Coco/R – A Generator for Fast Compiler Front-Ends

H. Mössenböck, ETH Zürich

Technical Report

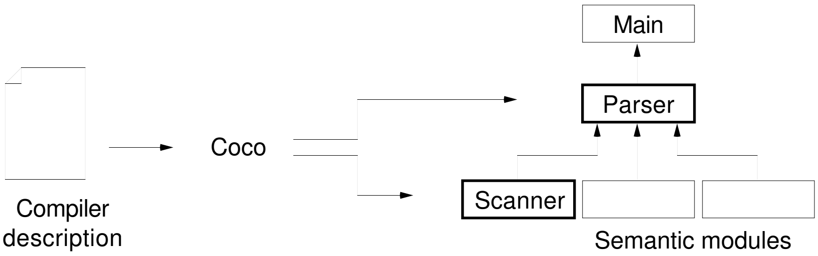
Abstract

Formal compiler descriptions serve two purposes: (1) they can be used as a reference document which specifies the syntax and the semantics of a language, and (2) they provide a convenient notation from which efficient compilers can be generated. Compiler generating systems put emphasis on either the one or the other of these aspects. The system described in this report mainly concentrates on the second goal. We show that it is possible to generate compilers that are as efficient as hand-coded and carefully optimized production-quality compilers. Our system generates recursive descent parsers with a simple error-handling mechanism and scanners with a special buffering scheme. 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 in their work, if it uses a cryptic notation or a multitude of options and special cases. The tool should make their work easier without limiting their flexibility. We used our system to generate an Oberon compiler that is even faster than the standard Oberon compiler used at ETH.

Contents

1. Introduction

Coco/R is a program that takes an augmented EBNF grammar of a language and generates a recursive descent parser and a scanner for this language. The programmer has to supply a main module that calls the parser, as well as semantic modules that are called from within the grammar (e.g., a symbol table handler and a code generator).



The input language of Coco/R (Cocol/R) is based on *attributed grammars*. Attributed grammars were introduced by *Knuth* [Knu68] as a formalism to specify the semantics of context-free languages. In their original form they are static descriptions. They describe dependencies between attributes of symbols without giving an order in which the dependencies are to be evaluated. Many compiler generators stick to this notation [GaGi84, KHZ82, Räi83]. For the implementation of efficient compilers, however, it may be better to look at attributed grammars as an algorithmic notation. The evaluation order of semantic actions is then determined by the textual order of the actions in the grammar. There are also several compiler generators, including Coco/R, that use this paradigm [John75, Gro88].

Coco/R is an improvement over an older version of this program (Coco [ReMö89]). The main difference between Coco and Coco/R is that Coco/R produces recursive descent parsers instead of table-driven parsers and that it integrates the scanner description and the parser description, thus avoiding interface problems between the generated parts. A main nuisance of Coco was that all attributes had to be declared in a global scope, making it necessary to stack attribute values from time to time. This was remedied in Coco/R. Attributes can be declared local to productions. A similar extension of Coco, based on table-driven parsing, has recently been described in [DoPi90].

The following example gives an impression of how a compiler description might look. A precise specification of the description language follows in Section 2. The example shows the translation of variable declarations. The task is to enter declared names into a symbol table and to compute addresses for variables. One starts with a context-free EBNF grammar that is usually already at hand

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

By simply writing down this rule, 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. This requires the following considerations:

- What are the semantic values of *VarDeclaration*, *Ident* and *Type*? In other words, what does the recognition of these symbols yield and what context information must be supplied in order to be able to recognize them? This leads to the so-called *attributes* of the symbols. The attribute of an *Ident* is its name, while the attribute of a *Type* is some node with type information. *VarDeclaration* does not produce an attribute. Instead, it needs an attribute from its context; i.e., it needs to know the next free address in the address space for variables. Attributes can be considered as (input or output) parameters of syntax symbols. They are denoted as follows:

```
Ident <name>
Type <typ>
VarDeclaration <adr>
```

- The next question is: what actions are necessary to translate a construct? These actions are formulated in a general purpose programming language (e.g., *Oberon* [Wirth89]) and are enclosed by the symbols "(." and ".)". A semantic action may appear anywhere on the right-hand side of a production and is executed at that point during parsing.

These consideration lead to an attributed production:

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

Although the format is free, it is wise to shift syntactic parts to the left and semantic parts to the right. This gives a nice separation between syntax and semantics and makes it immediately clear what actions are executed upon recognition of a certain syntax symbol. Note that the production also contains local declarations of variables needed in the semantic actions. Besides, globally declared or imported names can also be accessed.

An attributed grammar can be viewed as a special purpose language for writing compilers (or similar programs). It is a short-hand notation for the well-known recursive descent technique. Although it is not too hard to implement a compiler front-end by hand, a notation like the above can have advantages:

- It is easy to read. Syntax and semantics are clearly separated. Semantic actions are not buried between parsing statements.
- Routine activities like getting the next token from the scanner, handling alternatives, options and iterations, or error-handling don't have to be written down explicitly but are derived from the grammar.
- It is faster and safer to implement a compiler in this high-level notation than in a general purpose programming language. During language design several alternatives of a construct can be tried out and their implementations can be prototyped.
- Irregularities in the grammar like circular productions or violations of the LL(1) property can go undetected when the parser is implemented by hand. For a generator it is easy to check for these irregularities.

Compiler generators enable programmers who are not experienced compiler writers to process little languages. Examples for little languages are numerous in programming [Ben88], ranging from command languages to descriptions of data structures on a file.

The rest of this report describes the input language Cocol/R, shows how the generator can be used, and gives an overview of its implementation together with measurements. The appendix contains an example of a compiler description for a small language.

2. The Compiler Description Language Cocol/R

A compiler description can be viewed as a module consisting of imports, declarations and grammar rules that describe the lexical and syntactical structure of a language as well as its translation into a target language. The vocabulary of Cocol/R uses identifiers, strings and numbers in the usual way:

```
ident = letter {letter|digit}.
string = '"' {anyButQuote} '"' | "'" {anyButApostrophe} "'".
number = digit {digit}.
```

Upper case letters are distinct from lower case letters. Strings must not cross line borders. Keywords are

```
ANY CASE CHARACTERS CHR COMMENTS COMPILER CONTEXT END FROM IGNORE
NESTED PRAGMAS PRODUCTIONS SYNC TO TOKENS WEAK
```

The following metacharacters are used to form EBNF expressions:

```
( )       for grouping
{ }       for iterations
[ ]       for options
< >       for attributes
(.  .)    for semantic parts
= . | + - as explained below
```

Comments are enclosed in "(*" and "*)" and may be nested. The semantic parts may contain declarations or statements in a general purpose programming language. The language actually used is implementation dependent. This implementation uses Oberon.

