A Generator for Production Quality Compilers

Hanspeter Mössenböck

ETH Zürich
Institut für Computersysteme
ETH-Zentrum, CH-8092 Zürich

Abstract

This paper presents a compiler description language and its implementation Coco/R (Compiler Compiler for Recursive Descent). Coco/R reads an attributed EBNF grammar of a language and translates it into a recursive descent parser and a scanner for that language. The programmer has to supply a main program that calls the parser and semantic modules that are called from within the parser. Coco/R evolved from two predecessors: the scanner generator Alex [Möss86] and the parser generator Coco [ReMö89]. Their input languages were merged and simplified due to our experiences with these tools over several years (a similar tool with a slightly different motivation also emerged from Alex and Coco [DoPi90]). Using Coco/R, compilers can be generated that are as efficient as hand-coded and carefully optimized production quality compilers. Almost as important as efficiency is the simplicity and adequacy of the system. Programmers are not willing to use a tool if it does not come in handy to their work, if it uses an arcane notation or a bulk of options and special cases. Coco/R puts simplicity and efficiency over power.

1. INTRODUCTION

Sometimes the most simple techniques are also the most efficient ones. While hand-written compilers usually are implemented in recursive descent, most of the generated compilers use table-driven LL(1) or LALR(1) techniques. After experiences with several parsing methods [Möss87] I returned to recursive descent parsing since I believe that there is hardly anything so efficient, and at the same time so convenient, as this technique. Its advantages are:

- *No table access*. For table-driven parsers to be space-efficient the tables have to be compressed and accessing them needs decoding. In recursive descent parsing, recognizing the current symbol requires only a simple comparison.
- *Easy semantic evaluation*. Semantic actions are embedded directly into the parser and do not have to be collected into a procedure that is called whenever an action is to be executed. Every production corresponds to a parsing procedure with its own scope for local variables.

- *Transparency*. Recursive descent parsers can be read and understood while the tables of a table-driven parser remain a mystery for the programmer.
- *Controlling the parser*. Since parsing and semantic analysis are intertwined so closely one can control the parser from the semantic actions. This makes it possible to parse languages whose grammars are not LL(1).
- *Adequate parser size*. Table-driven parsers are usually smaller than recursive descent parsers. However, while their size is nearly the same for both small and large grammars, the size of a recursive descent parser depends on the size of the grammar. For small grammars a recursive descent parser is probably smaller than a table-driven parser.

One of the major problems with recursive descent parsing is that sophisticated error-handling is harder to implement than for table-driven parsers. The error-handling technique presented in Section 5 attacks this problem.

2. THE COMPILER DESCRIPTION LANGUAGE

The compiler description language Cocol/R consists of four parts:

- A context-free EBNF grammar describing the structure of the input to be parsed.
- Attributes attached to the nonterminals of the grammar. They denote the result of the translation of that nonterminal (synthesized attributes) or the context to be used for the translation of that nonterminal (inherited attributes). Attributes are enclosed by angle brackets (e.g. <type>).
- Semantic actions that may occur at any point on the right-hand side of a production. Semantic actions are statement sequences written in the implementation language of the target compiler (here Oberon [Wirth88]) and bracketed by "(." and ".)". They can use variables and attributes declared local to a production or global to the whole grammar.
- A specification of the lexical properties of the input such as the structure of tokens or the form of comments.

Descriptions of this kind are often called *attributed grammars*. However, attributed grammars originally were conceived as static descriptions [Knuth68] where the semantic actions specify dependencies between attributes without giving an order in which the attribute values are to be computed. In Cocol/R a grammar is a dynamic description (an algorithm): a semantic action is simply a piece of code that is executed after parsing the symbol to the left and before parsing the symbol to the right of it. Grammars of this kind are also called *syntax-directed translation schemes* [ASU86]. In fact, Cocol/R grammars can be regarded as a short-hand notation for writing recursive descent compilers.

