grammatical formalisms can be incorporated into a system as a set of ways of building parsers. The grammars will also include a way of specifying a logical version of attribute grammars: as a result of a parse, a derivation tree is created. This derivation tree is a first-order term which can then be traversed to produce either code directly, or yet another intermediate representation of the program; in either case, the code or intermediate form is again a term or terms in logic. Optimization algorithms too can be specified in Prolog, and eventually, machines.

The main idea is that the uniform representation in the very high level programming language of Prolog, together with Prolog's meta-level features, permits the creation of an environment for compiler development which makes it possible to incorporate far more of current technology than has been heretofore possible. It is important to emphasize the level of the programming language not only for its program manipulation (meta-level) possibilities, but also for the fact that complicated algorithms can be represented quite clearly in a small number of lines of code: a programmer can thus design a system which does more. We are also assuming the availability of *efficient* implementations of Prolog, i.e., Prolog compilers whose speed and efficiency are comparable to the best implementations of LISP. The expert system for compiler development will provide at least rapid prototyping of language implementations; where Prolog compilers are

available the expert system can provide efficient implementations of language processing systems.

Lexical expertise.

If one is designing a language processor, it is convenient to provide a stage of lexical analysis in which characters are grouped into meaningful tokens for syntactic analysis. A technique used in the formalization of lexical analysis has been to use regular expressions or grammars to specify the allowed tokens. The regular expressions can be used to generate a finite state non-deterministic automaton, which in turn can be minimised to yield an efficient scanner recognizer. This strategy, however, involves a formalization which is interesting, but which may be unnecessarily powerful for practical situations.

In practical situations, lexical analysis can be specified by the following logic grammar rules:

```
lexemes(X)    ::=        space , lexemes(X).
lexemes([X|Y]) ::=       lexeme(X) , lexemes(Y).
lexemes([])   ::=        [].
```

The first rule may be read: *lexemes* consist of *space*, followed by more *lexemes*; *X* represents the *lexemes*. The second rule may be read: *lexemes* consists of a *lexeme* followed by more *lexemes*; the *lexemes* are represented by the list *[X/Y]* built up from the *lexeme X* and the *lexemes Y*. The third rule may be read: *lexemes* may be empty, represented by the empty list *[]*. (See the first appendix for a complete logical specification of the lexical analyzer.) These logic grammar rules are compiled to Prolog clauses by a simple processor (see papers on DCGs, DCTGs):

```
lexemes(X,Start,End)    ::=space(Start,Mid), lexemes(X,Mid,End).
lexemes([X|Y],Start,End) ::= lexeme(X,Start,Mid), lexemes(Y,Mid,End).
lexemes([],Start,End)    ::= connect([],Start,End)
```

The extra arguments, *Start*, *Mid*, *End* represent the input string which is being analysed. The predicate *connect* specifies that its first argument is the difference between the second and third arguments, i.e., that the first argument is the head of the input string:

```
connect(first,[first|Rest],Rest).
```

If one then specifies rules for what a *space* may be, and what a *lexeme* may be, then these logic grammar rules are also translated into Prolog clauses, and the entire set of such Prolog clauses generated from grammar rules constitutes a lexical scanner. Here are the rules saying what a *lexeme* is:

```
lexeme(Token)                ::=word(W) , { is_token(W,Token) }.
lexeme(Con)                  ::= constant(Con) .
lexeme(Punct)            ::= punctuation(Punct) .
lexeme(op(Binding,Op))   ::= op(Binding,Op) .
lexeme(relop(Rel))       ::= relop(Rel).
```

Without going into detailed explanation, these rules define a *lexeme* as either a *word* satisfying a constraint that the *word W* is represented by *Token* (*Token* may be either an identifier or the representation of a reserved word), or a *constant*, or *punctuation* or an *operator*, or a *relop* (relational operator). The following program of the sample language

```
read X; write X + 127
```

is analysed into the following list of lexemes:

```
lexemes: [read,id(X),;,write,id(X),op(+),num(127)]
```

Now if one were writing a compiler by hand each of these would in turn have to be specified by further rules. However, an expert system would simply prompt the user to list what the various kinds of lexemes are. For example, the following dialogue might ensue:

System:
What are punctuation symbols and token names?

User:
(lparen
) rparen
etc.