2.1 Overall Structure

A compiler description is made up of the following parts

```
Cocol =   "COMPILER" ident
          arbitraryText
          ScannerSpecification
          ParserSpecification
          "END" ident "." .
```

The name after the keyword COMPILER is the *grammar name* and must match the name after the keyword END. The grammar name also denotes the topmost nonterminal (the start symbol). After the grammar name arbitrary Oberon text may follow that is not checked by Coco/R. It usually contains imports of Oberon modules and declarations of global objects (constants, types, variables, or procedures) that are needed in the semantic actions later on. The remaining parts of the compiler description specify the lexical and syntactical structure of the language to be processed.

2.2 Scanner Specification

A scanner has to read source text, skip meaningless characters, and recognize tokens which have to be passed to the parser. Tokens may be classified as *literals* and *token classes*. Literals (e.g., "END", ":=", etc.) are written as strings and denote themselves. They are introduced right in the productions and do not have to be declared. Token classes (e.g., identifiers or numbers) have a certain structure that must be declared by a regular expression in EBNF. There are usually many different instances of a token class (e.g., many different identifiers) which are all recognized as the same token.

```
ScannerSpecification =
    { "CHARACTERS" {SetDecl}
    | "TOKENS" {TokenDecl}
    | "PRAGMAS" {PragmaDecl}
    | CommentDecl
    | VariousDecl
    }.
```

A scanner specification consists of 5 optional parts that may be written in arbitrary order.

Character sets. This section allows the declaration of names for character sets like letters or digits. These names may be used in the other sections of the scanner specification.

```
SetDecl    = ident "=" Set.
Set        = BasicSet { ("+"|"-") BasicSet}.
BasicSet   = ident | string | "CHR" "(" number ")" | "ANY".
```

SetDecl associates a name with a character set. Basic character sets are denoted as

string	a set consisting of all characters in the string
ident	the previously declared character set with this name
CHR(i)	a character set consisting of a single element with ordinal value i
ANY	the set of all characters

Character sets may be formed from basic sets by the operators

+	set union
-	set difference

Examples

digit = "0123456789".	the set of all digits
hexdigit = digit + "ABCDEF".	the set of all hexadecimal digits
eol = CHR(13).	end-of-line character
noDigit = ANY - digit.	Any character that is not a digit

Tokens. A token is a terminal symbol for the parser but a syntactically structured symbol for the scanner. This structure has to be described by a regular expression in EBNF.

```
TokenDecl    = Symbol ["=" TokenExpr "."].
TokenExpr    = TokenTerm {"|" TokenTerm}.
TokenTerm    = TokenFactor {TokenFactor} ["CONTEXT" "(" TokenExpr ")"].
TokenFactor  = Symbol | "(" TokenExpr ")" |  "[" TokenExpr "]" | "{" TokenExpr "}".
Symbol       = ident | string.
```

Tokens may be declared in any order. A token declaration defines a symbol together with its structure. Usually the symbol on the left-hand side of the declaration is an identifier. It is declared to stand for the structure described on the right-hand side of the declaration. (For special purposes the symbol on the left-hand side may also be a string, in which case no right-hand side may be specified; see Section 4.)

The right-hand side of a token declaration specifies the structure of the token by a regular EBNF expression. This expression may contain literals denoting themselves (e.g., "END") and names of character sets (e.g., letter) denoting an arbitrary character from this set. It must *not* contain names of previously declared tokens. The CONTEXT phrase in a *TokenTerm* means that the term is only recognized when its right-hand context in the input stream is the *TokenExpr* specified in brackets. If the right-hand side of a declaration is missing, no scanner is generated. This gives the programmer the chance to provide a hand-written scanner (see Section 4).

Examples

```
ident    = letter {letter | digit}.
real     = digit {digit} "." {digit} ["E" ["+"|"-"] digit {digit}].
number   = digit {digit} | digit {digit} CONTEXT ("..").
```

The CONTEXT phrase in the above example allows a distinction between reals (e.g., 1.23) and range constructs (e.g., 1..2) that could otherwise not be scanned with a single character lookahead.

Note: The scanner exports two variables, *pos* and *len*, which are the source position and the length of the most recently recognized token. It also exports a procedure *GetName*(↓*pos*, ↓*len*, ↑*sourceText*) which can be used to obtain the source text of the token at position *pos* having the length *len*. See also Section 3.

Pragmas. A pragma is a token that may occur anywhere in the input stream (e.g., end-of-line symbols or compiler options). It would be too cumbersome to handle the many places in which they could occur in the grammar. Therefore a special mechanism is provided to process pragmas without including them in the productions. Pragmas are declared like tokens, but they may have an associated semantic action that is executed whenever they are recognized by the scanner.

```
PragmaDecl  = TokenDecl [SemAction].
SemAction   = "(." arbitraryText ".)".
```

Example

```
option = "$" {letter} .        (. Scanner.GetName(Scanner.pos, Scanner.len, str); i := 1;
                                  WHILE i < Scanner.len DO
                                    IF str[i] = "A" THEN …
                                    ELSIF str[i] = "B" THEN …
                                    END;
                                    INC(i)
                                  END .)
```

Comments. Comments are difficult (nested comments are even impossible) to specify with regular expressions. This makes it necessary to have a special construct to express their structure. Comments are declared by specifying their opening and their closing brackets. It is possible to declare several kinds of comments. Comment brackets must not be longer than 2 characters.

CommentDecl = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr ["NESTED"].

Examples

```
COMMENTS FROM "(*" TO "*)" NESTED
COMMENTS FROM "--" TO eol
```

Various. The following options serve to parameterize the generated scanner.

VariousDecl = "IGNORE" ("CASE" | Set).

IGNORE CASE specifies that lower case letters are treated like upper case letters in names. IGNORE Set specifies the set of meaningless characters that are to be skipped by the scanner (e.g., tabulators and eol). Blank is meaningless by default.

2.3 Parser Specification

The parser specification is the main part of the compiler description. It contains the productions of an attributed grammar specifying the syntax of the language to be recognized as well as its translation. The productions may be given in any order. References to yet undeclared nonterminals are allowed. Any name that is not declared as a terminal token is considered to be a nonterminal. There must be exactly one production for every nonterminal. There must be a production for the start symbol (the grammar name).

ParserSpecification	=	"PRODUCTIONS" {Production}.
Production	=	ident [FormalAttributes] [LocalDecl] "=" Expression "." .
FormalAttributes	=	"<" *arbitraryText* ">".
LocalDecl	=	"(." *arbitraryText* ".)".
Symbol	=	ident | string.

Productions. A production may be considered as a procedure that parses a nonterminal. It has its own scope for attributes and local objects and is made up of a left-hand side and a right-hand side which are separated by an equal sign. The left-hand side specifies the name of the nonterminal together with its formal attributes and local declarations. The right-hand side consists of a context-free EBNF expression that specifies the structure of the nonterminal as well as its translation. The formal attributes are written like formal parameters in Oberon. They are enclosed in angle brackets. In analogy to input parameters and output parameters (variable parameters) we use the terms input attributes and output attributes. The local declarations are arbitrary Oberon declarations enclosed in "(." and ".)". A production constitutes a scope for its formal attributes and its locally declared objects. Terminals and nonterminals, globally declared objects, and imported modules are visible in any production.