The following example should give an impression of a Cocol/R grammar (a full specification of Cocol/R can be found in [Möss90]). Consider the processing of variable declarations for a Pascal-like language. The context-free syntax of this construct is

```
VarDeclaration = Ident {"," Ident} ":" Type ";".
```

By simply writing down this production, one already gets a parser that can check variable declarations syntactically. To process them semantically as well, one has to think about how variable declarations are translated. The translation of an identifier probably yields its name, and the translation of a type yields some type information. These are synthesized attributes of the nonterminals *Ident* and *Type*. The effect of the whole translation should be to enter the variables into a symbol table. We keep the symbol table as an abstract data structure into which *VarDeclaration* enters the variables without returning a synthesized attribute. However, it has an inherited attribute that specifies the next available address for the new variables. The nonterminals can then be seen as sub-translators with the following tasks:

Ident <name>	recognize an identifier and return its name.
Type <typ>	recognize a *Type* and translate it into some type information *typ*.
VarDeclaration <adr>	recognize a *VarDeclaration*. *adr* denotes the next free address in the variable space before and after the processing of *VarDeclaration*.

The only remaining task now is to write semantic actions that enter the variables into the symbol table and compute their addresses. With Oberon [Wirth88] as the language of semantic actions this reads as follows:

```
VarDeclaration <VAR adr: LONGINT>
                          (.  VAR obj, obj1: SymTab.Object; typ: SymTab.Type;
                              n, a: LONGINT; name: ARRAY 32 OF CHAR; .)
=  Ident <name>           (.  obj := SymTab.Enter(name); obj^.next := NIL; n := 1 .)
   { "," Ident <name>     (.  obj1 := SymTab.Enter(name); obj1^.next := obj; obj := obj1; n := n + 1 .)
   }
   ":" Type <typ> ";"     (.  adr := adr + n * typ^.size; a := adr;
                              WHILE obj # NIL DO
                                 a := a - typ^.size; obj^.adr := a; obj^.typ := typ; obj := obj^.next
                              END .) .
```

Note that the attribute of the left-hand side nonterminal is declared with its type and the keyword VAR which denotes that *adr* is both an inherited and a synthesized attribute. Attributes on the left-hand side of a production are called formal attributes in contrast to actual attributes appearing on the right-hand side of a production. This naming reflects the similarity between attributes and parameters in programming languages. A production constitutes a scope for locally declared objects (*obj*, *obj1*, etc.). In addition, globally declared or imported names can be accessed (e.g. *SymTab*). The format for writing down a production is free. However, it is wise to shift syntactic parts to the left and semantic parts to the right. This gives a clear separation between syntax and semantics and makes the grammar more readable.

Whereas terminals enclosed in quotes denote themselves (e.g. keywords), structured terminals, such as identifiers, have to be declared in a scanner specification section:

```
CHARACTERS
   letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
   digit = "0123456789".
TOKENS
   identifier = letter {letter | digit}.
```

The lexical value of a token (its character sequence) can be accessed by its position *pos* in a buffer and by its length *len*. Both variables are exported by the generated scanner. With this information we can formulate the *Ident* production.

```
Ident <VAR name: ARRAY OF CHAR>
= identifier                    (. Scanner.GetName(Scanner.pos, Scanner.len, name) .) .
```

There are another few constructs to describe the structure of comments or to recognize tokens with a lookahead of more than one character, but essentially this is all a programmer has to know about the input language of Coco/R. The effort to learn this language is small, since semantic parts are written in a familiar programming language and syntactic parts are based on the well-known EBNF grammars. For an Oberon compiler a grammar with 892 lines is translated into a scanner and a parser that makes up 1836 lines.

3. THE GENERATED SCANNER

The scanner is generated from the token declarations and from the literal strings occurring in the productions. The token declarations (regular expressions in EBNF) are translated into syntax graphs from which a deterministic finite automaton is generated. This process is sketched in Figure 1 (the algorithms for the manipulation of the automaton are described in [Möss87]).