From this dialogue the system would construct grammar rules which would then be compiled into Prolog as above. The system would permit definitions in each of the categories of lexeme, but might also have standard definitions of identifiers and numbers. Thus, the lexical analyzer of Appendix 1 should be constructed by the system from a dialogue with the user who only has to know for the language being defined what the symbols are and what tokens he wants to represent the symbols. The explanation facility of the shell would be able to explain via the generated grammar rules why the input string

" abc := 123 "

is analysed as:

[id(abc), :=, 123]

If the input string had been

" abc := (123, 456) "

and, if no token had been defined for "(", the shell's *query-the-user* facility would allow the user to be consulted as to adding a token definition for "(" and ultimately yielding something like:

[id(abc),:=,lbrace,123,comma,456,rbrace]

rather than failing on lexical analysis.

Note that what we are generating is a set of logic grammar rules which are compiled into Prolog program clauses. The expert system is meant to be one entirely implemented in logic and running efficiently via a Prolog compiler. (The system can also be tailored to generate lexical scanners of a more traditional sort.) The lexical scanner generator outlined above is similar in spirit to [Horspool&Levey,1986]'s *Mkscan*, but is embedded in a more powerful and more flexible environment.

Parsing expertise.

The syntax of a programming language may be specified by context free rules of the following form:

program ::= statements
statements ::= statement, st1
st1 ::= tSEMICOLON, statements
statement ::= tIDENT, tASSIGN, expression
etc.

The symbols beginning with a lower case t are lexical token names.

What parsing method should be used when an expert system constructs a compiler? The simplest solution in a logic based system is to use the Prolog clauses which correspond to the grammar rules. The resulting parser is a top-down, left-to-right recursive descent parser (see [Davie&Morrison,1981] about this parsing method, though not in logic programming terms) which however does not permit left-recursive rules such as:

f ::= f,tPLUS,primary

The expert system would therefore have to check for the presence of such rules and provide transformations of the grammar rules to get rid of left recursion. Some context free grammars can provoke recursive descent parsers to take exponential time in parsing input; in practical programming language definition, however, recursive descent parsing time is almost always a linear function of the length of the input string. Constraints as in Definite Clause Grammars [Pereira&Warren,1980] or Definite Clause Translation Grammars ([Abramson,1984a] can be expressed by calls to Prolog predicates written within braces "{" and "}" to express non context free aspects of programming languages.

In addition, the expert system should have available other ways of parsing. It is possible, if the grammar does not contain empty rules, to use a bottom-up method of parsing know as left-corner parsing. In such a case, it is possible to generate Prolog clauses which implement this method of parsing [Matsumoto et al,1983]. What if, however, the grammar contains empty rules? Empty productions can be eliminated by known techniques and the expert system would have to incorporate these. This parsing method is, in the worst case of context free parsing, exponential, but is quite satisfactory in practical situations.

The expert system must be able if the user asks to try and use a more efficient parsing algorithm. The system should be able to test if the grammar falls into the class of LL(k) or LALR(k) grammars, and generate the appropriate tables for parsing - all represented as clauses of Horn clause logic. If the grammar does fall into such a class, then parsing time can be guaranteed to be a linear function of the length of the input string. If the grammar does not fall into one of these classes, there are some heuristics which can be tried to modify the grammar. These heuristics cannot be guaranteed to work, however. Also, some of the syntactic grammar rules suitable for LL(k) or LALR(k) parsing are sometimes regarded as not providing "natural" semantics. As a last resort, the system should use the Earley parsing algorithm, a method suitable for any context free grammar, parsing input in cubic time. See [Abramson,1986b] for an embedding of LL(k) parsing in Prolog, and [Nilsson,1986] for a logical treatment of SLR(1) grammars.

The expert system should also allow the addition of other parsing methods known to a user.

Explanation of parses, or failure to parse, must be provided by the system in terms comprehensible to the user, i.e., in terms of the original grammar, rather than in terms of failure of generated Prolog clauses. The shell we are using has been extended to give grammatical explanations where appropriate. The query-the-user facility, furthermore, permits the building up of a grammar incrementally provided that the grammatical categories are defined as being "askable".

Thus, the expert system generalizes what can be done by existing tools such as LEX and YACC but in a more powerful setting and with greater flexibility. Given a suitable way of interfacing logic and common programming languages, these tools could be incorporated into the expert system; however, the problem of interfacing a satisfactory explanation facility to existing tools is formidable. What seems more sensible is to incorporate logical versions of LEX and YACC for rapid prototyping and testing, and then using LEX and YACC themselves once the definitions for lexical and grammatical analysis have been debugged.