Example

```
Expression <VAR x: Item>          (. VAR y: Item; operator: INTEGER; .)
= … .
```

Expressions. An EBNF expression defines the context-free structure of some part of the source language together with attributes and semantic actions that specify the translation of this part into the target language.

```
Expression    = Term {"|" Term}.
Term          = Factor {Factor}.
Factor        = ["WEAK"] Symbol [Attributes]
              | SemAction
              | "ANY"
              | "SYNC"
              | "(" Expression ")"  |  "[" Expression "]"  |  "{" Expression "}".
Attributes    = "<" arbitraryText ">".
SemAction     = "(." arbitraryText ".)".
Symbol        = ident | string.
```

Nonterminals may have attributes. They are written like actual parameters in Oberon and are enclosed in angle brackets. If a nonterminal has formal attributes, every occurrence of this nonterminal must have a list of actual attributes that correspond to the formal attributes according to the parameter compatibility rules of Oberon. The conformance, however, is only checked when the generated parser module is compiled. A semantic action is an arbitrary sequence of Oberon statements enclosed in "(." and ".)".

The symbol ANY denotes any terminal that is not an alternative of this ANY symbol. It can be used to conveniently parse structures that contain arbitrary text. For example, the translation of a Cocol/R attribute list looks as follows:

```
Attributes <VAR pos, len: LONGINT> =
    "<"          (. pos := Scanner.pos + 1 .)
    {ANY}
    ">"          (. len := Scanner.pos - pos .) .
```

In this example the closing angle bracket is an implicit alternative of the ANY symbol in curly brackets. The meaning is that ANY matches any terminal except ">". *Scanner.pos* is the source text position of the most recently recognized terminal. It is exported by the generated scanner (see Section 3).

Error-handling. The programmer has to give some hints in order to allow Coco/R to generate good and efficient error-handling. First, *synchronization points* have to be specified. A synchronization point is a location in the grammar where especially safe terminals are expected that are hardly ever missing or mistyped. When the generated parser reaches such a point, it adjusts the input to the next symbol that is expected at this point. In most languages good candidates for synchronization points are the beginning of a statement (where IF, WHILE, etc. are expected), the beginning of a declaration sequence (where CONST, VAR, etc. are expected) and the beginning of a type (where RECORD, ARRAY, etc. are expected). The end-of-file symbol is always among the synchronization symbols which guarantees that synchronization terminates at least at the end of the source text. A synchronization point is specified by the symbol SYNC.

Error-handling can further be improved by specifying which terminals are "weak" in a certain context. A *weak terminal* is a symbol that is often mistyped or missing, such as the semicolon between statements. A weak terminal is denoted by preceding it with the keyword WEAK. When the generated parser does not find a terminal specified as weak, it adjusts the input to the next symbol that is either a legal successor of the weak symbol or a symbol expected at any synchronization point (symbols expected at synchronization points are considered to be very "strong", so that it makes sense that they never be skipped).

Example

```
StatementSeq  = Statement {WEAK ";" Statement}.
Declaration   = SYNC ("CONST" … | "TYPE" … | "VAR" … | …).
```

LL(1) requirements. Recursive descent parsing requires that the grammar of the parsed language satisfies the LL(1) property. This means that at any point in the grammar the parser must be able to decide on the basis of a single lookahead symbol which of several possible alternatives have to be selected. For example, the following production is not LL(1):

```
Statement  =  ident ":=" Expression
           |  ident ["(" ExpressionList ")"].
```

Both alternatives start with the symbol *ident* and the parser cannot distinguish between them when it comes to a statement and *ident* is the next input symbol. However, the production can easily be transformed into

```
Statement = ident ( ":=" Expression  |  ["(" ExpressionList ")"] ).
```

where all alternatives start with distinct symbols. There are LL(1) conflicts that are not as easy to detect as in the above example. For a programmer, it can be hard to find them if he has no tool to check the grammar. The result would be a parser that in some situations selects a wrong alternative. Coco/R checks if the grammar satisfies the LL(1) property and gives appropriate error messages that show how to correct any violations.

3. Using Coco/R to Generate a Compiler

The attributed grammar is the central document of a compiler implementation with Coco/R. A user has to perform the following tasks in order to write a compiler:

1. Write an attributed grammar;
2. Write semantic modules if necessary (import them in the attributed grammar);
3. Use Coco/R to generate a scanner and a parser from the attributed grammar;
4. Write a main module that calls the parser.

The command

Coco.Compile *name* ["/" {*letter*}]

translates the compiler description in file *name* (with the grammar name *G*, say) into a scanner module *G*S.Mod and a parser module *G*P.Mod. The following options may be specified

S prints the set of start and successor symbols for every nonterminal
X prints a cross-reference list of all terminals and nonterminals

3.1 Scanner Interface

```
DEFINITION GS;  (*generated scanner*)
    IMPORT Files;
    VAR
        src: Files.File;                (*source file; to be opened by the caller*)
        pos: LONGINT;                   (*source file position of current token*)
        line, col, len: INTEGER;        (*line, column, and length of current token*)
        Error: PROCEDURE (n: INTEGER; pos: LONGINT);  (*install error message procedure here*)
    PROCEDURE Reset;
    PROCEDURE Get(VAR sym: INTEGER);
    PROCEDURE GetName(pos: LONGINT; len: INTEGER; VAR name: ARRAY OF CHAR);
END GS.
```

Reset is called by the parser to initialize the scanner. Note that the main module is responsible to open the source file *src* prior to calling the parser. The parser then calls *Get* repeatedly to get the next token from the source text. Information about the most recently recognized token can be found in the variables *pos*, *line*, *col*, and *len*. The procedure *GetName*(↓*pos*,↓ *len*, ↑*name*) can be used to obtain the text of the token at position *pos* with length *len*.

Error messages. For every syntax error the parser calls the procedure variable *Error* with an error number and an error position as parameters. The user can install any procedure that prints a message or that saves the error information for later output. *Error* can also be used to report semantic errors. (Make sure to use semantic error numbers that do not interfere with syntax error numbers; e.g., start semantic error numbers at 200.) The error numbers together with an explanatory text are appended to the generated parser in the following form:

```
| 0: Msg("EOF expected")
| 1: Msg("ident expected")
| 2: Msg("string expected")
| 3: Msg("number expected")
…
```

This text can be copied to a procedure that prints textual error messages.

3.2 Parser Interface

```
DEFINITION GP;  (*generated parser*)
    PROCEDURE Parse;
END GP.
```

The main program simply has to open the source file and call *Parse* in order to start the compilation. An example of a simple main program is:

```
Texts.OpenScanner(s, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(s);
IF s.class = Texts.Name THEN
    GS.src := Files.Old(s.s);
    GS.Error := own error message procedure;
    IF GS.src # NIL THEN GP.Parse END;
END
```

3.3 Grammar Tests

Coco/R performs several tests to check if the grammar is well-formed. If one of the following error messages is produced, no compiler parts are generated.

No production for X
The nonterminal X has been used, but there is no production for it.

X cannot be reached
There is a production for nonterminal X, but X cannot be derived from the start symbol.