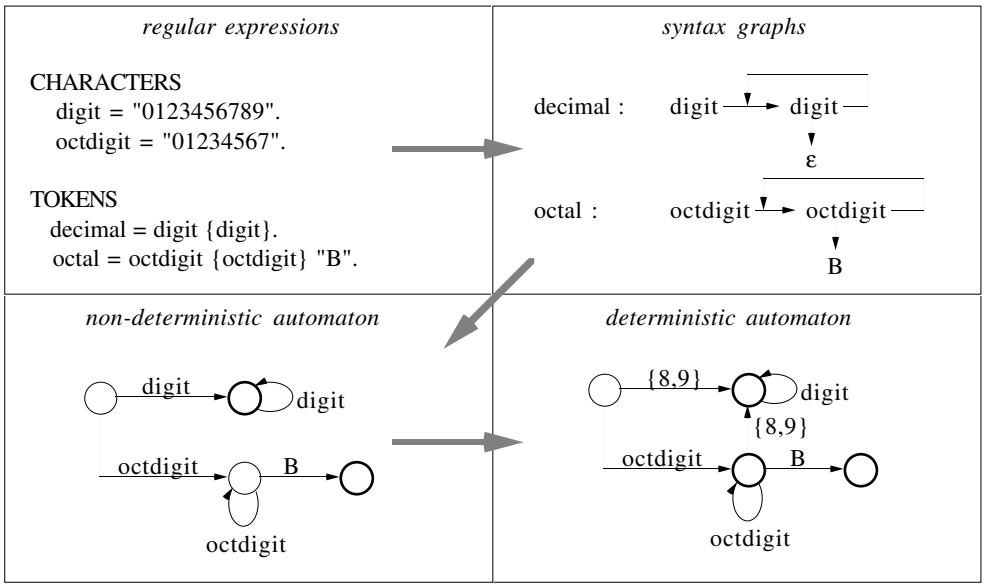
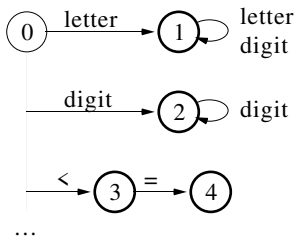


Figure 1 Transformation of regular expressions into a deterministic finite automaton

The automaton is not generated directly from the regular expressions but from syntax graphs. This allows making it more deterministic from the beginning, thus simplifying the later algorithms. Note that tokens may have very similar structures differing only in their last characters (like *decimal* and *octal*). Such ambiguities are resolved by Coco/R automatically.

To make the scanner as efficient as possible, the automaton is not implemented as a table-driven algorithm but it is casted into code. This saves table accesses at the cost of code size. A scanner that recognizes identifiers, numbers, and the literals "<" and "<=" is shown in Figure 2.




```
startState[letter] = 1
startState[digit]  = 2
startState["<"]    = 3
…
```

```
PROCEDURE Get(VAR sym: INTEGER);
  …  (* ch and chPos are global; set by NextCh *)
  state := startState[ch]; pos := chPos; len := 0;
  LOOP
    NextCh; INC(len);
    CASE state OF
      1: IF (ch>="A") & (ch<="Z") OR (ch>="a") & (ch<="z")
         OR (ch>="0") & (ch<="9") THEN (*state := 1*)
         ELSE sym := ident; CheckLiteral(sym); RETURN
         END
    | 2: IF (ch>="0") & (ch<="9") THEN (*state := 2*)
         ELSE sym := number; RETURN
         END
    | 3: IF ch = "=" THEN state := 4
         ELSE sym := lss; RETURN
         END
    | 4: sym := leq; RETURN
    …
    END
  END
END Get;
```

Figure 2 A scanner derived from an automaton

NextCh returns the next input character *ch* and *CheckLiteral* checks if the just recognized token is a literal (e.g. when an identifier has been recognized it must be checked if it is a reserved keyword). Unfortunately, the state transitions of the automaton have to be implemented by a loop and a test of a state variable. This could be done more efficiently if a Goto statement were available in Oberon. An experiment showed that generating the scanner in Assembler using Goto statements (and other optimizations) speeds it up by 30 %.