Parse tree representation and semantic attributes.

A parse tree is a representation of the proof that an input string is a valid "sentence" of the language specified by a context free grammar. It is also used to generate code - either an intermediate code, or machine code - by means of attribute evaluation. We represent a derivation tree in a fashion which allows subsequent evaluation of attributes specified as Horn clause rules.

(This derives from our work on Definite Clause Translation Grammars and is reported at greater length in [Abramson,1984a,b].)

The grammar rules which we use are of the form:

Syntax <:> Semantics

The syntactic portion of the rule is basically context free, but there are some additional notation conventions which permit interaction with the semantic portion of the rule which is written as a list of Horn clause like rules. For example, rather than the simple context free rule

program ::= statements

the user would write:

```
program ::= statements^^S
<:>
gen_code(Dic,Code) ::- S^^ gen_ code(Dic,Code).
```

The notation ^^*S* means that the name of the subtree for *statements* is *S*. In the semantic portion of the rule, use of the name *S* refers to this subtree. The semantic rule is the clause:

gen_ code(Dic,Code) ::- S^^ gen_ code(Dic,Code)

which is read: using a dictionary of symbols *Dic*, *Code* is generated at the root of the derivation tree for program using *gen_code* if using that same *Dic*, *Code* is generated for the subtree *S* for *statements*. (This notation is explained in detail in [Abramson,1984], and is also to be described in a chapter of [Dahl&Abramson,198?]. Appendix II contains a listing of the DCTG rules for a sample programming language.)

Here is another example of a DCTG rule which shows how code is generated for a simple assignment statement.

```
statement ::= tIDENT^^Id, tASSIGN, expression^^E
<:>
gen_code(Dic,[Exprcode,instr(store,Addr)]) ::-
  Id^^prefix(Identifier),
  lookup(Identifier,Dic,Addr),
  E^^gen_code(Dic,Exprcode).
```

The underlying context free syntax of this rule is:

statement ::= tIDENT, tASSIGN, expression

which defines a statement as an identifier, followed by an assignment symbol (whatever may have been chosen), followed by an expression. In the DCTG rule, the name *Id* is associated with the subtree for the identifier, and the name *E* with the subtree for the expression. The rule for computing the code may be read: using the dictionary (symbol table) *Dic*, the code generated for the statement is the code generated for the expression *Exprcode*, followed by a store instruction into the address *Addr* of the identifier, **if** the prefix code for the *Id* is evaluated in *Identifier*, and if the address of this *Identifier* is looked up in the dictionary *Dic* and found to be *Addr* and if the code generated for the *E* subtree using *Dic* is *Exprcode*. Note the logical variables *Exprcode* and *Addr*. These may be instantiated during a later stage of processing.

The parse tree representation which is used in the evaluation of semantic rules has the following form:

node(program,[S], (gen_ code(Dic,Code) ::- S^^ gen_ code(Dic,Code))).

The first argument is the name of the root of the derivation subtree, the second is the list of subtrees, and the third is a representation of the semantic rules associated with that root.

The parsing method used depends only on the syntactic portion of the rule considered as a context free rule; if a name for a subtree is supplied, it is used when the parse tree is constructed.

One problem which arises when a user's original grammar is modified by transformations is that the parse tree obtained will reflect the transformations imposed on the original grammar. This may be disturbing to the user, and might modify the semantic rules intended by the user. The expert system ought to be able to communicate to the user in terms of the original grammar. An aid to the solution of this problem lies possibly in the Modular Logic Grammars of [McCord,1985] which addressed a similar situation in natural language processing.

The parse tree for the sample program in the previous section can be neatly printed to yield:

```
program
  statements
    statement
      tREAD
        [read]
      tIDENT
        [id(X)]
    st1
      tSEMICOLON
        [;]
      statements
        statement
          tWRITE
            [write]
          expression
            sum
              product
                primary
                  tIDENT
                    [id(X)]
                rest_of_product
                  []
              rest_of_sum
                op_add
                  [op(+)]
                product
                  primary
                    tCONSTANT
                      [num(127)]
                  rest_of_product
                    []
                rest_of_sum
                  []
        st1
          []
```

The prefix code (where expressions are represented in Polish prefix notation) generated by the attribute rules for the sample program is:

```
[read(id(X)),write(expr(add,id(X),num(127)))]
```

Optimization expertise.