X cannot be derived to terminals
For example, if there is a production X = "(" X ")".

X --> Y, Y --> X
X and Y are nonterminals with circular derivations.

Tokens X and Y cannot be distinguished
The terminal symbols X and Y are declared to have the same structure, e.g.,

```
integer = digit {digit}.
real = digit {digit} ["." {digit}].
```

In this example, a digit string can be recognized as an integer or as a real.

The following messages are warnings. They may indicate an error but they may also describe desired effects. The generated compiler parts are valid. If an LL(1) error is reported for a construct X one must be aware that the generated parser will choose the first of several possible alternatives for X.

X deletable

X can be derived to the empty string, e.g., X = {Y}.

LL(1) error in X: Y is start of more than one alternative

Several alternatives in the production of X start with the terminal Y, e.g.,

```
Statement = ident ":=" Expression | ident [ActualParameters].
```

LL(1) error in X: Y is start and successor of deletable structure

Deletable structures are […] and {…}, e.g.,

```
qualident = [ident "."] ident.
Statement = "IF" Expression "THEN" Statement ["ELSE" Statement].
```

The ELSE at the start of the else-part may also be a successor of a statement. This LL(1) conflict is known under the name "dangling else".

4. Hints for Advanced Users of Coco/R

Providing a Hand-Written Scanner

Scanning is a time-consuming task. The scanner generated by Coco/R is optimized, but it is implemented as a deterministic finite automaton, which introduces some overhead. A manual implementation of the scanner is slightly more efficient. For time-critical applications a programmer may want to generate a parser but provide a hand-written scanner. This can be done by declaring all terminal symbols (including literals) as tokens but without defining their structure by an EBNF expression, e.g.,

```
TOKENS
    ident
    number
    "IF"
    …
```

If a named token is declared without structure, no scanner is generated. Tokens are assigned numbers in the order of their declaration; i.e., the first token gets the number 1, the second the number 2, etc. The number 0 is reserved for the end-of-file symbol. The hand-written scanner has to return token numbers according to this convention. It must have the interface described in Section 3.

Tailoring the Generated Compiler Parts to One's Needs

Using a generator usually increases productivity while at the same time flexibility is decreased. There are always special cases that can be handled more efficiently in a hand-written implementation. A good tool handles routine matters in a standard way but gives the user the chance to change them if he wants to. Coco/R generates the scanner and the parser from source texts (so-called frames) stored under the names *Scanner.FRM* and *Parser.FRM*. It does so by inserting grammar-specific parts into these frames. The programmer may edit the frames and may therefore change any of the internally used algorithms. For example, he can Implement a different buffering scheme for input characters.

Accessing the Lookahead Token

Section 3 specified the interface of the generated scanner. This interface is not complete. Actually, the scanner also exports information about the lookahead token:

```
nextPos: LONGINT;                  (*source file position of the lookahead token*)
nextLine, nextCol: INTEGER;        (*line and column number of the lookahead token*)
```

These variables refer to the most recently *scanned* token (the lookahead token), while the variables *pos*, *line* and *col* refer to the most recently *parsed* token.

Controlling the Parser by Semantic Information

Ideally, syntax analysis should be independent of semantic analysis (symbol table handling, type checking, etc.). Some languages like Ada and C, however, have constructs that can only be distinguished if one also considers semantic information, e.g., the type of the parsed symbols. Even Oberon has constructs that cannot be parsed by looking at their syntax alone. For example, a designator is defined in Oberon as

```
Designator = Qualident {"." ident | "^" | "[" ExprList "]" | "(" Qualident ")" }.
```

where x(T) means a type guard (i.e., x is asserted to be of type T). A designator may be used in a statement

```
Statement = … | Designator ["(" ExprList ")"] | …  .
```

Here x(T) can be interpreted as a designator x (a procedure name) and a parameter T. The two interpretations of x(T) can only be distinguished by looking at the type of x. If it is a procedure then the opening bracket is the start of a parameter list, otherwise the bracket belongs to a type guard.

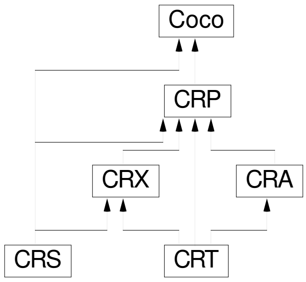
Cocol/R allows control of the parser from within semantic actions to a certain degree. A designator, for example, can be processed in the following way:

```
Designator <VAR x: Item> =
    Qualident <x>
    { …
    |                           (.  IF x is procedure THEN RETURN END .)
        "(" Qualident <y> ")"   (.  process type guard .)
    } .
```

When an opening bracket is seen after a *Qualident*, the alternative starting with an opening bracket is selected. The first semantic action of this alternative checks for the type of x. If x is a procedure, the parser returns from the production and continues in the Statement production.

5. Implementation

Coco/R was written in Oberon on a Ceres workstation. There is also a Modula-2 version for Macintosh computers. A preliminary version of Coco/R was implemented as a master's thesis ([Senn89]). Coco/R was used for its own implementation. Like any compiler, it consists of a scanner (CRS), a parser (CRP), a table handler (CRT), and two modules for output generation, one for the scanner (CRA) and one for the parser (CRX). During parsing the scanner description is transformed into a non-deterministic finite automaton and the attributed grammar is transformed into a graph. These data structures are used for grammar tests and for the generation of the two source modules.



5.1 Scanner Generation

During parsing of the compiler description, the token declarations are translated into a syntax graph from which a non-deterministic finite automaton is generated. In a second step, this automaton is made deterministic. The algorithms for the manipulation of the automaton are described in [Möss86]. Figure 1 sketches this process.

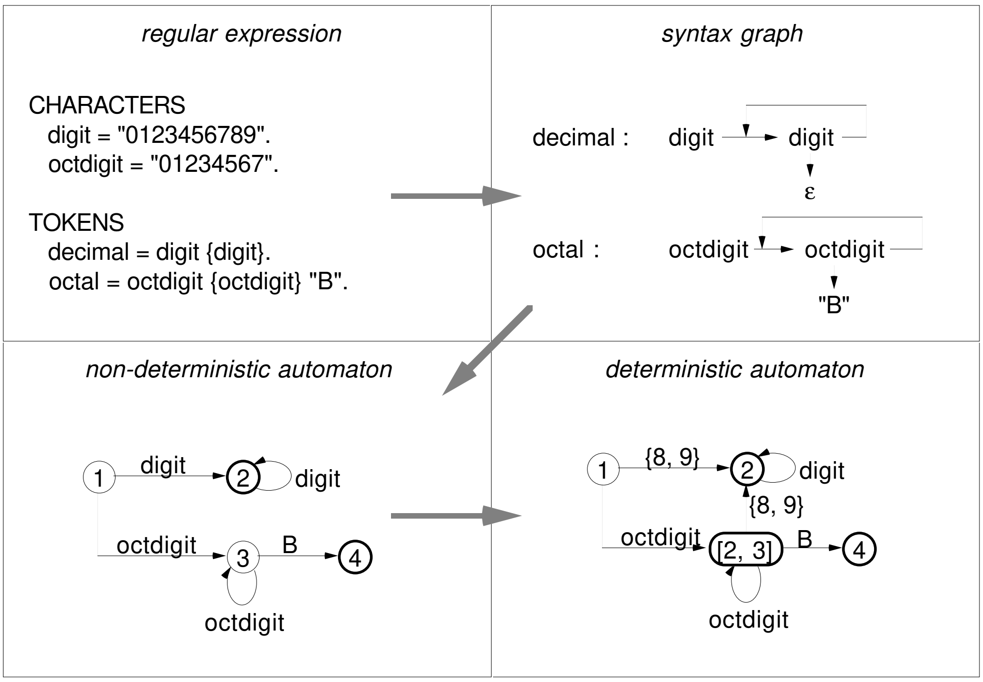


Fig. 1 Transformation of regular expressions into a deterministic finite automaton