Input buffering

The most time-consuming task in scanning is reading the source text. The scanner can be sped up significantly if reading can be made faster. Reading a text character by character is usually slower than reading it in blocks that correspond to disk sectors. With the large memories available today, it is even possible to read the whole source text into memory *at once*. In the Oberon system this is almost three times faster than reading it character by character from the file system buffer. Even large source programs rarely exceed 50 kilobytes. With several megabytes of memory available, this "waste" of 50 kilobytes seems justified if scanning can be speeded up so dramatically (the overall run time of a typical compiler is improved by 30 %).

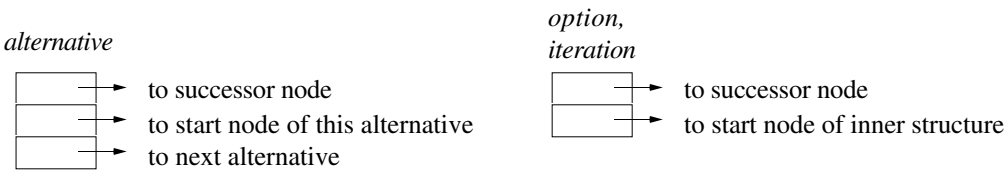
Having the whole source text in memory has yet another advantage: the source text can be used as a name list. The text of tokens like identifiers no longer has to be copied to a separate name list but can remain where it is, namely in the input buffer. One simply has to remember its position and its length. This idea is in accordance with the principle that during scanning every character should be "touched" as little as possible [Waite86].

Another advantage of this technique is that it permits to move back the input pointer to any previous position. This is useful for handling tokens which require a lookahead of more than one character. To recognize such tokens the right-hand context has to be analyzed, too. After the token and its context have been scanned, the input pointer is simply decreased by the length of the lookahead, so that this text will be reread by the scanner.

4. THE GENERATED PARSER

The parser is generated from the productions of the compiler description. It is implemented in recursive descent with semantic actions embedded between parsing statements. The parsing procedures cannot be generated on-the-fly while the grammar is processed since that would require to know the terminal start symbols of all nonterminals. These sets can only be computed when the whole grammar has been read. Therefore, the productions are first translated into syntax graphs, then the terminal start symbols are computed, and finally the parsing procedures are generated. The syntax graphs are also used to perform certain grammar tests (completeness, redundancy, LL(1) property).

Let's take a closer look at the syntax graphs. A node is generated for every symbol in the grammar and for every semantic action (a semantic action node contains the position and the length of the action in the source text). A sequence of symbols and actions results in a sequence of nodes. Alternatives, options and iterations are modelled by special nodes:



A production like

```
Expression <VAR x: Item>          (.  VAR y: Item; op: INTEGER; .)
=  SimExpr <x>
   [  Relop <op>  SimExpr <y>     (.  GenOp(op, x, y) .)
   |  "IN" SimExpr <y>            (.  GenIn(x, y) .)
   |  "IS"                        (.  IF x.mode >= Typ THEN Error(…) END .)
      qualident <y>               (.  IF y.mode = Typ THEN GenTypTest(x, y) ELSE Error(…) END .)
   ].
```

