Prolog in Practical Compiler Writing

J. PAAKKI
Nokia Research Center, P.O. Box 156, 02101 Espoo, Finland

We discuss the experiences gained with implementing the programming language Edison in Prolog. The evaluation of Prolog in this application area is based on a comparison with two other Edison compilers, one written in Pascal (procedural approach) and the other generated using the compiler writing systems PGS and GAG (declarative approach). The crucial hindrance to applying Prolog in practical compiler writing was found to be inefficiency of the standard Prologs. Conceptually, however, Prolog was found to be quite attractive and, as a combination of the procedural and declarative approaches, a promising candidate for the basis of a special compiler writing language that is currently under development.

Received September 1989, revised April 1990

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

1. INTRODUCTION

There are two basic paradigms to produce a compiler.[2] Most commonly a *procedural* approach has been applied where the whole compiler is written in some general purpose programming language. While being the standard way in industrial compiler production, this approach brings on some well-known problems: due to the low level concepts and to the lack of powerful application oriented tools the compilers tend to be rather lengthy, error-prone, and hard to maintain. These drawbacks can be circumvented with a *declarative* approach where the compiler is automatically generated from a high-level specification. This approach, however, has not been generally accepted by the industry, probably because the compiler generators have not yet reached the industrial quality and efficiency requirements.

It would be most feasible to find a middle-of-the-road method between the two extreme approaches: in that case it would be possible to make use both of the flexibility and efficiency of the procedural approach and of the soundness and compactness of the declarative approach. One of the most promising candidates for such a tool is Prolog with its declarative and operational side. The principal suitability of Prolog for writing compilers has been discussed e.g. by Warren[31] and by Cohen & Hickey.[6] However, no comprehensive evaluations of applying Prolog to the implementation of real, non-trivial programming languages have been reported.

The purpose of the experiment reported in this paper is to give some insight into this unexplored corner of Prolog and compiler writing. We have chosen Edison[5] as our case language, on one hand because of its modest size and on the other hand because of its nice selection of advanced programming concepts, such as modularization and concurrency. Thus writing a compiler for Edison made it possible to test Prolog in a realistic case with reasonable effort. The compiler was first written in C-Prolog,[22] and it was later ported into Quintus Prolog[25] in order to get more exact efficiency characteristics.

Since a stand-alone compiler written just in Prolog would give rather vague evaluation results, we have produced the same compiler also by applying the two popular approaches. In that way we can draw more reliable conclusions both from a methodological and from a practical point of view. We used Pascal and the compiler writing system employing GAG[13] and PGS[14] as representatives of the procedural resp. the declarative model.

This paper is organized as follows: in Section 2 we present the Edison implementation as done in Prolog. Section 3 briefly characterizes the procedural and declarative solutions and makes a comparison of all the three methods applied in the experiment. During the experiment we have explicitly noticed the potential power of Prolog in this application area, but also the shortcomings of standard Prologs. These observations have led to the idea of designing a new Prolog dialect especially for compiler writers. This dialect is outlined in Section 4 where also a summary of results is given.

2. THE EDISON IMPLEMENTATION

Since our aim was to evaluate Prolog in compiler writing and not to produce a quality compiler for Edison, we chose the implementation strategy sketched in Figure 1. We divided the process into separate passes, a solution that would be absolutely too inefficient in a real situation. However, in this way we could get a more exact view on Prolog's suitability in different subphases of compilation. In order to make it easier to compare our three approaches we used the same implementation scheme in the procedural and in the declarative compiler as well, except that scanning and parsing were interleaved.

Phases 1–3 constitute the actual compiler. Its input is an Edison program and its output a sequence of instructions for an abstract Edison machine.[5] This abstract machine is represented as an interpreter for the abstract code, written in Pascal (phase 4). Compilation covers the whole language, but for simplicity the library mechanism has been excluded from the interpretation.

2.1 Scanning

We wanted to evaluate the renowned definite clause grammar (DCG) formalism[23] in a realistic situation. That is why we had to separate scanning and parsing into consecutive passes (a DCG expects the source program to be presented as a list of tokens). The scanner was implemented in Prolog as a set of procedures, each recognizing elements of one token class (identifiers, strings, numbers, etc.). In Edison the class of a token can be decided from its first character, except for identifiers

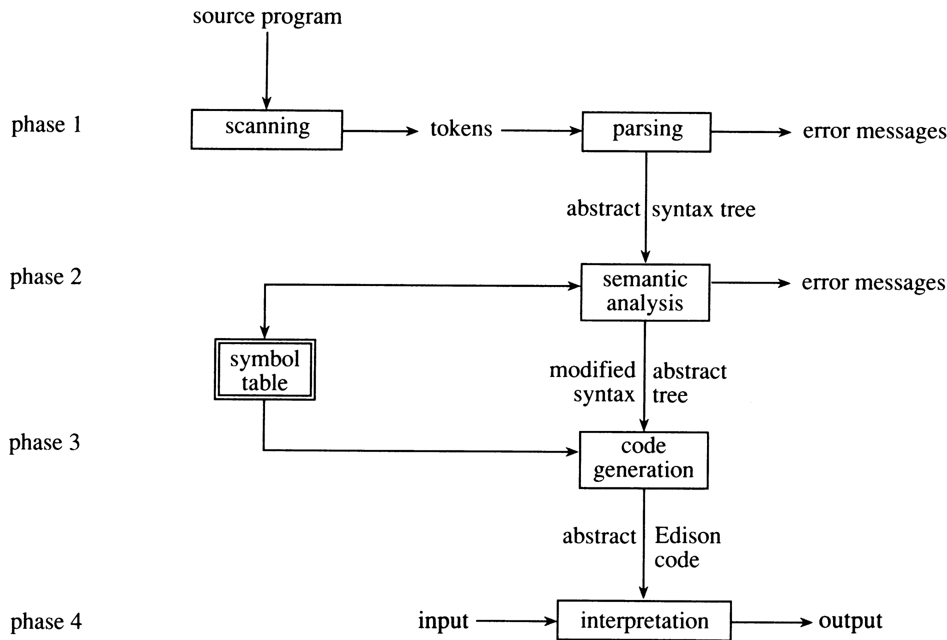

Figure 1. Implementation scheme.

and reserved words that are syntactically identical and have to be distinguished using an additional look-up.