The automaton is not generated directly from the regular expressions but from a syntax graph. This allows making it more deterministic from the beginning, thus simplifying the later algorithms. Figure 1 shows that tokens may have very similar structures, differing only in their last characters. These structures are automatically stripped of any disambiguity. The programmer does not have to take care of making the beginnings of tokens distinct. Even very complicated structures can be processed like the various kinds of numeric constants in Modula-2 (Figure 2; final states are denoted by bold circles):

```
integer =
  digit {digit}
| digit {digit} CONTEXT ("..")
| digit {hexdigit} "H"
| octdigit {octdigit} "B".

char =
  octdigit {octdigit} "C".

real =
  digit {digit} "." {digit}
  ["E" ["+"|"-"] digit [digit]].
```

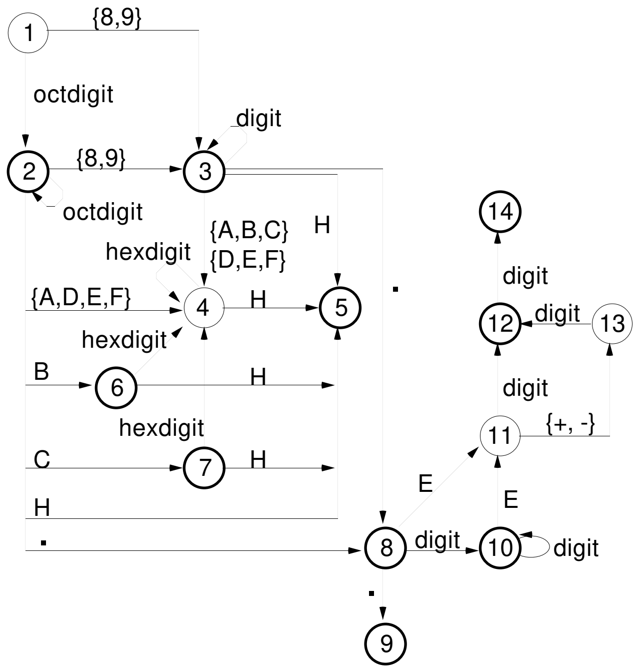
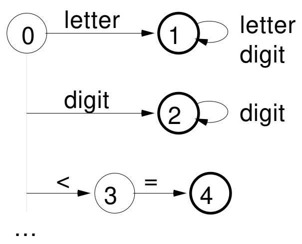


Fig.2 Automaton for the various kinds of numeric constants in Modula-2

If speed is important, a finite automaton is not the best possible implementation of a scanner. It would be more efficient to implement the recognition of tokens as ordinary procedures like in a hand-written scanner. However, if one looks at the Figure 2, one can imagine that it is not easy to generate such a scanner automatically. On the other hand, an automaton is space-efficient. Therefore we decided to generate the scanner as an automaton. To make it as efficient as possible, the automaton is not table-driven but implemented in code. For the token declarations

```
ident = letter {letter|digit}.
number = digit {digit}.
```

and the occurrence of the literals "IF", "THEN", "END", "<", and "<=" in the productions of the grammar the automaton of Figure 3 is generated. The textual representation of token classes like *ident* or *number* can be obtained via the procedure *GetName*(↓*pos*, ↓*len*, ↑*sourceText*) where *pos* and *len* are the position and length of the token in the source file. Note, that keywords cannot be handled by the automaton since they have the same structure as identifiers. This fact is taken into account by Coco/R: Any literal that matches a declared token is stored in a literal list. Later, when an identifier is recognized, it is checked whether it is a literal (*CheckLiteral*), and if so, the literal's token number is returned. Comments are not handled by the automaton. They are recognized by a special procedure.




```
Get(VAR sym: INTEGER)
  …
  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; 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;
```

Fig. 3 Implementation of an automaton

The most time-consuming task in scanning is reading the source text. The scanner can be speeded up significantly if reading can be made faster. To read a text character by character is usually slower than to read 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 more than three times faster than reading it character by character. Even large Oberon programs rarely exceed 40 kilobytes in source code. With several megabytes of memory available, this "waste" of 40 kilobytes seems justified if scanning speed can be improved so drastically (the overall run time of the compiler is improved by 30%).

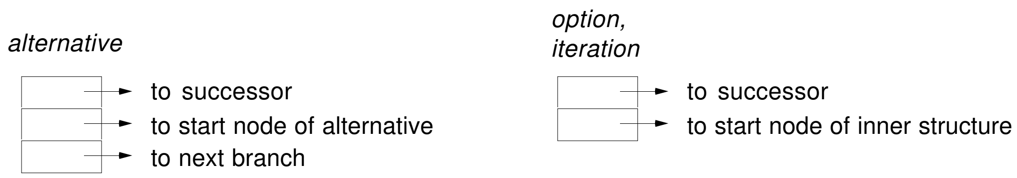
Having the whole source text in memory has yet another advantage: the source text can be used as a name list. The text of token classes, like identifiers, no longer has to be copied to a separate name list but can remain where it is. One simply has to remember its position and its length. This idea is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

Another advantage of this technique is that it permits the backup of the input pointer to any previous position. This is useful for handling tokens with CONTEXT phrases in Cocol/R. 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 CONTEXT phrase, so that this text will be read by the scanner again.

5.2 Parser Generation

The productions of the attributed grammar are translated into procedures of a recursive descent parser. However, it is not possible to generate the parser on the fly while the grammar is analyzed since certain sets of terminal symbols are required at various locations in the parsing procedures. These sets can only be computed when the whole grammar is known. Therefore, the productions are first translated into syntax graphs, then the symbol sets are computed, and finally the parsing procedures are generated from the graphs. The syntax graphs are also used for grammar tests (completeness, redundancy, LL(1) property).

