数理解析研究所講究録
363巻 1979年 59-99

DESIGN AND IMPLEMENTATION OF A MULTIPASS-COMPILER GENERATOR

Masataka Sassa, Junko Tokuda,
Tsuyoshi Shinogi, and Kenzo Inoue

(Department of Information Science
Tokyo Institute of Technology)

Abstract

A compiler generator (compiler-compiler) is presented for automatically generating compilers allowing multipass scan and optimization.

Description to the compiler generator is designed so that it is a complete and readable description of compiler and user program's run-time environments. The description includes those for lexical analyzer using regular expressions, syntax and semantics of each pass using an attribute-grammar-style description, and run-time prelude.

For incrementally generating a compiler, the compiler generator can partially (re)generate a compiler, i.e., create or update a compiler parts by parts, with respect to user specified parts among the whole description.

Keywords

compiler generator, compiler-compiler, multipass compiler, compiler description, attribute grammar

---

This report is a revised version of Research Reports on Information Science, No. C-24, Dept. of Information Science, Tokyo Institute of Technology.

1. Introduction

In many recent programming languages, definitions and uses of language constructs may appear in any order. This generosity causes some difficulties in compiling process.

One most commonly arising case is labels and "go to" statements using the labels. In a situation

```
begin
     . . . L: . . .
     begin
          . . .
          go to L ;                                   (1)
          . . . L: . . .
     end
     . . .
end
```

what "L" at (1) denotes cannot be determined at this point.

Another commonly arising difficulty in block structured languages is the identification of variables in procedure bodies, as follows (cf. [Yon76]).

```
begin
     procedure f ; ... x + g ... ;                    (2)
     integer procedure g ; g := . . . ;               (3)
     integer x ;                                      (4)
     . . .
end
```

The exact specification of "x" and "g" at (2) cannot be found until lines (3) and (4) are scanned.

Similar difficulties arise in processing mutually recursive procedures, or in processing declarations and uses of operator priority and mode in Algol 68 [Wij76].

As the above mentioned inverted sequences of definitions and uses cannot be analyzed efficiently in a single-pass compiler, we have proposed in a previous note [Ino77] a compiler generator which generates compilers allowing multipass scan and optimization.

Although most of existing compiler generators are just able

to generate single-pass compilers, some attempts were made at automatic generation of multipass compiler [Gan77]. The compiler generated by the above system creates at the first pass a (somewhat optimized) parse tree, and at later passes it gives and evaluates semantic attributes and makes optimizing transformation on the tree. However, it seems to proceed slowly, due to the creation of the tree. Further, note that this tree construction should be completed in the first pass, making the system essentially single-pass. This would cause difficulties in Algol 68 compilers, as mentioned above. For example, the precise analysis of "expression" cannot be made at the first pass in case operator precedence is defined later in the source text.

In contrast to the above system, we have adopted a real multipass parsing without creating the parse tree. This note discusses the design and implementation of our multipass-compiler generator, which may be simply referred as "generator" in this note, with its underlining design philosophy and its evaluation.

Description to the compiler generator is designed so that it is a complete and readable description of compiler and user program's run-time environments. The description includes those for lexical analyzer using regular expressions, syntax and semantics of each pass using an attribute-grammar-style description, and run-time prelude.

For incrementally generating a compiler, the compiler generator can partially (re)generate a compiler, i.e., create or update a compiler parts by parts with respect to user specified parts among the whole description.

2. Design philosophy

In designing our compiler generator we have stressed on the following criteria:

(1) a complete, readable and easily modifiable compiler description language,

(2) an efficiently usable generator structure which can flexibly regenerate a compiler whenever part of the description is modified, and

(3) machine independence considerations.

These criteria are briefly discussed in the following.