As an example, the scanner procedure for identifiers and reserved words is the following:

```
token(Start,Line,Next,Token):-
  letter(Start), ! ,
  get0(C),
  rest_id(C,Next,R),
  name(X,[Start|R]),
  id_or_reserved(X,Line,Token).
```

This procedure is applied if the last character read (`Start`) is a letter; note the commitment made using cut (`!`). `rest_id` reads the rest of the identifier/reserved word (`R`), stopping on character `Next`. `id_or_reserved` chooses between identifiers and reserved words and returns the representation of the token in `Token`. Because our compiler is a multi-pass one, we have to include line numbers (`Line`) in all the intermediate representations of the source program to support error reporting in subsequent phases.

2.2 Parsing

The parser was written as a DCG with two parameters, for the resulting abstract syntax tree and for a set of error recovery symbols. Syntactic error handling has been implemented using a practical variant of the straightforward 'panic mode' strategy[2] where the input is skipped until a suitable synchronizing token is found and parsing can continue. Besides with special error productions, the DCG is augmented with a nonterminal (producing an empty string) that is placed at the end of central productions and that in error situations synchronizes the input and the parser.

As an example, the DCG fragment for the concurrent (cobegin) statement of Edison is as follows:

```
stat(S,cobegin(Line,Clist)) -->
  [cobegin(Line)],
  process_stat_list([end|S],Clist),
  [end(_)],
  check(S).
```

This corresponds to the context-free production

```
stat -> 'cobegin' process_stat_list
'end' check.
```

Here `S` represents the set of tokens that can follow a concurrent statement or any of its ancestors in the underlying parse tree, and [end|S] represents the set of tokens that can follow the list of process statements. Note that in the latter case using just [end] as the synchronizing token set might have destructive consequences in the case of a missing end: the recovery process would not stop until reaching the next end-symbol, wherever that might appear. Carrying along the `S` set guarantees that synchronization will take place on a token that some enclosing syntactic unit is expecting to deal with. `check` is the augmented recovering non-terminal.

The abstract syntax tree for a source program is constructed as a Prolog term, i.e. in a textual prefix form. The abstract tree does not contain any semantic information; all the semantic processing is postponed until phase 2. In the DCG fragment above, `cobegin(Line,Clist)` is the resulting abstract form of a parsed concurrent statement.

For example, consider the statement

cobegin
  1 **do** x: = 10
also
  2 **do** y: = 20
end

starting on line 100 of the source program. In that case the scanner will convert it into the form

```
[cobegin(100),num(1,101),do(101),id(x,
101),d(: =,101),num(10,101),
also(102),num(2,103),do(103),id(y,103),
d(: =,103),num(20,103),end(104)]
```


Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

and the parser into the abstract form

```
cobegin(100,[also(101,num(1),
[: = (name(x,
101),num(10))]),also(102,num(2),
[: = (name(y,103),num(20))])]).
```

2.3 Semantic analysis

The semantic analyzer checks the validity of the static semantics of the source program on the basis of the abstract tree produced by the parser. As output the analyzer produces a modified tree, again as a Prolog term. The modified tree is constructed to serve the subsequent code generation phase: redundant declarations are removed, symbolic names are replaced with symbol table references, and expression lengths in terms of Edison words are included, as well as some type information.

The analyzer consists basically of a set of procedures, each checking and modifying one specific Edison construction. Since the program's abstract form is a Prolog term, we can pass it as a parameter to the semantic routines. The modified form is also expressed by a parameter. The analyzer has been written in a recursive descent style with one procedure for each nonterminal of the (abstract) syntax. Since no explicit lookahead mechanism nor syntactic error handling is needed (we are now processing syntactically correct abstract programs), the analyzer can be characterized as being a 'translation scheme'[2] which makes it quite readable and thus rather easy to maintain.

The heart of semantic analysis, the symbol table, was implemented in the internal indexed database. The existing literature on Prolog and compiler writing has not discussed this implementation strategy but has instead concentrated on term-based solutions, such as linear lists[6] or ordered binary trees.[31] While being convenient for simple introductory cases, such data structures are not suited for a more complex and practical symbol table representation. First, they are too clumsy for a language with block-structured scope rules, such as Edison. Second, as large data structures they would consume too much space when passed as parameters throughout the analyzer. An analogy can be drawn to attribute grammar based compiler generators where one of the main shortcomings has been space-inefficiency associated with symbol table attributes: usually the problem has been solved with some form of global attributes (e.g. HLP84[15]) or with automatic space management (e.g. GAG). And third, symbol identification would be much slower than in the hashed database; our experiments showed that locating a symbol with a key in a linear list is typically about 10 times slower than locating it in the internal database and even more slower in case of an unsuccessful search.