is translated into the following graph

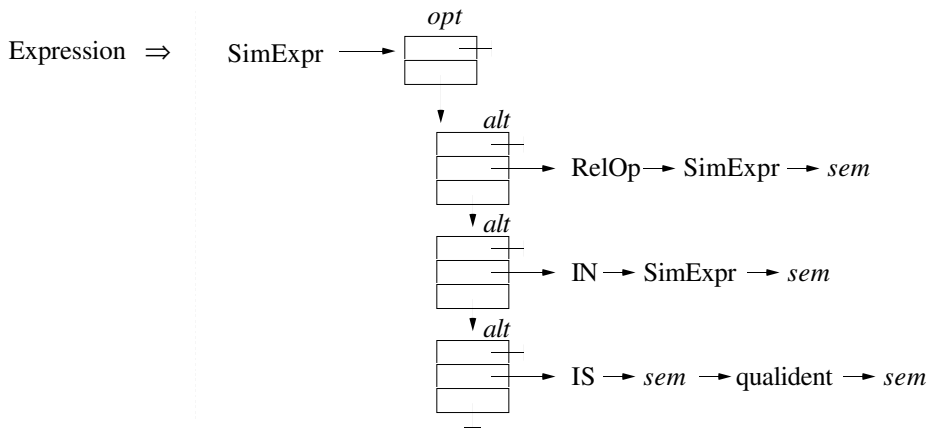


Figure 3 Syntax graph for the generation of parsing procedures

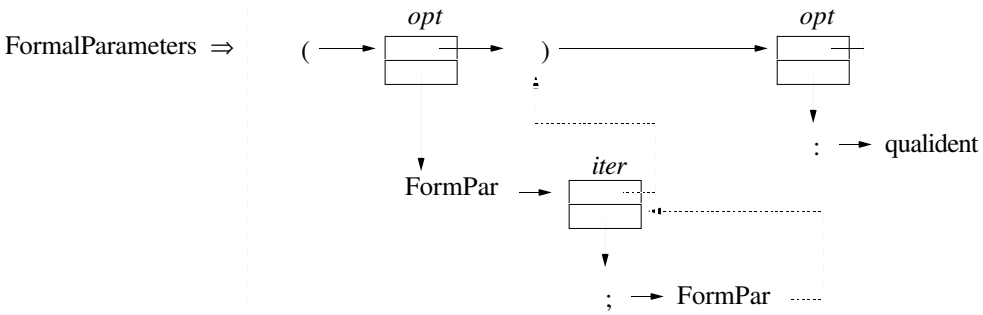
Note that these graphs are different from the graphs used for scanner generation (Figure 1). Alternatives, options and iterations are represented by special nodes. This makes the graphs better suited for the generation of recursive descent parsers. Having the graphs and the sets of terminal start symbols it is easy to generate parsing procedures. The graph from Figure 3 is translated into the following procedure (*sym* is the lookahead token and *Get* is the scanner):

```
PROCEDURE Expression (VAR x: Item);
   VAR y: Item; op: INTEGER;
BEGIN SimExpr(x);
   IF sym IN {eql, neq, lss, leq, gtr, geq, in, is} THEN
      IF sym IN {eql, neq, lss, leq, gtr, geq} THEN
         Relop(op); SimExpr(y); GenOp(op, x, y);
      ELSIF sym = in THEN
         Get(sym); SimExpr(y); GenIn(x, y);
      ELSE
         Get(sym); IF x.mode >= Typ THEN Error(…) END;
         qualident(y); IF y.mode = Typ THEN GenTypTest(x, y) ELSE Error(…) END;
      END
   END
END Expression;
```

A more interesting example is the following production, which also contains iterations (for brevity, semantic actions are not shown).

```
FormalParameters = "(" [FormPar {";" FormPar} ] ")" [":" qualident].
```

The corresponding syntax graph is



Dotted arrows denote pointers to the successors of inner structures. They help in the computation of start and successor sets. The graph is translated into the following procedure (*Expect*(*s*) tries to match *s* with the lookahead symbol and emits an error if it cannot):

```
PROCEDURE FormalParameters;
BEGIN
   Expect(leftpar);
   IF sym IN {ident, var} THEN
      FormPar;
      WHILE sym = semicolon DO Get(sym); FormPar END
   END;
   Expect(rightpar);
   IF sym = colon THEN Get(sym); qualident END
END FormalParameters;
```

The translation of EBNF productions into parsing procedures is straightforward and leads to compact parsers. There are only few cases where a procedure could be written more efficiently by hand.

5. SYNTAX ERROR-HANDLING

Good and efficient error-recovery is difficult in recursive descent parsers since little information about the parsing process is available when an error occurs. A generally used method [Wirth76] is to dynamically collect the legal successors of all nonterminals that are currently on the parsing stack. If an error occurs the input is skipped until a symbol is found that is in the successor set. The parsing stack is then unrolled up to a point where parsing can continue with that symbol. This technique, although systematically applicable, slows down error-free parsing and inflates the parser code.