2.1. Complete and readable compiler description language

Compiler generators are not only designed for saving human labor, but are also a tool for formalizing compilers, which are rather complex objects. Our point of view on the input description for the generator is that it should be ideally a complete documentation of "compiled language and its environments". However, since we are to make actual compilers, we search for a point of compromise with making an effort for formalizing the compiler description. The description includes user programs and data, the run-time prelude which may be considered as the middle level "environment", in addition to the usual compiler description which may be considered as the outmost "environment". In other words, our generator system unifies compiler generator and compiler. As the result of formalization, each description unit can be written in an appropriate and readable description language which will be discussed in later sections.

2.2. Generator structure allowing efficient partial regeneration

No compiler description will immediately be error free, nor will it be the final description when it is input to the generator for the first time. The completeness and easiness of modification of the description as discussed in the previous section can only be guaranteed by a system which avoids unnecessary efforts, such as the redoing of the whole generation process, on account of a single bug in the compiler description. We have designed a generator structure where the task of generating a compiler may be incremental by making the task divisible into efficient partial (re) generation steps corresponding to each of the description units.

2.3. Machine independence considerations

In designing the generator, the generated compiler and the object code, we have taken care to make machine-dependent parts be logically well discriminated from other parts. We shall discuss this point later in chapter 9.

3. Overview of the generator system

Our objective is to automatically generate a compiler which analyzes source text by multipass parsing. Schematic view of multipass parsing is shown in Fig. 3.1. The parser of each pass usually reads input text of the pass (blank portion in the figure) and copies into output text of the pass. When the parser catches the starting position of the partial grammar parsing for this pass, it enters the parsing mode, analyzes the input text (shaded portion in the figure), and outputs the goal symbol. Thus

the length of intermediate text decreases as the text goes through passes.

Now, the schematic view of the multipass-compiler generator is shown in Fig. 3.2.

The input description to the generator consists of the following description units, written respectively in appropriate description languages:

(1) Lexical analyzer : regular expressions.

(2) Syntax and semantics for each pass : attribute grammars-style description + procedure-oriented language (Algol 68-style macro-extended Fortran, to be briefly shown in a later section).

(3) Lexical and semantic subordinate routines : procedure-oriented language (as above).

(4) Run-time prelude (standard environment) : the language acceptable by the generated compiler, or others.

(5) User program and data : the language acceptable by the generated compiler.

The generator inputs the above description and creates a multipass compiler. The generator consists of a master control program, several sub-generators, and interface files for passing information among those sub-generators. The presense of interface files affords a key for the efficient partial regeneration of a compiler. This subject will be descussed in detail in chapter 7.

The compiler generated by our system analyzes source texts by multipass parsing, and outputs object codes. The compiler is usually made in a structure where analyzers for each pass are overlayed with one another. Further details will be discussed in a later section.

4. Description of lexical analyzer

There are several tools for automatically generating lexical analyzers, for example LEX [Les75]. We adopt here a similar approach using regular expressions.

Figure 4.1 is an example description of lexical analyzer. It consists of a set of "class"es, the first characters of terminal symbols (tokens), a set of "table"s, the table of keywords etc., and a set of "symbol"s, the specification of terminal symbols corresponding to each "class" using a description in regular expressions together with procedure names and terminal symbol names. Names prefixed by "?" are procedures which are pieces of codes to be executed whenever a terminal symbol specified by the relevant regular expression is recognized. Names prefixed by "!" indicate terminal symbols which are to be used in the description of syntax.

Example of identifier :

In Fig. 4.1, suppose that "I10" is given as input. The lexical analyzer first reads the letter "I", selects the "class" LET, and "jumps" to the "symbol" IDSEQ. Then, it scans input while the input character belongs to LET or DIG or space. (< ... | ... | ... > in the description means (... | ... | ...)* in the usual mathematical notation.) Thus, "1" and "0" are scanned. Scanned characters are usually stored in a string area. Afterwards, the procedure IDENT, provided by compiler-writer, is called perhaps for making an entry "I10" in the symbol table. Finally, the lexical analyzer returns ID as the indication of the token, which can be used as a terminal symbol in the description of syntax. Of course ID is internally represented by an integer number.

Subordinate routines such as IDENT can be written by compiler writer using an Algol 68-style language (Fig. 4.2). They are translated into Fortran using a pattern matching macro processor [Sas79].