Symbol insertions were implemented with the database predicate **recorda**, identifications with **recorded** and **instance**, and symbol hidings (when leaving a scope) with **erase**. The predicates were good enough for our experimental purposes. However, since they are general and not designed for special applications, they turned out to be too primitive for some tasks that are normal in language processing. Examples are grouping of local entities and update of symbol entries. These actions made it necessary to introduce some auxiliary solutions, such as keeping track of scopes with lists of symbol references.

As an example of semantic analysis and transformation, we present the fragment for Edison's concurrent statement. The first argument always represents the input tree and the second one the output tree.

```
modify_stat(cobegin(Line,Pstats),
  cobegin(Line,Nr,MPstats)):-
  process_stats(Pstats,MPstats),
  length(Pstats,Nr).

process_stats([Pstat|Rest],
  [MPstat|MRest]):-
  modify_process(Pstat,MPstat),
  process_stats(Rest,MRest).

process_stats([],[]).

modify_process(also(Line,Const,Stats),
  also(Line,Val,MStats)):-
  eval_const(Const,Type,Val),
  (Type==int→true;
  error('process constant must be
  int',Line)),
  modify_stat_list(Stats,MStats).
```

Here *length* returns the number (`Nr`) of process statements (`Pstats`) which is needed in code generation, `eval_const` returns the type (`Type`) and value (`Val`) of the process constant (`Const`), *error* gives an error message, and `modify_stat_list` analyzes and transforms the list of process statements (`Stats`) into `MStats`. Note that line numbers are passed forward to the code generation phase where they are still needed for generating dynamic error checks.

Now the concurrent statement used as an example in Chapter 2.2 will be transformed into the following form:

```
cobegin(100,2,[also(101,1,
[: = (1,var(x_ref),
const(10))],also(102,2,
[: = (1,var(y_ref),const(20))])]).
```

Here `x_ref` and `y_ref` are references to x's and y's entries in the symbol table, respectively. We assume that both x and y will occupy 1 word of memory at runtime.

2.4 Code generation

The task of the code generation phase is to form the abstract Edison code that corresponds to the source program. The resulting code is a sequence (list) of instructions for an Edison stack machine.

The structure of the code generator is roughly the same as the structure of the semantic analyzer: a syntax-directed translation scheme. The code generator is, however, much simpler since no complex symbol table operations are needed. For Edison procedures we had to build an additional table merely for code addresses because the main symbol table in the database could not be supplemented with address information during this phase: the entries in Prolog's internal database are essentially write-once. In this case we used a list implementation since that solution made it possible to backpatch forwarded procedure calls implicitly with Prolog's elegant delayed binding mechanism. An example

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

of such a situation is the next piece of an Edison program:

```
pre proc p
proc q begin p end      ''1''
post proc p begin...end ''2''
```

Now the call for p at ''1'' can be generated by leaving p's address undefined (bound with symbol p) in the code. The unification mechanism will take care of filling in that address automatically (and all the other undefined calls for p as well) when it becomes known at point ''2''.

As an example, the code generator fragment for the concurrent statement is as follows. The input tree is represented by the first and the output code by the last argument.

```
code_stat(cobegin(Line,Nr,Pstats),Sl,
  Adri, Env, Adrs,
  goto(Gdispl)&Pcode&cobegin(Nr,Line,
  Arglist)):-
Pstart is Adri+2,
Cobegin_length is 2*Nr+3,
code_proc_list(Pstats,Sl,Pstart,Env,
  Cobegin_length,Padr,Arglist,Pcode),
Adrs is Padr+Cobegin_length,
Gdispl is Padr-Adri.

code_proc_list([also(Line,
  Val,Stats)|[]],Sl,
  Adri,Env,Cobegin_length,Adrs,[Val,
  Pdispl],process(Temp_length,
  Line)&Scode&also(Adispl)):-
Sstart is Adri+3,
code_stat_list(Stats,Sl,Sstart,Env,
  Temp_length,Sadr,Scode),
Adrs is Sadr+2,
Pdispl is Adri-Adrs,
Adispl is Cobegin_length+2.

code_proc_list([also(Line,Val,
  Stats)|Rest],Sl,Adri,Env,
  Cobegin_length,Adrs,[Val,Pdispl|Rlist],
  process(Temp_length,
  Line)&Scode&also(Adispl)&Rcode):-
Sstart is Adri+3,
code_stat_list(Stats,Sl,Sstart,Env,
  Temp_length,Sadr,Scode),
Rstart is Sadr+2,
code_proc_list(Rest,Sl,Rstart,Env,
  Cobegin_length,Adrs,Rlist,Rcode),
Pdispl is Adri-Adrs,
Adispl is Adrs+Cobegin_length-Sadr.
```

For syntactic convenience the code is represented as an expression over instructions and &-operators. Each instruction and each argument is assumed to occupy one word of memory. The relative address of each instruction is expressed by parameter pairs `Adri-Adrs`, `Sstart-Sadr`, `Rstart-Adrs`. The procedure table is represented by `Env` (the code address list) and `Sl` (the static level). `code_stat_list` generates the code (`Scode`) for the process statement list (`Stats`) and returns the amount of temporary storage needed for executing the statements (`Temp_length`).

For instance, the abstract code for the concurrent statement used as an example in previous chapters would be as follows (for clarity, we give it as a sequence of instructions and not as a Prolog expression). We clarify this example further by using symbolic code addresses instead of relative ones and by substituting the assigning codes with the corresponding source statements:

```
   goto(M)
L1: process(2,101)
   x: =10
   also(N)
L2: process(2,102)
   y: =20
   also(N)
M: cobegin(2,100,[1,L1,2,L2])
N:
```

The first instruction to be executed is `cobegin (M)` which divides the free space evenly among the two processes `L1` and `L2` to be executed simultaneously (or in cyclical order in a single-processor system, as in this case). The *process* instructions check that there is enough local stack space for the processes (2 words in this case which is the amount of space needed for executing the assignments). An `also` statement terminates the corresponding process, and the last process reaching `also` continues execution at `N`. In a single-processor system `also` switches to the next process in case there are processes waiting.