The code generated from the grammar provided in Appendix II is code for a simple abstract machine, and no cleverness has been incorporated into the attribute rules to make sure that the best code has been generated for a program. This is really a sort of intermediate code which requires at least one more step: code generation from this abstract code to some particular target machine. The prefix code generated from the sample program might thus be further translated to code for a textbook target machine:

```
0: instr(read,5)
1: instr(load,5)
2: instr(addc,127)
3: instr(write,0)
4: instr(halt,0)
5: block(1)
```

This assembly code includes allocation of storage for both program and data (location 5 is a one word block of storage). Ideally, however, more than one step is needed between the code produced by syntax directed translation and real machine code: there should be intermediate code analysis and optimization (both local and global) to make sure that the best (or at least good) real machine code is produced. In this example the value of what is read is stored into location 5 and immediately loaded into an accumulator. Location 5 is never referred to again in the program, suggesting an improvement to this code if it is possible to read directly into an accumulator.

Here the expert system will have to analyse the code generated by attribute evaluation, dividing the list of instructions into basic blocks and flow graphs (terminology taken from [Aho&Sethi&Ullman,1986]), deriving next-use information, making register allocations and assignments, and applying peephole optimizations. The basic blocks can here be represented by directed acyclic graphs (dags), and global optimization algorithms working on these dags can perform loop optimizations, constant folding, code improvements, etc. The data flow equations involved in this stage can all be phrased and implemented in Prolog, and the data structures involved will be Prolog functors.

Beyond the textbook.

Such a system encapsulates textbook knowledge about compiler writing. But it may be possible to go further towards eliminating user intervention. The major point of intervention by the user is in providing the mapping from the syntactic structure of a program to a machine. There are possibilities, however, of utilizing abstract specifcations of languages and machines to specify compilers. Experiments have been conducted by [Ganzinger&Giegerich,1985] along these lines with a purely functional language, Henderson's LispKit LISP [Henderson,1980], and could be extended initially either to another pure functional programming language (SASL [Turner,1979], or HASL [Abramson,1986a]) or to a logic programming language, and subsequently to standard programming languages.

The task here is not simple of course. Specifying how a machine is to be used is more than a matter of specifying the machine's instructions. There may be restrictions on the way memory is divided and used; calling conventions for subroutines may be established; certain data structures such as save areas may also be given a conventional form; etc. In any case, the way a machine is used must be specified by some rules even if these are usually specified in English (unfortunately, sometimes ambiguously!). The expert system would require that such rules be specified in logic. Indeed, this requirement may be a help in making sure that the rules of machine use are clearly specified.

Similarly, the task of formally specifying the semantics of programming languages is a complex one, especially for those languages which are neither logical nor functional. However, the trend in languages is towards those which have clear and simple semantics. If this trend continues, then some of the problems of specifying difficult programming constructs can be ignored.

It should also be noted that this expert system has knowledge not just of programming language translation. Consider the grammatical expertise involved, the representation of grammatical acceptability, and the evaluation of attribute rules. Various logic grammar formalisms have been widely used as tools for natural language analysis (See [Dahl&Abramson,1984], [McCord,1985], [Pereira,1981] and references cited in these papers). An analogue of evaluation of attribute rules in natural language analysis is the derivation of the "logical form" of a sentence, e.g., a first-order term representing the meaning of the sentence. Indeed DCTGs have been used to specify such a logical form for sentences of a very simple subset of English. Omitting the knowledge of machine code generation and optimization from the compiler development system yields an expert system which can be a tool in computational linguistics.

Status of the project.

The project is in its second year now. The first year's work involved building an expert system shell modeled on APES, modified to deal with DCTG knowledge and explanations, and also the design of a Prolog compiler. A Prolog compiler is desirable since the entire system is to be a logical one and the greatest efficiency in execution speed is needed to eventually make this more than a toy. This year several parsing algorithms are to be implemented and incorporated in the system, as well as some optimizations. On the theoretical end we shall begin studying the possibilities suggested in the section **Beyond the textbook**. We hope to report in later papers on the results of this year's work.

Acknowledgments

I would like to thank IBM Canada for its generous support of this project through an SUR Grant. Thanks go to Peter Lüdemann for comments on drafts of this paper.

References

[Abramson,1984a]
Abramson, H., *Definite Clause Translation Grammars*, Proceedings IEEE Logic Programming Symposium, 6-9 February 1984, Atlantic City, New Joisey.