The description style for lexical analyzer extends regular expressions in several aspects. They are illustrated by examples using Fig. 4.1.

Example of bold tag symbol:

Let us assume that the hardware representation of "bold tag symbols" in source text is preceded by ".". When the lexical analyzer reads the first character of ".BEGIN", it selects the "class" PERIOD and "jumps" to "symbol" PERSEQ. It skips the first character which belongs to PERIOD (in our description, "-" means skip, i.e., do not store in the string area) and scans input while LET(LET|DIG)* is recognized. Now, the string "BEGIN" is saved in the string area. Then, it searches for "BEGIN" in "table" TERM2, and since it will succeed the indication of token becomes BEGIN. (If it failed, the .OUT part of the description would be selected.)

Example of string:

The description of string in Fig. 4.1 assumes a string denotation form which is enclosed by either single-quotes (') or double-quotes ("). To include a single-quote or a double-quote itself in a string enclosed by the same characters, this denotation form implies a representation by their succession.

Suppose for example that 'DON''T' is given as input. The lexical analyzer selects the "class" QUOTE, "jumps" to "symbol" STRING. (Here, "$" specifies any character if it appears the

first time in a regular expression. Otherwise, it specifies the character which matches the character previously designated as "$".) The lexical analyzer skips the first character (by "-$") which is a single-quote in this case, scans input while the input character is not a single-quote (by "¬$"), or skips the first occurrence of single-quotes if they succeed (by "-$$"), until finally finding a single-quote which is to be skipped (by "-$"). Thus in the above case, DON'T is stored in the string area. Now, the procedure STORE is called, which should return an integer 1 or 2. According to this return value, either CHARCON or STRCON is selected as terminal symbol.

Example of colon-sequence:

The reader will easily find that the "symbol" COLSEQ accepts the following inputs

:=: := :/=: :

("!" not followed by names specifies that the token scanned until that time is used as the terminal symbol. "/" means empty.)

5. Multipass parsing

In order to achieve multipass parsing, only partial portions of the input text of each pass are analyzed by the parser, and the information resulting from semantic actions is carried to the next pass (Fig. 3.1). Upon the end of the final pass, the complete object code corresponding to the source text can be generated. This partial grammar parsing of a pass is illustrated in Fig. 5.1. The i-th pass parser is given a partial grammar (i-th pass syntax) G_i which is a subgrammar of the whole grammar G. It scans the i-th text which is the text output by the

(i-1)-th pass. The parsing is made on partial portions of the text (shaded portion of the text in the figure) which correspond to sentences of the language specified by G_i. The goal symbol S_i of the partial grammar G_i is output to the (i+1)-th text. This goal symbol S_i is treated as a terminal symbol at the next pass. For the other portion of the text not in the language specified by G_i (blank portion of the text in the figure), the i-th pass parser only copies i-th text into (i+1)-th text. Thus, in general, plural analyzed portions are interspersed in the input text of a pass.

Although more detailed and formal description on multipass parsing can be found in [Shi78, Shi79], we shall briefly present the main features of the facilities provided in our parsing scheme, drawing on an example description of a parser for Algol 68 (Fig. 5.2). In the research reported here, we have restricted ourselves to the application of SLR(1) parsing techniques in all passes, though, in the general case, each pass is independent from the others with respect to the class of grammars accepted.

5.1. Starting and terminating partial grammar parsing

Let $L(G_i)$ be the language generated by G_i. In order to catch the starting position for i-th pass parsing, two terminal symbol sets must be given (Fig. 5.1). The first is the set of terminal symbols preceding $L(G_i)$, which will be called $Prec_i$. The second is $FIRST1_{Gi}(S_i)$, the set of first terminal symbols of $L(G_i)$. While the latter can be computed automatically, the former cannot since it is not part of the language $L(G_i)$. Thus, it should be included explicitly in the description of the syntax of each pass. Given the above two sets of terminal symbols, the i-th pass

parsing is "triggered" whenever a terminal symbol a ϵ $Prec_i$ is followed by a terminal symbol b ϵ $FIRSTl_{Gi}(S_i)$, in the i-th text.