2.5 Interpretation

The abstract code generated by the compiler was made executable by writing an interpreter for that code (in Pascal). This interpreter can be seen as an 'abstract Edison stack machine' that processes its input in the memory organized as a stack. The interpreter was written according to the implementation of Brinch Hansen.[5] Since this phase is outside the actual compilation, we do not discuss it here further.

3. COMPARISON WITH OTHER METHODS

In order to get a better insight into Prolog's suitability in compiler construction the same Edison compiler was produced procedurally and declaratively as well. Since we wanted to make comparisons on each of the compilation subphases, we followed the scheme of Figure 1 also in these versions. The procedural compiler was originally written in Berkeley Pascal[11] and later ported into DEC VAX Pascal,[30] the declarative one was produced with the integrated generator pair PGS[14]–GAG.[13]

The scanners and parsers of the compilers were produced in the order PGS – Prolog – Pascal, and the semantic analyzers and code generators in the order Prolog – Pascal – GAG. The production order has of course significance in the evaluation because all the pieces were written by the same person and thus the solutions made in the first version inevitably had an influence on the subsequent ones.

3.1 The procedural method

The procedural compiler was assembled from three components: scanner/parser, semantic analyzer, code generator. All these were written following a recursive descent style. The source program and all its intermediate

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

representations were given in a textual form which made them suitable for this kind of processing.

The compiler makes use of standard recursive descent techniques as presented e.g. by Welsh & McKeag.[32] The organization of the symbol table was typical for one-pass languages. It is an integrated data structure with four tables: character representations of the symbols, symbol attributes, scopes, and a hashing table. The structure made the implementation of Edison's type and scope rules quite flexible, and most notably no methodological restrictions were met as when using the database in Prolog (see 2.3).

The abstract instructions of the target code were collected in an array before printing. This was due to backpatching which now could be done by traversing lists of incomplete instruction locations in the array. The backpatching process was thus more laborious than in the Prolog version where it was carried out implicitly by unification.

3.2 The declarative method

The declarative version of the compiler was produced with two advanced compiler writing systems: PGS was employed for producing the first pass (integrated scanner and parser), and GAG for producing the other two passes (semantic analyzer, code generator). Actually PGS was used in production of each phase since it is closely coupled with GAG: the attribute evaluators produced by GAG work in a parse tree which is produced by a parser generated by PGS. Again all the intermediate representations of the source program were textual.

PGS is a conventional LALR(1) parser generator based on context-free grammars and a simple attributation facility, similar to YACC.[10] Besides the parser, PGS produces automatically a syntactic error recovery routine, but no scanner. Fortunately the PGS system contains a standard scanner for Pascal which in this case could be used for Edison with small modifications. The attributation facility was used for building the abstract syntax tree for the source program.

The input to GAG is an ordered attribute grammar[12] which can be extended with external abstract data types, implemented in Pascal. In our experiment we specified the compiler's symbol table and target code as abstract data types mainly because of efficiency: as is well known, the strict attribute grammar formalism falls short when representing large variable data structures. Since the symbol table solution in the procedural version (see 3.1) was found flexible enough, we adopted it in this GAG version as well. Also the target code representation was similar to that in the procedural version, i.e. an array where backpatching was carried out through explicit list traversals.

3.3 Evaluation of the methods

It was remarkably easy to produce the Edison compiler with Prolog as the implementation language. The mixture of a declarative framework and embedded conventional programming concepts was found to provide a rigorous tool for writing this kind of a compiler. This observation came by no means as a surprise since some central features of Prolog are intrinsically compilation-oriented, for instance DCGs, logical variables, or trees as the central data type. This suitability of Prolog for rapid implementation can be made more concrete by giving in Table 1 the amount of production days (DAYS) and the number of written lines, excluding comments, (LOC) for each compiler version.

Table 1. Production days

	Prolog	Pascal	PGS/GAG
DAYS	26	27	30
LOC	2400	4800	4500

One may wonder why each version took almost the same amount of time to produce although the Prolog version is only about half as long as the other two. The explanation to this is the production order of the versions: most notably the Prolog version of each compiler phase was produced before the corresponding Pascal version. That is why all the fundamental issues on the problem domain were already solved when writing the Pascal versions and the whole work left was almost just straight coding. If we included the design time in the DAYS row of the Pascal and PGS/GAG versions as well, these figures would be significantly bigger. The relevant conclusions from Table 1 are that we could produce with Prolog a 4-pass compiler for Edison in 5 weeks, and that the amount of code was only half of both the procedural and the declarative code.

Another factor that reduced the working time in the Pascal and PGS/GAG versions is the position of Prolog between these two. In this experiment this view could be verified in practice since it was a natural choice to use the Prolog version of the semantic analyzer and code generator as a model both for the procedural and for the declarative versions: on one hand a compiler phase written in Prolog could easily be transformed into a recursive descent Pascal program, and on the other hand into an attribute grammar. However, we also were careful in relying on the procedural and declarative traditions of building compilers and thus did not produce just three incarnations of a Prolog program, but indeed three conceptually different compilers.

Table 2. Scanner execution times (cpu sec.)