[Abramson,1984b]
Abramson, H., *Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars*, Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Nov. 6-9, 1984.

[Abramson,1986a]
Abramson, H., *A Prological Definition of HASL, a Purely Functional Language with Unification-Based Conditional Binding Expressions*, in Logic Programming: Functions, Relations, and Equations, DeGroot, D. & Lindstrom, G. (editors), Prentice-Hall, 1986.

[Abramson,1986b]
Abramson, H., *Sequential and Concurrent Logic Grammars*, Third International Conference on Logic Programming, Springer Lecture Notes in Computer Science #226, 1986, pp. 389-395.

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

[Aho&Ullman,1973]
Aho, A.V. & Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, 1978.

[Aho&Ullman,1973]
Aho, A.V. & Ullman, J.D., *The Theory of Parsing, Translation, and Compiling*, 2 volumes, Prentice-Hall, 1973.

[Bornat,1979]
Bornat,R., *Understanding and Writing Compilers*, Macmillan, 1978.

[Clocksin&Mellish,1981]
Clocksin, W.F. & Mellish,C.S., *Programming in Prolog*, Springer, 1981.

[Colmerauer,1978]
Colmerauer, A., *Metamorphosis Grammars*, in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

[Dahl&Abramson,1984]
Dahl,V. & Abramson, H., *Gapping Grammars*, Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden, 1984.

[Dahl&Abramson,198?]
Dahl, V. & Abramson, H., *Logic Grammars*, in preparation.

[Dahl&St.Dizier,1985]
Dahl,V. & Saint-Dizier,P., *Natural Language Understanding and Logic Programming*, North-Holland, 1985.

[Davie&Morrison,1981]
Davie,A.J.T. & Morrison,R., *Recursive Descent Compiling*, Ellis Horwood - John Wiley, 1981.

[Ganzinger&Giegerich,1985]
Ganzinger, H. & Giegerich, R., *An Experiment in Logic Specification of Compilers and Interpreters*, Draft report, FB Informatik Universität Dortmünd, 1985.

[Hammond,1982a]
Hammond, P., *APES: A detailed description*, Dept. of Computing, Research Report 82/10, Imperial College, London, 1982.

[Hammond,1982a]
Hammond, P., *APES: A user manual*, Dept. of Computing, Research Report 82/9, Imperial College, London, 1982.

[Henderson,1980]
Henderson,P., *Functional Programming*, Prentice-Hall, 1980.

[Horspool&Levey,1986]
Horspool, R.N. & Level, M. *Mkscan - an interactive scanner generator*, to appear in Software - Practice and Experience.

[Knuth,1968]
Knuth, D.E., *Semantics of Context-Free Languages*, Mathematical Systems Theory, vol. 2, no. 2, 1968, pp. 127-145.

[Maluszynski&Nilsson,1981]
Maluszynski, J. & Nilsson, J.F. *A notion of grammatical unification applicable to logic programming languages*, Department of Computer Science, Technical University of Denmark, Doc. ID 967, August 1981.

[Maluszynski&Nilsson,1982]
Maluszynski, J. & Nilsson, J.F. *Grammatical Unification*, Information Processing Letters, vol. 15 no. 4, October, 1982.

[Matsumoto et al,1983]
Matsumoto, Y., et al, *BUP: A Bottom-up parser Embedded in Prolog*, New Generation

Computing, vol. 1, no. 2, pp. 145-158, 1983

[McCord,1985]
McCord, M., *Modular Logic Grammars*, Proceedings ACL Conference, 1985.

[Moss,1979]
Moss, C.D.S., *A Formal Description of ASPLE Using Predicate Logic*, DOC 80/18, Imperial College, London.

[Moss,1981]
Moss, C.D.S.,*The Formal Description of Programming Languages using Predicate Logic*, Ph.D. Thesis, Imperial College, 1981.

[Moss,1982]
Moss, C.D.S., *How to Define a Language Using Prolog*, Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,Pittsburgh, Pennsylvania, pp. 67-73, 1982.

[Nilsson,1986]
Nilsson,U., *AID: An Alternative Implementation of DCGs*, Draft Report, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden.

[Pereira,1981]
Pereira, F.C.N., *Extraposition Grammars*, American Journal of Computational Linguistics, vol. 7 no. 4, 1981, pp. 243-255.