Similarly, to find the terminating position of partial grammar parsing, we must explicitly specify in the description the set of terminal symbols succeeding $L(G_i)$, which will be called $Succ_i$ (Fig. 5.1). The parsing proceeds until the parser goes into the accept state. Precisely speaking, the set $Succ_i$ is required so as to decide whether the parser goes into the accept state when it is in the LR(0) conflict state.

To specify these two sets, a description

S_i .BEFORE $Succ_i$ .AFTER $Prec_i$ . . .

is used (see #(D)# in Fig. 5.2).

5.2. Extra goal symbol

The determination of starting position for parsing by two adjacent input terminal symbols as mentioned above has a weak point. As an example, suppose that we are collecting declarations of an Algol 68 program in the first pass. We encounter a difficulty arising from the similarity between "declaration" and "cast" (Fig. 5.3) :

(1) declaration

. . . ; ref int x ; . . .

(2) cast

. . . ; ref int (y) := z ; . . .

In these two cases, assuming that

; ϵ $Prec_i$ and ref ϵ $FIRSTl_{Gi}(S_i)$

hold, the parser enters the partial grammar parsing when catching ";" and "ref". It is undesirable in the cast case.

In order to escape from this difficulty, we adopt a strategy to exit from parsing as soon as we see "(". Namely, we parse only "ref int" in the cast case.

In general, we have designed our parser to be able to exit from parsing, given the specification of a set of nonterminal symbols (S_i')s such that

$$S_i \overset{*}{\underset{G_i}{\Rightarrow}} S_i' \alpha$$

and a set of succeeding terminal symbols (which is a set including "(" in the above case). Here, S_i' is called "extra goal symbol".

The above "cast" case can be specified by

S_i' .BEFORE '('

An example specification can be found at #(D')# in Fig. 5.2.

5.3. Saving and restoring parser state

Another problem of partial grammar parsing arises in the following case. Supposing again that we are collecting declarations of an Algol 68 program in the first pass, the input

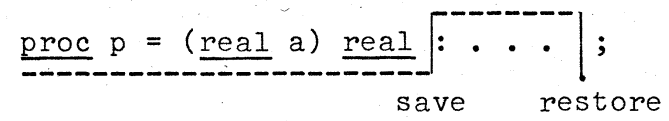
. . . ; int i := 10 , j ;

causes a problem that it contains a portion ":= 10" which we do not want to parse in this pass. Such a case is solved by adding a mechanism for saving parser state, restoring it and continueing partial grammar parsing thereafter. In the above case, the parser state is saved at position ":=" and restored at position "," as follows.

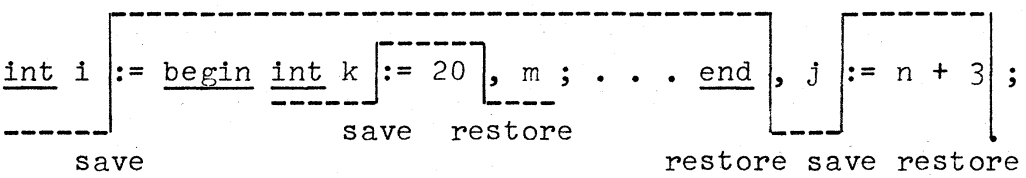
. . . ; int i [:= 10], j ;
save restore

(Underlined portion is the text parsed by partial grammar.)

Another example is



After saving parser state, the partial grammar parsing continues as usual. Thus, nested saving and restoring may arise as follows.



An example description is

```
MVAR .BEFORE ':='                    (save)
     .RETURN ',', ';'                (restore)
```

5.4. Replacing input terminal symbols

There are cases where one finds the necessity of replacing an input terminal symbol by another terminal symbol according to the previously gathered information.