Length	Pascal	Quintus1	Quintus2
10	0.03	0.04	0.4
60	0.07	0.4	2.7
300	0.4	3.3	21.3
900	0.7	7.1	—
1800	1.3	22.6	—

Table 3. Parser execution times (cpu sec.)

Length	Pascal	Quintus1	Quintus2
10	0.01	0.04	0.3
60	0.02	0.3	1.3
300	0.06	2.1	900
900	0.2	7.5	—
1800	0.3	600	—

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

Table 4. Parser execution times for invalid programs (cpu sec.)

length/errors	Pascal	Quintus1
10/5	0.02	0.07
60/10	0.04	0.5
900/50	0.2	11.5

When analyzing the execution times of our three compiler versions, we notice drastically two sides of the Prolog version: while Prolog performs fairly well in semantic analysis and code generation, it is hopelessly slow in scanning and parsing. In Tables 2 and 3 we give execution times for the VAX Pascal and Quintus Prolog versions of the scanner and parser, resp., for legal Edison programs. The Quintus Prolog times are given both for compiled (`Quintus1`) and for interpreted (`Quintus2`) programs. Output times of the produced trees are excluded. Some test runs (–) could not be completed due to memory overflow. All the times are given in seconds of cpu time on a VAX/8800, and the source program lengths in lines of code. The PGS version could not be analyzed in the same environment because the system was not available on this mainframe. However, an analysis on a VAX/750 showed that these phases produced by PGS are at least 5–10 times faster than the corresponding phases written in Prolog.

We also compared the speed of the compiled parsers in VAX Pascal and Quintus Prolog when analyzing syntactically erroneous programs. The cpu times are given in Table 4 for source programs of different length and including different numbers of errors.

As the tables show, these phases implemented in Prolog are absolutely too inefficient to be used in practice. The speed is acceptable only for very small source programs (less than 100 lines); for large source programs the intermediate results might grow so big that they could not be handled at all.

The scanner written in Prolog behaves rather deterministically. Recall that in Edison the token class can be decided from the first character of the token, and that our scanner makes use of this fact by committing to a specific token class after analyzing the first character (see 2.1). After a commitment the scanner never backtracks between different token class procedures, and input is also never backtracked. Thus having only a local 'shallow' form of backtracking[28] implies the Edison scanner written in Prolog to require linear time with respect to the length of the source program. However, even this restricted way of backtracking, combined with recursion as the iteration tool, makes the scanner creep for large program sizes. Also space consumption may be huge (though not analyzed in this experiment in more detail).

The situation is worst in parsing. The standard DCG implementation we employed yields unavoidably a nondeterministic parser, even when we have ordered the alternative productions as well as made extensive use of error productions and synchronizing nonterminals that avoid backtracking to some extent by synchronizing input with parsing (see 2.2). In general, backtrack parsing as a function of program length takes in worst-case exponential time.[3] Thanks to explicit synchronization, the average-case performance of our Edison parser written in Prolog is better than exponential. There are cases, however, that lead to backtracking in a way that exploses the parser time consumption (note the longest source program in Table 3). One example of such a case are assignment statements vs. procedure calls. Pure context-free information of `X` is not sufficient for the parser to choose between the following statements:

```
X: =1;
X(1);
```

In case the current input symbol is identifier `X`, the parser makes a guess for an assignment statement and processes `X` as a variable; however, if the token following `X` happens to be ''('', the parser must backtrack, adjust the input, and try a procedure statement instead. Note that in contrast to scanning, already processed tokens (here `X`) must be shifted from the current history of parsing back to the input stream; thus backtracking can be here characterized as being of a real, 'deep'[28] nature. Another frequently occurring case making the parser extensively backtrack are productions with an empty right-hand side. In a DCG these have to be placed as the last alternatives for the associated nonterminals, and they will thus not be reached by the parser until first having unsuccessfully tried the non-empty alternatives.

Table 5. Semantic analyzer and code generator execution times (cpu sec.)

Length	Pascal	KA-Prolog	C-Prolog	GAG
10	0.3	0.6	1.2	4.8
70	1.6	2.2	6.5	11.5
300	10.1	13.4	45.4	59.2
900	26.3	32.3	113	177

The semantic analyzer and the code generator for Edison, written in Prolog, were surprisingly close in speed to those written in Pascal and produced by GAG.[19] In Table 5 we give total execution times for the Berkeley Pascal, the compiled KA-Prolog,[18] the interpreted C-Prolog, and the GAG versions of the semantic analyzer and the code generator. Here the times are given in seconds of cpu time on a VAX/750.

As can be seen, the compiled Prolog versions do not consume much more time than the Pascal ones. In this case the Prolog programs are most deterministic since (1) the input is guaranteed to be syntactically correct, (2) the scanner and the parser provide to the semantic analyzer additional information in the abstract input tree for making selection between alternative choices possible immediately during unification of the procedures' first arguments; for instance assignment statements and procedure calls are now distinguishable by their principal functors:

```
: = (Var,Expr)      (assignment)
call(Proc,Args)     (procedure call)
```

(3) the semantic analyzer supplements the determinating input for the code generator still further by context-sensitive information. Hence the Edison compiler written in Prolog requires both in semantic analysis and in code generation only linear time with respect to input length. This general observation and the experimental data in Table 5 show that deterministic Prolog programs that

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

can fully utilize unification of trees might not be very much slower than the corresponding Pascal programs, and that they can be even faster than the corresponding automatically generated Pascal programs.