Another technique has therefore been suggested in [Wirth86]. Recovery takes place only at certain synchronization points in the grammar. When an error occurs it is reported but parsing continues up to the next synchronization point where the grammar and the input are synchronized again. Usually there are only few synchronization points in a grammar. The symbol sets used for synchronization there do not have to be collected at run rime but can be

precomputed at parser generation time. This method is better than the previous one, since it does only slightly affect error-free parsing and keeps the parser small. However, it requires the designer of the grammar to mark synchronization points explicitly.

Synchronization points. In Cocol/R a synchronization point is specified by the keyword SYNC at the appropriate point in a production. Good candidates for synchronization points are locations where particularly safe symbols (such as keywords) are expected and that are often visited by the parser. Typical examples are the beginning of a statement (IF, WHILE, REPEAT, etc. are expected here), the beginning of a declaration (CONST, TYPE, VAR, PROCEDURE expected) or the beginning of a structured type (RECORD, ARRAY, POINTER, etc. expected).

A synchronization point is translated into a loop that skips all symbols which are not expected at this point (except the end-of-file symbol). The set of these symbols is precomputed at parser generation time. The following example shows two synchronization points and their counterparts in the generated parser.

production	*generated parsing code*
Declarations =	

SYNC	WHILE ~(sym IN {const, type, var, proc, begin, end, eof}) DO
	Error(…); Get(sym)
	END;

{	WHILE sym IN {const, type, var, proc} DO
("CONST" {ConstDecl ";"}	IF sym = const THEN Get(sym); …
| "TYPE" {TypeDecl ";"}	ELSIF sym = type THEN Get(sym); …
| "VAR" {VarDecl ";"}	ELSIF sym = var THEN Get(sym); …
| ProcDecl	ELSE ProcDecl
)	END;
SYNC	WHILE ~(sym IN {const, type, var, proc, begin, end, eof}) DO Error(…); Get(sym) END
}.	END

To avoid spurious error messages, an error is only reported if a certain amount of text (e.g. 10 characters) has been correctly parsed since the last error.

Weak symbols. The knowledge of synchronization points is already sufficient to recover from errors. However, recovery can be improved if the parser also knows about "weak" symbols that are often mistyped or missing (such as semicolon). These symbols are marked in the grammar by the keyword WEAK. If the parser tries to recognize a weak symbol and does not find it, it reports an error and skips the input until a legal successor of the expected symbol is found (or a symbol that is expected at any synchronization point; this is a useful heuristic that avoids skipping safe symbols). The following example shows the translation of a weak symbol ":=" (it is considered as weak because it is often mistyped as "=").

production	*generated parsing code*
Statement =	
ident	Expect(ident);
WEAK ":="	Weak(becomes, {*start symbols of Expression*});
Expression .	Expression

The procedure *Weak* is implemented as follows:

```
PROCEDURE Weak(s: INTEGER; expected: Set);
BEGIN
   IF sym = s THEN Get(sym)
   ELSE
      Error(s); WHILE sym ∉ expected ∪ {symbols expected at synchronization points} DO Get(sym) END
   END
END Weak;
```

Weak symbols give the parser another chance to synchronize in case of an error. Again, the set of expected symbols is precomputed at parser generation time and causes no run time overhead in error-free parsing.