Example :

The identification of variables in Algol 60 procedure bodies cannot be made until the relevant declaration is scanned. For example, the exact specification of "x" and "g" cannot be determined in scanning procedure "f" in the following (cf. [Yon76]):

```
begin
     procedure f ; ... x + g ... ;
     integer procedure g ; g := . . . ;
     integer x ;
     . . .
end
```

In general, such a problem can be treated either by (1)

using the same terminal symbols in the syntax for both cases and distinguishing in the semantics, or by (2) using different terminal symbols in the syntax by distinguishing in the lexical analyzer. Since method (2) seems more simple using multipass parsing, we have included in our system a feature to replace input terminal symbols by other terminal symbols. The replacement is achieved in the lexical analyzer. Assuming that we collect declarations in the first pass, the treatment of "x + g" in each pass will be for example

```
                          x  +  g
(first pass)             id  + id
(second pass)            id  + fid
```

with the following second pass syntax

```
<function reference> ->  fid
<variable>           ->  id
```

To realize this, the lexical analyzer of the second pass replaces input terminal symbol "id" by either "id" or "fid" according to the symbol table information made at the first pass. The relevant description of the second pass to the generator will be

```
.REPLACE 'ID' ?IDFID !(ID,FID) ;
```

where "IDFID" is a procedure to discriminate the two cases. This feature can be applied to the treatment of operators in Algol 68 where their priorities can be declared later in a source text. (see #(F)# in Fig. 5.2).

5.5. Additional features for multipass parsing

In order to fully exhibit characteristic features of multipass parsing and to make the number of passes to the minimum, additional features are included.

5.5.1. Catching range structures

In block structured languages, we would want to catch range structures of source text in addition to the partial grammar parsing which, for example, collects declarations. Some examples of ranges are

begin . . . end

do . . . od

Here, begin etc. are called "range opening symbols" and end etc. are called "range closing symbols".

However, recall that our parser can be given only one partial grammar for each pass. This results in that "begin" etc. are hard to be catched in the same pass as the pass for collecting declarations since "begin" etc. are outside the partial grammar for variable declarations. Thus, we made an additional mechanism to catch range structures.

We have extended the concept of block structures to allow for further subdivision. For example, to deal with the following nested structure of Algol 68:

```
if
|
|  then
|  |
|  else
|  |
fi
```

we define

if . . . then
then . . . else
else . . . fi

as subdivisions of ranges, which will be called "subranges".

The information collected in a subrange can be attached as an attribute to the range opening symbol in the text of later passes. For example, after collecting declarations at the first pass, we can pass the set of declarations for each subrange to the next pass as follows.

```
if
 |-----------------> (i)  (symbol table containing i and j)
    (int i, j ;)

     then
       |----------> (ii) (symbol table containing x)
         (real x ;)

     else
       |----------> (iii)(symbol table containing y)
         (real y ;)
fi
```

Full determination of nesting structures for such subranges is delayed until the next pass, where "active" declarations can be determined at each stage of scan as (i) or (i)+(ii) or (i)+(iii) by connecting declarations in each subrange.

For the above objectives, semantic actions can be called at the time range opening and closing symbols are scanned (#(A)# in Fig. 5.2).

5.5.2. Catching label definitions

A label is a typical example for which uses may precede the definition. Here, we realize again that label definitions are hard to be catched in the same pass as the pass for collecting declarations since label definitions are outside the partial grammar for variable declarations. However, we recognize that label definition forms are usually simple and present almost the same syntax in most programming languages. We have taken

advantage of this fact to process them in a specialized way, in order to enhance efficiency. Thus, labels can be catched by specifying the two terminal symbols for label definitions, for example,

id :

and the set of terminal symbols preceding it, for example

{ begin, then, else, out, do, exit, |, ; }

(#(C)# in Fig. 5.2).