Syntax graphs. A node is generated for every symbol in the grammar and for every semantic action. A sequence of symbols and actions is translated into a sequence of nodes. A semantic action node contains the position and the length of the action in the source text. Alternatives, options and iterations are modelled by special nodes of the following form



A production like

```
Expression <VAR x: OGT.Item>    (. VAR y: OGT.Item; op: INTEGER .)
= SimExpr <x>
  [  Relop <op>                 (. IF x.typ.form = Bool THEN OGE.MOp(op, x) END .)
     SimExpr <y>                (. OGE.Op(op, x, y) .)
  |  "IN" SimExpr <y>           (. OGE.In(x, y) .)
  |  "IS"                       (. IF x.mode >= Typ THEN err(112) END .)
     qualident <y>              (. IF y.mode = Typ THEN OGE.TypTest(x, y) ELSE err(52) END .)
  ].
```

is translated into the following graph

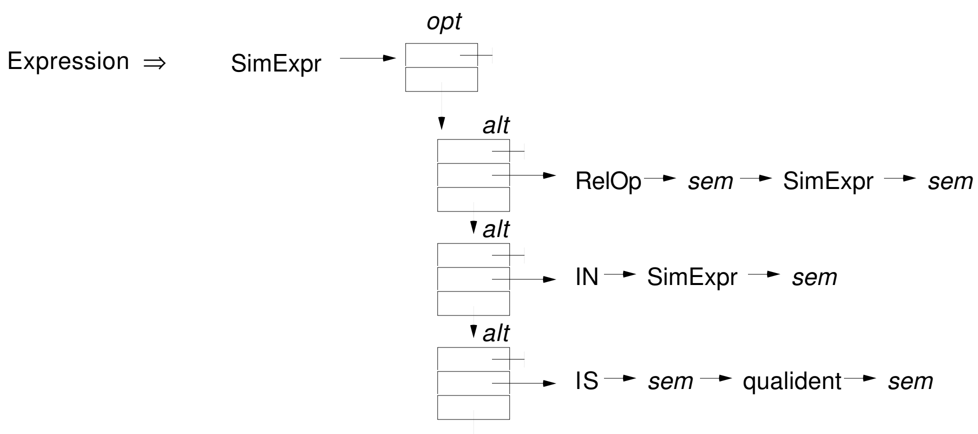


Fig. 4 Syntax graph for the generation of parsing procedures

Note that this kind of graphs is different from the syntax 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 symbol sets, it is easy to generate parsing procedures. A sequence of nodes is translated into a sequence of parsing constructs. Semantic actions are simply copied from the source text without modification. The following table shows that every grammar item can be replaced mechanically by the equivalent parsing item.

	grammar item	*parsing item*
terminal	t	Expect(t)
nonterminal	nt <a, b>	nt(a, b)
semantic action	(. *anyText* .)	*anyText*

Terminals are recognized by the procedure

```
PROCEDURE Expect(s: INTEGER);
BEGIN IF sym = s THEN Get ELSE Error(s) END
END Expect;
```

The procedure *Get* requests the next input token from the scanner and stores it in the global variable *sym*. It is also responsible for filtering out pragmas.

```
PROCEDURE Get;
BEGIN
    LOOP Scanner.Get(sym);
        IF sym is pragma THEN Handle it ELSE EXIT END
    END
END Get;
```

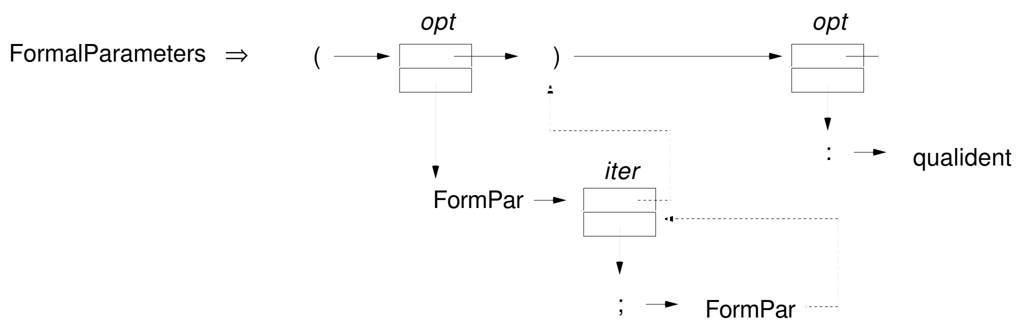
Alternatives, options and iterations are translated into control structures. Whenever possible, redundant checks are eliminated. The following procedure is generated from the graph in Figure 4.

```
PROCEDURE Expression (VAR x: OGT.Item);
    VAR y: OGT.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); IF x.typ.form = Bool THEN OGE.MOp(op, x) END;
            SimExpr(y); OGE.Op(op, x, y);
        ELSIF sym = in THEN
            Get; SimExpr(y); OGE.In(x, y);
        ELSE
            Get; IF x.mode >= Typ THEN err(112) END;
            qualident(y); IF y.mode = Typ THEN OGE.TypTest(x, y) ELSE err(52) 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:

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

5.3 Error Recovery

Good and efficient error recovery is difficult in recursive descent parsers since little information about the parsing process is available when an error occurs. What has to be done in case of an error:

1. Find all symbols with which parsing can be resumed at a certain location in the grammar reachable from the error location (recovery symbols).
2. Skip the input up to the first symbol that is in the recovery set.
3. Drive the parser to the location where the recovery symbol can be recognized.
4. Resume parsing from there.

In recursive descent parsers, information about the parsing location and about the expected symbols is only implicitly contained in the parser code (and in the procedure call stack) and cannot be exploited for error recovery. One method to overcome this is to compute the recovery set dynamically during parsing. Then, when an error occurs, the recovery symbols are already known and all that one has to do is to skip erroneous input and to "unroll" the procedure stack up to a legal continuation point [Wirth76]. 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. Errors at other points are reported but cause no recovery. Parsing simply continues up to the next synchronization point where the grammar and the input are synchronized again. This requires the designer of the grammar to specify synchronization points explicitly – not a very difficult task if one thinks for a moment. The advantage is that no recovery sets have to be computed at run time. This makes the parser small and fast.

Synchronization points. In Cocol/R a synchronization point is specified by the keyword SYNC (see Section 2.3). A good synchronization point is a location in the grammar where particularly safe symbols (like keywords) are expected and that is often visited by the parser. Typical candidates are the beginning of a statement, the beginning of a declaration or the beginning of a structured type. A synchronization point is translated into a loop that skips all symbols not expected at this point (except end-of-file). The set of these symbols can be precomputed at parser generation time. The following example shows two synchronization points and their counterparts in the generated parser.