The total times showed that compiling an Edison program of modest size (less than 1000 lines) with the Prolog implementation is typically 6–10 times slower than doing it with the Pascal implementation. For source programs of small size the overall performance of the Prolog implementation is quite close to the performance of the PGS/GAG implementation: the superiority of PGS over Prolog in scanning and parsing is balanced by the superiority of Prolog over GAG in semantic analysis and code generation. When compared to the other two versions, the deeply backtracking parser makes the Prolog version an absolute loser for Edison programs of reasonable size (more than 1000 lines), and for Edison programs containing a significant amount of 'worst-case' code, such as procedure calls.

From a conceptual point of view, Prolog is much related to one-pass attribute grammars (or affix-grammars). In both approaches one can specify a compiler with the syntax-directed method where semantic actions are embedded within context-free productions. In attribute grammars these actions are given as functions, in Prolog as predicates. The relationship becomes more concrete when comparing Prolog with L-attributed grammars[4, 17] that are integrated with top-down parsing, and with compiler writing systems implementing L-attributed grammars, such as LILA(MIRA)[16] and Coco.[26] The execution model of Prolog is a complete depth-first traversal of the search tree obtained by always choosing the leftmost goal, which is analogous to producing the parse tree for a source program in a depth-first left-to-right (top-down) fashion.

The most significant difference between these two compiler writing tools is that Prolog uses non-deterministic parsing whereas the L-attributed systems employ deterministic parsing. The difference is most unfortunate from a practical point of view since nondeterminism makes the parsers produced with conventional Prologs or DCGs too ineffective for competing in efficiency with deterministic ones (see Tables 3–4). Prolog can be related with general attribute grammars as well;[7] we feel, however, that the execution mechanism of Prolog brings it in practice closer to the more restricted L-attributed class.

This experiment revealed that DCGs are very convenient to use for introductory or prototyping purposes but too primitive for practical parsing. Besides having the efficiency problem, they also omit such fundamental aspects in context-free grammars and parsing as left-recursion, regular grammar expressions, integration with scanning, and syntactic error handling. All these could be expressed with Prolog (e.g. we included error handling in our Edison DCG) but then we would lose the compactness and simplicity of the formalism. The result would be a normal recursive descent parser, as written in some procedural language. In our experiment we produced one such parser (in Pascal) which was found rather awkward to do, mainly because of its low-levelness.

Because of these reasons, we feel that the declarative, automated approach is currently the best one for scanning and parsing (even though in our case the scanner was actually not generated). Some recent generators produce scanners and parsers that are nearly as efficient as the corresponding hand-written ones,[8] and their high-level inputs support easy modification, maintenance, and transportation of the products.

When evaluating the tools in the semantic analysis field the support for symbol table management is most crucial. Both Prolog and GAG provide some predefined high-level facilities for this task, while in Pascal one has to build the whole thing from scratch. Some projects on this topic have shown how hard it is to design a set of simple and powerful table management primitives that would be also general enough for a large number of different cases;[15, 27] it was thus no surprise that neither the database or term-based facilities in Prolog nor the standard list operations in GAG were absolutely ideal for Edison.

In principle our idea of unifying two related concepts, symbol tables and databases, worked quite well in Prolog. However, we were forced to use a couple of supporting constructions since some central symbol table concepts were missing in Prolog's database world. Using Pascal we could create a symbol table solution that, while being of low level, was at least flexible and efficient. Recall that we decided to model this implementation in the GAG version as well, but now the interface between the attribute grammar and the external symbol table became rather messy.

Prolog turned out to be strongest in the code generation phase where its sophisticated pattern matching primitive, unification with delayed binding, made it remarkably easy to map the abstract tree to the abstract code. No low-level tricks were needed as in the Pascal and GAG versions to backpatch incomplete instructions. The code generator was also relatively the most efficient component of the Prolog version.

As a summary of the evaluation, none of the analyzed methods and tools is ideal for compiler construction. We feel that there is still much work to be done in all these approaches when trying significantly to simplify the production of reliable, maintainable, and efficient compilers. The Prolog methodology provides clear possibilities to make further progress: while some work in this direction has been reported,[1, 9] the idea is still quite unexplored. Our opinion is that by combining the best characteristics of both the procedural, the declarative, and the Prologian approach a very strong compiler writing tool can be developed. In the next section we briefly introduce one such tool, as a Prolog dialect.

4. SUMMARY AND FUTURE WORK

We have discussed the suitability of Prolog in practical compiler construction. The discussion was based on an experiment where producing a compiler in Prolog for the programming language Edison was compared with producing corresponding compilers using a procedural and a declarative approach.

The experiment showed how well Prolog supports the rapid prototyping method of software production: in about one month we could design and implement a complete Edison compiler (in Prolog) that could easily be transformed into an equivalent, more efficient one (in Pascal). Besides utilizing in this way the operational side of Prolog, we also exploited its declarative side by using

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

the compiler phases in Prolog as models for the corresponding attribute grammars. Thus we can well characterize Prolog as a middle-of-the-road method for compiler construction, although the fixed execution model of Prolog brings it closer to the procedural than to the declarative approach.

Conceptually Prolog provides a number of attractive high-level features for compiler writing, such as DCGs for parsing, an internal database for symbol table management, and unification with logical variables for code generation. The unfortunate fact that makes the Prolog approach unsuitable for constructing production-quality compilers is inefficiency of the current Prolog dialects. Even the many compiler oriented features mentioned above turned out to be too general for this particular application area and thus either too space- and time-consuming or too inflexible. The compiler writing research has generated an extensive set of standard deterministic concepts and techniques that are strangers within the logic programming paradigm, such as lexical, syntactic and semantic error recovery, integration of lexical and syntactic analysis, or scoped and hashed symbol tables. The lack of these standard deterministic compiler writing facilities made our Edison compiler written in Prolog too slow for serious programming; one can state that Prolog is convenient for compiler writers but rather inconvenient for compiler users.