6. Description of semantics

There has not yet been agreement on the best formal notation for the semantics. Attribute grammars as introduced by Knuth [Knu68] make descriptions readable, formal and modifiable. However, their use may result in a dilemma, that is, their power is rather restricted or insufficient compared with real compilers which use hand-coded routines. Another attempt can be found in the CDL compiler-compiler [Kos74] using Affix grammars. It is intrinsically based on top-down parsing. Syntax and semantics are mixed up in a production rule which, in our opinion, results in a rather puzzling representation similarly to hyper-rules of the Wijngaarden grammar as used in [Wij76]. Considering these facts, we have adopted a modified type of attribute grammars as the description style for semantic actions. In our style, evaluation by semantic attributes may be intermixed with evaluation by programs in procedure-oriented language with using variables and tables as in hand-coded routines.

In contrast to the original attribute grammar, the evaluation of semantics in generated compilers is directed by

bottom-up syntax analysis without actually building the syntax tree. Therefore, in a strict sense, inherited attributes cannot be accepted by our system. However, we have confirmed from experience that uses of global entities whose values are determined in previous passes in the multipass parsing can mostly replace uses of inherited attributes. Moreover, considering that the slow processing speed of semantic evaluation in the original attribute grammar results partly from passing of inherited attributes through parse trees, we believe this restriction to be reasonable. In this way, our intermixed description style can realize compile-time semantic evaluation speed as efficient as hand-coded compilers, still preserving readability of description.

An example of intermixed description for syntax and semantics can be seen in Figure 6.1. Semantic attributes are enclosed by < and > in production rules, for example DECS and VALUE are synthesized attributes. CODEFILE is implicitly declared as a synthesized attribute and it corresponds to object codes. ENV is a global entity which can be considered as an inherited attribute.

Our system supports two types of semantic evaluation. The first type of semantic evaluation proceeds as in usual attribute grammars. Namely, if a same attribute appears in both sides of a production rule, such as in

IF<DECS> -> "IF"<DECS> (1)

THEN-CLAUSE<VALUE> -> THEN<DECS> THEN-PART<VALUE> (2)

then, assignment of right-hand side attribute into left-hand side one takes place. In case (1) DECS is assigned, and in case (2) VALUE is assigned. The second type of semantic evaluation

proceeds according to the program enclosed between <* and *>. It is written in a procedure-oriented language, whose meaning would be obvious. As the procedure-oriented language we presently use an Algol 68-style macro-extended language similar to those used for lexical analyzer (cf. Fig. 4.2). The use of macros can further make description readable. For example,

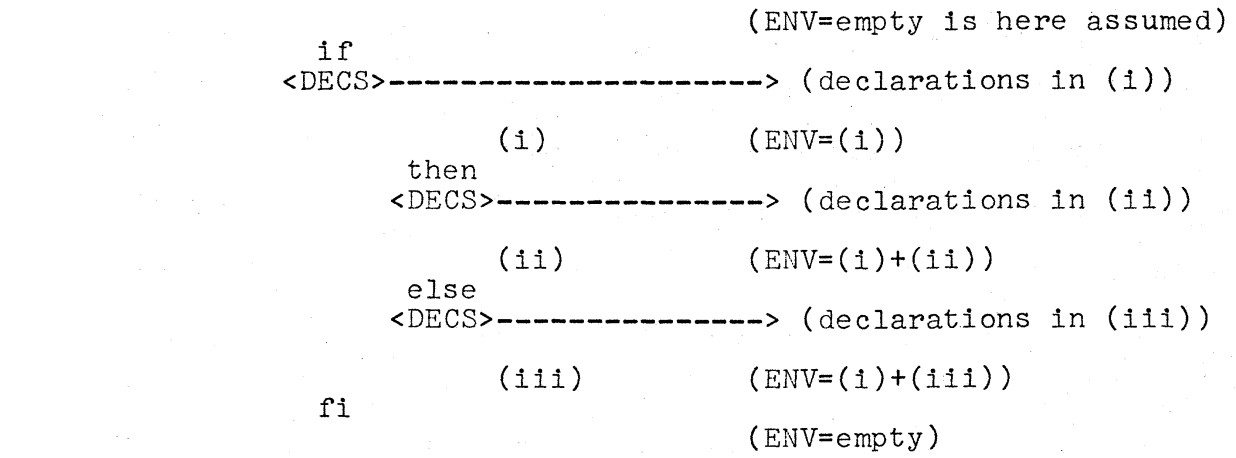
```
ENV +:= DECS                    or
CODEFILE +:= CODE(MNUM)+ ...
```

is no other than procedure call(s).