[Pereira,1982]
Pereira, F.C.N. (editor), *C-Prolog User's Manual*, University of Edinburgh, Department of Architecture, 1982.

[Pereira&Warren,1980]
Pereira, F.C.N. & Warren, D.H.D, *Definite Clause Grammars for Language Analysis*, Artificial Intelligence, vol. 13, pp. 231-278, 1980.

[Salim,1985]
Salim, J., *An Expert System Shell for Processing Logic Grammars*, M.Sc. Thesis, University of British Columbia, Vancouver, B.C.

[Sergot,1983]
Sergot, M., *A Query-the-user facility for logic programming*, Integrated Interactive Computer Systems, P. Degano & E. Sandewall (eds) North-Holland, 1983.

[Turner,1979]
Turner, D.A., *A new implementation technique for applicative languages*, Software - Practice and Experience, vol. 9, pp. 31-49.

[Warren,1977]
Warren, David H.D., *Logic programming and compiler writing*, DAI Research Report 44, University of Edinburgh, 1977.

Appendix I. Lexical rules.

```
reserved(div,op(2,intdiv)).
reserved(mod,op(2,mod)).
reserved(if,if).
reserved(then,then).
reserved(else,else).
reserved(while,while).
reserved(do,do).
reserved(read,read).
reserved(write,write).

lexemes(X)    ::= space , !, lexemes(X).
lexemes([X|Y]) ::= lexeme(X) , !, lexemes(Y).
lexemes([])   ::= [].

lexeme(Token) ::=
      word(W) , { is_token(W,Token) }.
lexeme(Con) ::= constant(Con) .
lexeme(Punct)       ::= punctuation(Punct) .
lexeme(op(Binding,Op))  ::= op(Binding,Op) .
lexeme(relop(Rel)) ::= relop(Rel).

is_token(W,Token) :- name(X,W), token(X,Token).

token(X,Token) :- reserved(X,Token) , !.
token(X,id(X)).

space ::= " " , !.
space ::= [10], !. /* carriage return */

num(num(N)) ::= number(Number) , ! , { name(N,Number) }.
number([D|Ds]) ::= digit(D) , digits(Ds).

digit(D) ::= [D] , { is_digit(D) }.

is_digit(D) :- D>47, D<58. /* 0-9 */

digits([D|Ds]) ::= digit(D) , digits(Ds).
digits([])    ::= [].

word([L|Ls]) ::= letter(L) , lords(Ls).

letter(L) ::= [L] , { is_letter(L) }.

is_letter(L) :- L>96, L<123, !. /* a-z */
is_letter(L) :- L>64, L<90. /* A-Z */

lords([L|Ls]) ::= ( letter(L) ) , lords(Ls).
lords([L|Ls]) ::= ( digit(L) ) , lords(Ls).
lords([])    ::= [].

op(1,'+')  ::= "+" .
op(1,'-')  ::= "-" .
```

```
op(2,'*')  ::= "*" .
op(2,'/')  ::= "/" .

relop(le)   ::= "<=", !.
relop(lt)   ::= "<" .
relop(ge)   ::= ">=", !.
relop(gt)   ::= ">" .
relop(ne)   ::= "~=", !.
relop(eq)   ::= "=" .

constant(C)  ::= num(C) , !.

punctuation(lparen)      ::= "(" , !.
punctuation(rparen)      ::= ")" , !.
punctuation(':=')        ::= ":=", !.
punctuation(';')         ::= ";" , !.

/* The following predicates constitute the interface
  between lexical and syntactic analysis. Predicates
  with names starting with 't', eg, tCOLON, are the
  terminals in syntactic analysis.
*/
tLPAREN      ::= [lparen].
tRPAREN      ::= [rparen].
tASSIGN      ::= [':='].
tIF          ::= [if].
tTHEN        ::= [then].
tELSE        ::= [else].
tWHILE       ::= [while].
tDO          ::= [do].
tREAD        ::= [read].
tWRITE       ::= [write].
tSEMICOLON   ::= [';'].
tIDENT       ::= [id(Id)]<:>prefix(Id).
tCONSTANT    ::= [num(C)]<:>prefix(C).
tOP(1)       ::= [op(1,'+')]<:>prefix(add).
tOP(1)       ::= [op(1,'-')]<:>prefix(sub).
tOP(2)       ::= [op(2,'*')]<:>prefix(mult).
tOP(2)       ::= [op(2,'/')]<:>prefix(div).
tOP(2)       ::= [op(2,mod)]<:>prefix(modulus).
tOP(2)       ::= [op(2,intdiv)]<:>prefix(intdivide).
op_com       ::= [relop(lt)]<:>prefix('<').
op_com       ::= [relop(le)]<:>prefix('<=').
op_com       ::= [relop(gt)]<:>prefix('>').
op_com       ::= [relop(ge)]<:>prefix('>=').
op_com       ::= [relop(eq)]<:>prefix('=').
op_com       ::= [relop(ne)]<:>prefix('~=').
```