Prolog turned out to be strongest in code generation and semantic analysis. These phases in our Edison compiler were conceptually quite elegant and high-level, as well as reasonably efficient. In scanning and parsing the situation is different: the DCG formalism might be good enough for small toy examples but its conventional implementation is absolutely too naive for larger or more practical cases. We have showed this fact both by a static complexity analysis of the parser generated by a DCG, as well as by an empirical performance study. It must be noted, however, that the problems with DCGs arise merely from its conventional *implementation*, and not from the *notation* which is rather attractive.

Since the conceptual advantages with the Prolog approach are evident, we believe that one can develop a practical Prolog-based compiler writing tool by refining the central features of the language exclusively for compilational purposes and by designing an efficient deterministic implementation for these features. We are currently working on the design and implementation of one such Prolog dialect.[20]

The leading objectives behind the dialect are simplicity, efficiency, compactness, and high-level support on the application area. Some important features of the dialect will be

- determinism; the model has been taken from 'guards' found in many parallel logic programming languages[29]
- practical DCGs; already a prototype implementation of this feature has been produced with emphasis on deterministic parsing and automatic error handling[21]
- a symbol table oriented internal database
- functionality in the simple form of functional terms[24] that roughly correspond to inherited and synthesized attribute values
- redundancy, such as modules, types, and modes.

Acknowledgements

The implementation work has been carried out at the German National Research Center for Computer Science (GMD) in Karlsruhe, and at the Department of Computer Science, University of Helsinki. I would like to thank Prof. Gerhard Goos, Prof. Stefan Jähnichen, Prof. Uwe Kastens, and Dr. Josef Grosch for their valuable comments on the experiment. The comments of Prof. Kai Koskimies, Prof. Reino Kurki-Suonio, Prof. Esko Ukkonen, and an anonymous referee on the draft of this paper are appreciated.

REFERENCES

1. H. Abramson, Towards an expert system for compiler development. Technical Report 87-33, Dept. of Computer Science, University of British Columbia (1987).
2. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*. Addison-Wesley (1986).
3. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Volume I: *Parsing*. Prentice-Hall (1972).
4. G. V. Bochmann, Semantic evaluation from left to right. *Communications of the ACM* **19** (2), 55–62 (1976).
5. P. Brinch Hansen, *Programming a Personal Computer*. Prentice-Hall (1982).
6. J. Cohen and T. J. Hickey, Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems* **9** (2), 125–163 (1987).
7. P. Deransart and J. Maluszynski, Relating logic programs and attribute grammars. *Journal of Logic Programming* **3** (2), 119–155 (1985).
8. J. Grosch, Generators for high-speed front-ends. *Proc. of the Workshop on Compiler Compiler and High Speed Compilation*, pp. 133–144. Akademie der Wissenschaften der DDR (1989).
9. P. R. Henriques, A semantic evaluator generating system in Prolog. *Proc. of PLILP'88, Int. Workshop on Programming Language Implementation and Logic Programming*, edited P. Deransart, B. Lorho, J. Maluszynski, pp. 201–218, LNCS 348, Springer-Verlag (1989).
10. S. C. Johnson, YACC, yet another compiler compiler. Report CS-TR-32, Bell Laboratories, Murray Hill, N.J. (1975).
11. W. N. Joy, S. L. Graham, C. B. Haley, M. K. McKusick and P. B. Kessler, Berkeley Pascal user manual, version 3.0. Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of Berkeley (1983).
12. U. Kastens, Ordered attributed grammars. *Acta Informatica* **13**, 229–256 (1980).
13. U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*. LNCS 141, Springer-Verlag (1982).
14. E. Klein and M. Martin, The parser generating system PGS. *Software – Practice and Experience* **19** (11), 1015–1028 (1989).
15. K. Koskimies, O. Nurmi, J. Paakki and S. Sippu, The design of a language processor generator. *Software – Practice and Experience* **18** (2), 107–135 (1988).
16. J. Lewi, K. DeVlaminck, J. Huens and M. Huybrechts, *A Programming Methodology in Compiler Construction (I and II)*. North-Holland (1979).
17. P. M. Lewis, D. J. Rosenkrantz and R. E. Stearns, Attri-

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

buted translations. *Journal of Computer and System Sciences* **9**, 279–307 (1974).
18. N. Lindenberg, A. Bockmayr, R. Dietrich, P. Kursawe, B. Neidecker, C. Scharnhorst and I. Varsek, KA-Prolog: Sprachdefinition. Arbeitspapiere der GMD 249, Gesellschaft für Mathematik und Datenverarbeitung mbH (1987).
19. J. Paakki, A note on the speed of Prolog. *ACM SIGPLAN Notices* **23** (8), 73–82 (1988).
20. J. Paakki, A Prolog-based compiler writing tool. *Proc. of the Workshop on Compiler Compiler and High Speed Compilation*, pp. 107–117. Akademie der Wissenschaften der DDR (1989).
21. J. Paakki and K. Toppola, An error-recovering form of DCGs. *Acta Cybernetica* (to appear).
22. F. Pereira (ed.), C-Prolog user's manual, version 1.5. EdCAAD, Dept. of Architecture, University of Edinburgh (1984).
23. F. C. N. Pereira and D. Warren, Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* **13**, 231–278 (1980).
24. K. Pöysä, Extending Prolog with functional features (in Finnish). Report C-1989-71, Dept. of Computer Science, University of Helsinki.
25. Quintus Prolog Reference Manual, Version 6. Quintus Computer Systems, Inc., April 1986.
26. P. Rechenberg and H. Mössenböck, *A Compiler Generator for Microcomputers*. Carl Hanser Verlag and Prentice-Hall International (UK) Ltd. (1989).
27. S. P. Reiss, Generation of compiler symbol processing mechanisms from specifications. *ACM Transactions on Programming Languages and Systems* **5** (2), 127–163 (1983).
28. L. Sterling and E. Shapiro, *The Art of Prolog*. The MIT Press (1986).
29. A. Takeuchi and K. Furukawa, Parallel logic programming languages. *Proc. of the Third International Conference on Logic Programming*, edited E. Shapiro, pp. 242–254. LNCS 225, Springer-Verlag (1986).
30. VAX Pascal Reference Manual. Digital Equipment Corporation (1987).
31. D. Warren, Logic programming and compiler writing. *Software – Practice and Experience* **10** (2), 97–125 (1980).
32. J. Welsh and M. McKeag, *Structured System Programming*. Prentice-Hall International (1980).