When an iteration starts with a weak symbol, this symbol is called a *weak separator*. A mistyped separator is especially harmful, since it causes the iteration to terminate. Therefore, weak separators are handled in a special way. If they cannot be recognized, the input is skipped until a symbol is found that is contained in one of the following three sets:

- α symbols that may follow the weak separator
- β symbols that may follow the iteration
- γ symbols expected at any synchronization point (including eof)

The following example shows the translation of a weak separator

production	*generated parsing code*
StatSequence =	
Stat	Stat;
{ WEAK ";" Stat}.	WHILE WeakSep(semicolon, α, β) DO Stat END

In this example, α is the set of start symbols of a statement (ident, IF, WHILE, etc.) and β is the set of successors of a statement sequence (END, ELSE, UNTIL, etc.). Both sets are precomputed at parser generation time. *WeakSep* is implemented as follows:

```
PROCEDURE WeakSep(s: INTEGER; sySucc, iterSucc: Set): BOOLEAN;
BEGIN
   IF sym = s THEN Get(sym); RETURN TRUE
   ELSIF sym ∈ iterSucc THEN RETURN FALSE
   ELSE Error(s); WHILE sym ∉ sySucc ∪ iterSucc ∪ γ DO Get(sym) END;
      RETURN sym ∈ sySucc  (*TRUE means "s inserted"*)
   END
END WeakSep;
```

The observant reader may have noticed that the set β in the example contains the successors of a statement sequence in *any possible* context. This set is too large. If the statement sequence occurs in a repeat statement, only UNTIL is a legal successor, but not END or ELSE. I tolerate this fault, since it allows us to precompute the set β at parser generation time. The occurrence of END or ELSE is very unlikely in this context and can only lead to incorrect synchronization, causing the parser to synchronize again.

The following example demonstrates that the above method yields good error-recovery. I generated an Oberon compiler and compiled the following erroneous program taken from [Wirth86]. The parser recovered surprisingly well.

```
     MODULE Error;
     CONST M := 10, N = 100  X = 10;
***          ^ "=" expected
***               ^ ";" expected
***                          ^ ";" expected
     VAR , a, b, c;
***      ^ unexpected symbol in Block

     PROCEDURE P;
     BEGIN
       s := 0; a = 5 * (b - 1 END;
***              ^ error in Stat
***                       ^ error in Stat
***                               ^ ident expected

     BEGIN
       > a > b;
***    ^ unexpected symbol in Stat
***         ^ error in Stat
       WHILE a DO
         BEGIN > b; - c := 0;
***      ^ unexpected symbol in Stat
***            ^ unexpected symbol in Stat
***                 ^ unexpected symbol in Stat
         WHILE a > 0 BEGIN
***                  ^ "DO" expected
           IF ODD a c := c * - b;
***                ^ error in Factor
***                  ^ error in Stat
***                           ^ error in Factor
           b := 2 * b a := a / 2
***                    ^ error in Factor
         END;
         P := 0; P; 666;
***                  ^ unexpected symbol in Stat
       END .
***        ^ ";" expected
***         ^ "END" expected
```

The proposed error-recovery technique is cheap. It costs only a check at every synchronization point and therefore does not slow down error-free parsing. The code for error-handling makes up 10 % of the parser code (without semantic actions).

Oberon parser without error-handling	3019	Bytes	(object code)
Error-handling procedures (fixed size)	248	Bytes	
Synchronization points, weak symbols	81	Bytes	

6. MEASUREMENTS

Coco/R is implemented in Oberon on a Ceres workstation. There is another Oberon implementation for Sun Sparcstation as well as Modula-2 implementations for Macintosh and MS-DOS.

I compared an Oberon compiler generated by Coco/R with a hand-written Oberon compiler and measured the time to compile a 867-line Oberon program (7169 tokens or 24254 characters) on a Ceres workstation with a NS32532 processor running at 25 MHz and on a Sun Sparcstation 1 with a SPARC processor running at 20 MHz. The back-end of both compilers is the same, only the scanner and the parser are different.