Appendix II. Syntax and attribute rules.

```
% this is an implementation in Definite Clause Translation Grammars of
% the toy language in David Warren's paper on Logic Programming and
% Compiling which appeared some time ago in Software: Practice and
% Experience. There are slight differences in the target machine,
% a lexical grammar, and a Translation Grammar in place of Warren's
% straight Prolog code.
%
% Here in place of gen_code in the text of the paper we simply write code.

program ::= statements^^S
<:>
code(Dic,Code) ::- S^^code(Dic,Code).

statements ::= statement^^S, st1^^S1
<:>
code(Dic,[SCode,S1Code]) ::-
  S^^code(Dic,SCode),
  S1^^code(Dic,S1Code).

st1 ::= tSEMICOLON, !, statements^^S
<:>
code(Dic,Code) ::- S^^code(Dic,Code).

st1 ::= []
<:>
code(_,[]).

statement ::= tIDENT^^Id, tASSIGN, expression^^E
<:>
code(Dic,[Exprcode,instr(store,Addr)]) ::-
 Id^^prefix(Identifier),
 lookup(Identifier,Dic,Addr),
 E^^code(Dic,Exprcode).

statement ::= tWHILE, test^^T, tDO, statement^^S
<:>
code(Dic,[label(L1),Testcode,Docode,instr(jump,L1),label(L2)]) ::-
 T^^code(Dic,L2,Testcode),
 S^^code(Dic,Docode).

statement ::= tIF, test^^T, tTHEN, statement^^S1, tELSE, statement^^S2
<:>
code(Dic,[Testcode,Thencode,instr(jump,L2),label(L1),Elsecode,label(L2)]) ::-
 T^^code(Dic,L1,Testcode),
 S1^^code(Dic,Thencode),
 S2^^code(Dic,Elsecode).

statement ::= tREAD, tIDENT^^I
<:>
code(Dic,[instr(read,Addr)]) ::-
 I^^prefix(Identifier),
 lookup(Identifier,Dic,Addr).

statement ::= tWRITE, expression^^E
<:>
code(Dic,[Exprcode,instr(write,0)]) ::-
```

```
 E^^code(Dic,Exprcode).

statement ::= tLPAREN, statements^^S, tRPAREN
<:>
code(Dic,Scode) ::- S^^code(Dic,Scode).

test ::= expression^^E1, op_com^^O, expression^^E2
<:>
code(Dic,Label,[Exprcode,instr(Jumpif,Label)]) ::-
 E1^^prefix(Arg1),
 E2^^prefix(Arg2),
 O^^prefix(Op),
 encode_prefix(expr(sub,Arg1,Arg2),0,Dic,Exprcode),
 unlessop(Op,Jumpif).

expression ::= exp1(0)^^E
<:>
(code(Dic,Code) ::- E^^prefix(Prefix),
  encode_prefix(Prefix,0,Dic,Code) ),
(prefix(X) ::- E^^prefix(X)).

exp1(Binding) ::= primary^^P, exp2(Binding)^^E2
<:>
prefix(X) ::- P^^prefix(Primary),
       E2^^prefix(Primary,X).

exp2(Binding) ::= tOP(Q)^^Op, { Binding < Q },
             exp1(Q)^^E1, exp2(Binding)^^E2
<:>
prefix(F,X) ::- Op^^prefix(Operator),
         E1^^prefix(F1),
         E2^^prefix(expr(Operator,F,F1),X).

exp2(_) ::= []
<:>
prefix(F,F).

primary ::=  tCONSTANT^^C
<:>
prefix(num(X)) ::- C^^prefix(X).

primary ::=  tIDENT^^I
<:>
prefix(id(X)) ::- I^^prefix(X).

primary ::=  tLPAREN, expression^^E, tRPAREN
<:>
prefix(X) ::- E^^prefix(X).
```