```
production                      generated parsing procedure

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

To avoid spurious error messages, an error is only reported when a certain amount of text 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 (like semicolon). These symbols are marked in the grammar by the keyword WEAK (see Section 2.3). If the parser tries to recognize a weak symbol and finds it missing, it reports an error and skips the input until a legal successor of the 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.

```
                                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
   ELSE
      Error(s); WHILE sym ∉ expected ∪ {symbols expected at synchronization points} DO Get END
   END
END Weak;
```

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

When an iteration starts with a weak symbol, this symbol is called a *weak separator* and is handled in a special way. If it cannot be recognized, the input is skipped until a symbol that is contained in one of the following three sets is found:

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

```
                                   generated parsing procedure
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 can be precomputed at parser generation time. *WeakSep* is implemented as follows:

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

The observant reader may have noticed that the set β contains the successors of a statement sequence in any possible context. This set may be too large. If the statement sequence occurs within a repeat statement, only UNTIL is a legal successor, but not END or ELSE. We accept 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 our method yields good error recovery. We 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 error messages are kept short and simple. This is due to our conviction that experienced programmers do not need a detailed explanation of what actions the parser performed in order to recover from the error. In almost all cases it is sufficient to point to the error location and to give a rough hint.

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

We compared an Oberon compiler generated by Coco/R with a manually implemented Oberon compiler. The back-end modules of both compilers are the same. Only the scanner and the parser are different. We measured the time to compile a 867-line Oberon program (7169 tokens or 24254 characters) on a Ceres-2 workstation with a NS32532 processor running at 25 MHz.

Original compiler	3.9 sec
Generated compiler	3.0 sec

The generated compiler is 23 % faster than the original 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 original one. Scanning and parsing contribute to the overall run time of the generated compiler in the following way:

Scanning	0.61 sec	20 %
Parsing	0.12 sec	4 %

This gives a compilation speed of

Scanning	11 625	tokens/sec
Parsing	51 127	tokens/sec
Total compilation speed	2 395	tokens/sec

Comparing the object code of the two compilers yields the following measures:

	Original compiler	Generated compiler	
Scanner	3 672 Bytes	3 944 Bytes	+ 7%
Parser (incl. semantic actions)	11 740 Bytes	12 236 Bytes	+ 4%

7. Summary

Attributed grammars, when regarded as an algorithmic notation, are a special purpose language to describe translation processes. They serve three purposes:

- *Specification*. A translator can be specified and designed this way before it is implemented in a conventional programming language.
- *Documentation.* An attributed grammar is a concise documentation of a translation process. It contains the same information as the program that implements the translator but in more compact form.
- *Implementation.* If a tool like Coco/R is available, an attributed grammar is already the implementation of the translator.

Attributed grammars can be used to specify all kinds of programs that process a single stream of structured input data. They can be applied not only to proper compilers but also to compiler-like programs like cross-reference generators, pretty printers or complexity analyzers and even to tasks that do not fall in the traditional scope of compiler construction, like the processing of data files that describe pictures, formatted text or database information.

Among the advantages of using a tool like Coco/R are a fast and safe implementation of translators, high flexibility in experimenting with a language design, and a translator description that is more concise and more readable than an implementation in a conventional programming language. The effort to learn the description language is small, since semantic parts are written in a familiar programming language and syntactic parts are based on the well-known formalism of EBNF grammars. The translators generated by Coco/R are fast enough to compete with production-quality compilers.

Appendix A Cocol/R Grammar

```
Cocol          = "COMPILER" ident
                 { ANY }
                 { Declaration }
                 "PRODUCTIONS"
                 { ident [Attributes] [SemText] "=" Expression "."}
                 "END" ident ".".
Declaration    =  "CHARACTERS" { SetDecl }
               |  "TOKENS"     { TokenDecl }
               |  "PRAGMAS"    { PragmaDecl }
               |  "COMMENTS" "FROM" TokenExpr "TO" TokenExpr ["NESTED"]
               |  "IGNORE" ( "CASE" | Set ).

SetDecl        = ident "=" Set ".".
Set            = SimSet { "+" SimSet | "-" SimSet }.
SimSet         = ident | string | "CHR" "(" number ")" | "ANY".

TokenDecl      = Symbol ["=" TokenExpr "."].
TokenExpr      = TokenTerm { "|" TokenTerm }.
TokenTerm      = TokenFactor { TokenFactor} [ "CONTEXT" "(" TokenExpr ")" ].
TokenFactor    = ( Symbol |  "(" TokenExpr ")" |  "[" TokenExpr "]" |  "{" TokenExpr "}" ).

PragmaDecl     = TokenDecl [SemText].

Expression     = Term { "|" Term }.
Term           = Factor { Factor}.
Factor         = (  ["WEAK"] Symbol [Attributes]
               |  SemText
               |  "ANY"
               |  "SYNC"
               |  "(" Expression ")"  |  "[" Expression "]"  |  "{" Expression "}"
               ).
Symbol         =  ident | string.
Attributes     = "<" { ANY } ">".
SemText        = "(." { ANY } ".)".
```

Appendix B Sample Attributed Grammar in Cocol/R

The following attributed grammar describes a compiler for a simple programming language. It uses a symbol table handler (TL) and a code generator (TC) that generates code for a stack machine. These two modules are not described further. The purpose of this grammar is to give a coherent example of an attributed grammar. It is not necessary for the reader to understand the translation process in all details, although the semantic actions in this grammar are rather similar to actions contained in any compiler.

```
COMPILER Taste

(*------------------------------------- imports and global declarations ----------------------------------------*)
IMPORT TL (*table handler*), TC (*code generator*);

CONST
   plus = 0; minus = 1; times = 2; slash = 3; equ = 4; lss = 5; gtr = 6; (*operators*)
   undef = 0; int = 1; bool = 2;  (*types*)
   vars = 0; procs = 1;  (*object kinds*)
   ADD = 0; SUB = 1; MUL = 2; DIVI = 3; EQU = 4; LSS = 5; GTR = 6; (*machine instructions*)
   LOAD = 7; LIT = 8; STO = 9; CALL = 10; RET = 11; RES = 12;
   JMP = 13; FJMP = 14; HALTc = 15; NEG = 16; READ = 17; WRITE = 18;

TYPE
   Name = ARRAY 32 OF CHAR;

PROCEDURE Err(nr: INTEGER);
BEGIN TasteS.Error(100 + nr, TasteS.pos) END Err;

PROCEDURE StringToVal(s: ARRAY OF CHAR; VAR val: INTEGER);
   VAR i: INTEGER;
BEGIN
   val:=0; i := 0;
   WHILE s[i] # 0X DO val := 10 * val + ORD(s[i]) - ORD("0"); INC(i) END
END StringToVal;


(*----------------------------------------- scanner specification ------------------------------------------------*)
CHARACTERS
   letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
   digit = "0123456789".
   eol = CHR(13).
    tab = CHR(9).

TOKENS
   ident = letter {letter | digit}.
   number  = digit {digit}.