Compilation time of an 867 line program	Ceres (sec)	Sparc (sec)
Hand-written compiler	3.6	n.a.
Generated compiler	3.0	0.75

The generated compiler is 20 % faster than the hand-written compiler. This is due to the fact that the generated compiler reads the source text into main memory at once, while the original compiler reads it character by character. Without this improvement the generated compiler is about 10 % slower than the hand-written one. The speed of the several phases is:

Speed of the generated compiler	Ceres (tokens/sec)	Sparc (tokens/sec)
Scanning	15 360	39 390
Parsing	51 130	73 910
Total Compilation	2 400	9 560

Scanning, as well as the whole compilation, is considerably faster on the Sparcstation because of its faster disk access while for parsing the Ceres catches up with Sparc due to its faster set operations. I avoided the measure "lines per second" since it depends on the density of the source code. One may assume 5-8 tokens per line as a mean value. Comparing the object code of the two compilers on the Ceres yields the following results:

Object code in bytes (on Ceres)	Scanner	Parser (incl.sem.actions)
Hand-written compiler	3 672	11 740
Generated compiler	3 944	12 236

Compared to the table-driven compilers generated by the old Coco system the compilers generated by Coco/R are more than twice as fast. This is due to improvements in the scanner and efficient semantic processing in the recursive descent parser.

7. CONCLUSION

The field of compiler construction and compiler generation has long become mature. Various methods of scanning, parsing and semantic evaluation have been studied extensively. Time has come now to make good use of that knowledge and to design tools that are as simple and as efficient as we can achieve today. Coco/R is an effort in this direction. It is definitely not a prototype but a production quality tool with simplicity and efficiency as a major design goal.

A simple tool should be based on a familiar notation, it should require as little input as possible and its function should be transparent and predictable to its user. For the generated parts to be able to compete with production quality compilers they must be small and fast. This can be achieved by using single-pass compilation with semantic actions executed during parsing, by burning the grammar into code instead of using table-driven techniques and by using efficient error-handling that does not slow down error-free parsing. For many translation tasks, such as the processing of small command languages, this kind of compilation scheme is quite sufficient (more powerful schemes would even be an overkill) and also proper compilers for many real programming languages such as Modula-2 or Oberon do not require heavier guns.

Coco/R matches the above requirements to a large degree. The compilers generated by it are comparable to hand-written compilers both in speed and in size. The input language Cocol/R is easy to learn since it is small and based on familiar concepts (EBNF grammars and a general purpose programming language). A compiler description in Cocol/R is about half the size of the compiler parts generated from it. It provides a better overview of the syntactic and semantic activities in a compiler and is therefore more readable.

Let me conclude with a personal remark: Many compiler generating systems strive for power instead of simplicity and efficiency. I believe that this has done harm to the design of programming languages. The power of compiler generators may well have encouraged the complexity of some of today's languages. Are such powerful tools really necessary? I believe they are not. For me, there seems to be little reason why a programming language should not be designed in a way that makes it amenable to single-pass compilation and top-down parsing.

REFERENCES

[ASU86] A.V.Aho, R.Sethi, J.D.Ullman: Compilers. Addison-Wesley, 1986.

[DoPi90] H.Dobler, K.Pirklbauer: Coco-2 – A New Compiler Compiler. Technical report 90/1, Institut für Informatik, Universität Linz, 1990.

[Knuth68] D.E.Knuth: Semantics of Context-Free Languages. Math.Systems Theory 2, 1968.

[Möss86] H.Mössenböck: Alex – A Simple and Efficient Scanner Generator. SIGPLAN Notices , Vol.21 (5), May 1986.

[Möss87] H.Mössenböck: Compilererzeugende Systeme für Mikrocomputer. Dissertation, Institut für Informatik, Universität Linz, 1987.

[Möss90] H.Mössenböck: Coco/R – A Generator for Fast Compiler Front-Ends. Report 127, Institut für Computersysteme, ETH Zürich, 1990.

[ReMö89] P.Rechenberg, H.Mössenböck: A Compiler-Generator for Microcomputers. Prentice-Hall, 1989.

[Waite86] W.M.Waite: The Cost of Lexical Analysis. SOFTWARE Practice & Experience, Vol.16 (5), May 1986.

[Wirth76] N.Wirth: Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

[Wirth86] N.Wirth: Compilerbau. Teubner Studienbücher, 1986.

[Wirth88] N.Wirth: The Programming Language Oberon. SOFTWARE Practice & Experience, Vol.18 (7), July 1988.