Downloaded from http://comjnl.oxfordjournals.org at West Virginia University on April 23, 2010

Correspondence

Sir,

The recent paper by R. G. Dromey and T. A. Chorvat in the Journal[1] prompts me to set the record straight on 'Program Inversion'. In the late 1960s I was a member of a programming team led by Michael Jackson. Individually and collectively, members of this team devised many of the techniques that form the basis of the Jackson Structured Programming methodology.[5] (I don't intend to detract from Jackson's contribution; there is a big gap between a collection of *techniques* and an integrated *methodology*. The synthesis is Jackson's own.)

Much of the team's work involved systems programming in assembly language. Structured programming was supported by macros, and the resulting programs looked very much like Jackson's Program Design Language, even including the 'posit' conditional structure. Wherever possible, programming techniques were supported by software, usually in the form of macros.

'Program Inversion' stems from an idea originated by Brian Boulter and myself, and was originally aimed at software re-use rather than resolving boundary clashes. We recognised that the same algorithm could appear in different guises, which we would now call its different inversions. Since we both had a hardware background, our original idea was to implement library procedures as finite-state machines in a universal form that could be used directly in any of their possible inversion modes. (Vestiges of this method can be found in my 'Lateral Programming' methodology.[2]) This proved to be inefficient, and the code was hard to read, so we then adopted a manual method based on transforming flowcharts. When flowcharts became passé, this evolved into the co-routine-derived approach that is familiar from Jackson's book.

I mention this to make it clear that the connections between JSP and finite-state automata, pointed out by Hughes,[4] were always understood by Jackson's team. But Jackson knew his audience – mainly Cobol programmers without formal Computer Science education – well enough to know that the *explanation* of his methodology should be directly in terms of the program text. It is worth noting here that Hughes' speculation that any pair of programs coupled by inversion can be reduced to a single program is certainly true. If no other means were available, we can find the state-machines for each program of the pair, then define a larger machine whose state space is the cartesian product of the two individual state spaces. Alternatively, we can take the inverted program, express it as a **case** statement based on its internal state, work backwards to its structure diagram, then substitute the diagram wherever the inverted program is called from the other. Of course, this method is rarely practical, and the resulting program would usually be very difficult to understand.

This is where Dromey and Chorvat have made an important contribution. Jackson's team understood that boundary clash problems *could* be resolved by Dromey and Chorvat's 'forced synchronisation' method. In terms of regular expressions, we were well aware that $x^* = (x^*)^* = (x|\varepsilon)^*$. As evidence, consider my 1980 paper on file updating,[3] where forced synchronisation (in all but name) is used to resolve the clash between the old and new master files. However, our use of this method relied on insight by the programmer, and Dromey and Chorvat must be commended for devising their understandable and generalised 'prototype' methodology to replace what was previously a programming trick.

One final comment. Since Hughes' paper, it has often been stated that JSP corresponds to the algebra of regular expressions. This is a reasonable mistake, because Jackson's book never deals with recursive problems or recursive data structures. But this is because it was addressed primarily to Cobol programmers, and Cobol does not support recursion. JSP is more closely related to the LL(1) grammars. For problems that are dominated by the structure of their input data, it is essentially the same method as recursive-descent parsing. That Jackson (and I) recognised this is belatedly acknowledged in the notes on First and Follow sets that appear in his later book on system development.[6] In fact, JSP can handle regular expressions only if they convert directly to *deterministic* automata, and are therefore a subset of LL(1) grammars.

Yours faithfully

BARRY DWYER

Department of Computer Science,
The University of Adelaide,
Box 498, G.P.O.,
Adelaide,
South Australia 5001

References

1. R. G. Dromey and T. A. Chorvat, Structure clashes – an alternative to program inversion. *The Computer Journal* **33** (2), 126–132 (1990).
2. B. Dwyer, Lateral programming: a proven technique. In *Tutorial: Software Design Strategies*, edited G. D. Bergland and R. D. Gordon 214–222, IEEE (1979).
3. B. Dwyer, One more time – how to update a master file. *Communications of the ACM* **24** (1), 3–8 (1981).
4. J. W. Hughes, A formalization of explication of the Michael Jackson method of program design. *Software Practice and Experience* **9** (4), 191–202 (1979).
5. M. A. Jackson, *Principles of Program Design*. Academic Press, London (1975).
6. M. A. Jackson, *System Development*. Prentice-Hall, Englewood Cliffs, NJ (1982).