IGNORE eol + tab

COMMENTS FROM "(*" TO "*)" NESTED
```

```
(*------------------------------------------------------ parser specification -----------------------------------------------------------*)
PRODUCTIONS

Taste                          (.  VAR name, progName: Name; obj: TL.Object; .)
=  "PROGRAM"
   Ident<progName> ";"         (.  TC.progStart := TC.pc .)
   Body
   Ident<name> "."             (.  IF name # progName THEN Err(0) END; TC.Emit(HALTc) .).
(*--------------------------------------------------------------------------------------------------------------------------------------*)
Body                           (. VAR name, name1; Name; fix, type: INTEGER; obj: TL.Object; .)
=                              (. TL.EnterScope;   fix := TC.pc + 1; TC.Emit2(JMP,0) .)
   {   "VAR"
       { Ident<name> ":"       (. obj := TL.NewObj(name, vars) .)
         TypeId<obj^.type> ";"
       }

   |   "PROCEDURE"
       Ident<name> ";"         (. obj := TL.NewObj(name, procs);  obj^.adr := TC.pc .)
       Body
       Ident<name1> ";"        (. TC.Emit(RET);  IF name # name1 THEN Err(0) END .)
   }
   "BEGIN"                     (. TC.Fixup(fix); TC.Emit2(RES, TL.DataSpace()) .)
   StatSeq
   "END"                       (. TL.LeaveScope .).
(*--------------------------------------------------------------------------------------------------------------------------------------*)
TypeId<VAR type: INTEGER>
=  "INTEGER"                   (.  type := int .)
|  "BOOLEAN"                   (.  type := bool .).
(*--------------------------------------------------------------------------------------------------------------------------------------*)
StatSeq = Stat {";" Stat}.
(*--------------------------------------------------------------------------------------------------------------------------------------*)
Stat                           (.  VAR name: Name; type, fix, fix2, loopstart: INTEGER; obj: TL.Object;
=  [    Ident<name>            (.  obj := TL.Obj(name) .)
       (   ":" "="             (.  IF obj^.kind # vars THEN Err(4) END .)
           Expression<type>    (.  IF type # obj^.type THEN Err(2) END;
                                   TC.Emit3(STO, TL.curLevel - obj^.level, obj^.adr) .)
       |                       (.  IF obj^.kind # procs THEN Err(5) END;
                                   TC.Emit3(CALL, TL.curLevel - obj^.level, obj^.adr) .)
       )
   |   "IF" Expression<type>   (.  IF type # bool THEN Err(3) END; fix := TC.pc + 1; TC.Emit2(FJMP, 0) .)
       "THEN" StatSeq
       [   "ELSE"              (.  fix2 := TC.pc + 1; TC.Emit2(JMP, 0); TC.Fixup(fix); fix := fix2 .)
           StatSeq
       ]
       "END"                   (.  TC.Fixup(fix) .)
   |   "WHILE"                 (.  loopstart := TC.pc .)
       Expression<type>        (.  IF type # bool THEN Err(3) END;  fix := TC.pc + 1; TC.Emit2(FJMP, 0) .)
       "DO" StatSeq "END"      (.  TC.Emit2(JMP, loopstart); TC.Fixup(fix) .)
   |   "READ" Ident<name>      (.  obj := TL.Obj(name); IF obj^.type # int THEN Err(1) END;
                                   TC.Emit3(READ, TL.curLevel - obj^.level, obj^.adr) .)
   |   "WRITE" Expression<type>   (.  IF type # int THEN Err(1) END; TC.Emit(WRITE) .)
   ].
(*--------------------------------------------------------------------------------------------------------------------------------------*)
Expression<VAR type: INTEGER>  (. VAR type1, op: INTEGER; .)
=  SimExpr<type>
   [ RelOp<op> SimExpr<type1>  (. IF type # type1 THEN Err(2) END; TC.Emit(op); type := bool .)
   ].
(*--------------------------------------------------------------------------------------------------------------------------------------*)
```

```
RelOp<VAR op: INTEGER>
=  "="                          (.  op := equ .)
|  "<"                          (.  op := lss .)
|  ">"                          (.  op := gtr .).
(*---------------------------------------------------------------------------------------------------------*)
SimExpr<VAR type: INTEGER>      (. VAR type1, op: INTEGER; .)
=  Term<type>
   { AddOp<op> Term<type1>      (. IF (type # int) OR (type1 # int) THEN Err(1) END; TC.Emit(op) .)
   }.
(*---------------------------------------------------------------------------------------------------------*)
AddOp<VAR op: INTEGER>
=  "+"                          (. op:=plus .)
 | "-"                          (. op:=minus .).
(*---------------------------------------------------------------------------------------------------------*)
Term<VAR type: INTEGER>         (.  VAR type1, op: INTEGER; .)
=  Factor<type>
   {   MulOp<op> Factor<type1>  (. IF (type # int) OR (type1 # int) THEN Err(1) END; TC.Emit(op) .)
   }.
(*---------------------------------------------------------------------------------------------------------*)
MulOp<VAR op: INTEGER>
=  "*"                          (. op := times .)
|  "/"                          (. op := slash .).
(*---------------------------------------------------------------------------------------------------------*)
Factor<VAR type: INTEGER>        (.  VAR name: Name; val, n: INTEGER; obj: TL.Object; .)
=  (    Ident<name>              (.  obj := TL.Obj(name); type := obj^.type;
                                     IF obj^.kind = vars THEN
                                        TC.Emit3(LOAD, TL.curLevel - obj^.level, obj^.adr)
                                     ELSE Err(4)
                                     END .)
   |   "TRUE"                   (.  TC.Emit2(LIT, 1); type := bool .)
   |   "FALSE"                  (.  TC.Emit2(LIT, 0); type := bool .)
   |   number                   (.  TasteS.GetName(TasteS.pos, TasteS.len, name);
                                    StringToVal(name, n);  TC.Emit2(LIT, n); type:=int .)
   |   "-" Factor<type>         (.  IF type # int THEN Err(1); type := int END; TC.Emit(NEG) .)
   ).
(*---------------------------------------------------------------------------------------------------------*)
Ident <VAR name: Name> =
   ident                        (.  TasteS.GetName(TasteS.pos, TasteS.len, name) .).

END Taste.
```

References

[Ben88] J.Bentley: More Programming Pearls. Addison-Wesley 1988

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

[GaGi84] H.Ganzinger, R.Giegerich: Attribute Coupled Grammars. SIGPLAN Notices 19 (1984), 6, 157-170

[Gro88] J.Grosch: Generators for High-Speed Front-Ends. Lecture Notes in Computer Science 371, Springer Verlag, 1988

[KHZ82] U.Kastens, B.Hutt, E.Zimmermann: GAG: A Practical Compiler Generator. Lecture Notes in Computer Science 141, Springer Verlag, 1982

[John75] S.C.Johnson: YACC – Yet another Compiler-Compiler. Tech.Report No 32, Bell Laboratories, July 1975

[Knu68] D.E.Knuth: Semantics of Context-Free Languages. Mathematical Systems Theory 2 (1968), 127-145

[Möss86] H.Mössenböck: Compilererzeugende Systeme für Mikrocomputer. Ph.D. thesis, Universität Linz, 1986

[Räi83] K.-J. Räihä, et al.: Revised Report on the Compiler Writing System HLP78. Report A-1983-1, Department of Computer Science, University of Helsinki

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

[Senn89] R.Sennhauser: Übersetzung attributierter Grammatiken. Diploma thesis, ETH Zurich, 1989

[Waite86] W.M.Waite: The Cost of Lexical Analysis. Software – Practice and Experience 16 (1986), 5, 473-488

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

[Wirth86] N.Wirth: Compilerbau. 4th edition. Teubner Studienbücher, 1986

[Wirth89] N.Wirth: The Programming Language Oberon. Report 111, ETH Zurich, September 1989