Subordinate routines called in this semantic part can also be written in the Algol 68-style macro-extended language stated above.

In practical implementation, operations on semantic attributes are converted into operations among semantic stack elements, for both types of the above semantic evaluation. This conversion and the separation of intermixed description into syntax and semantic actions are all made by a "master control program" of the generator (discussed later). Some optimization is made in the first type of semantic evaluation. Namely, if same attributes appear in both sides of a production rule at the same position in the semantic stack as in (1), the assignment operations (of right-hand side attributes into left-hand side ones) are automatically omitted by the system since they are unnecessary.

As an example of Fig. 6.1, we illustrate the treatment of active declarations. Suppose that declarations are collected at the first pass, and that the following text is given to the second pass.



The semantic attribute DECS which is attached to some terminal symbols in the first pass has declarations in the corresponding subrange as its value. ENV is a global entity corresponding to the environment (collection of active declarations at each stage). In scanning "if", the reduction by the production rule (1) takes place, and DECS (for declarations in (i)) is added to ENV. Thus, if no surrounding declarations had existed, ENV may be expressed as ENV = (i). In scanning "then", another reduction by the rule (3) takes place, and DECS (for (ii)) is added to ENV resulting in ENV = (i)+(ii). At the moment immediately before scannning "else", DECS (for (ii)) is subtracted from ENV using the reduction by the rule (5), and in scanning "else", DECS (for (iii)) is added to ENV using the reduction by the rule (7). Now, ENV = (i)+(iii). Thus, the range structure of "if statement" in Algol 68 can be easily realized using subranges.

7. Generator structure for efficient partial regeneration

As stated in section 2.2, we have designed a generator structure so that the task of generating a compiler is divisible into efficient partial (re)generation steps corresponding to each of the description units.

Fig. 7.1 shows the modular structure of the system. The input to the generator consists of description units corresponding to compiler phases. We use the term "grammar" to indicate the unification of syntax and semantics. The "pragmat" part of the description (.PR ... .PR) controls all generation steps such as the specification of generate/not-generate and parameter options for each description unit. OS-dependent parts of the description are wholly confirmed in this "pragmat" part.

The master control program operates only the necessary generation steps according to the above generate/not-generate specification. In order to provide necessary information for partial generation, all interfaces between generation steps, once generated, are automatically preserved or updated in files. Namely, for a generation step specified as "not-generate", the master control program simply takes the corresponding interface files, instead of operating the generation step. For a generation step specified as "generate", the master control program activates the generation step, updating the corresponding interface files. Those interface files are automatically named with using user specified prefix. Further, more partial regeneration in the unit of subordinate procedures or semantic actions is usually allowed owing to the update feature of relocatable binary files, found in most operating systems.

Facilities for generation-time and compile-time automatic

call libraries are also provided.

8. Generated compilers

8.1. Structure of generated compilers

Fig. 8.1 shows the structure of a compiler generated by this system. It consists of modules corresponding to each description unit. The lexical analyzer generated from the corresponding part of the description works coupling with the analyzer of pass 1. As to the analyzer of pass 2 or 3 etc., a standard lexical analyzer is supplied by the system. The natural overlay structure between passes etc. is assumed as a default unless otherwise specified.

The generated compiler translates a source text into codes in a machine-independent intermediate language through multipass parsing. Codes in this intermediate language are then submitted to the optimizer, and lastly to a code generator where they are converted into relocatable binary machine codes.

8.2. Code generator

A machine-independent intermediate language (IL) aiming at production of optimized codes has been designed by Uehara et al. [Ueh78]. Although IL is out of the subject of this report, we shall briefly present its main features.

The language level is kept low enough with excluding peculiarities of particular machines, such as register length and organization, addressing method, and instruction set. An IL statement is a quadruple with one operator field and three operand fields. An address is expressed by (base register no., offset) pair which is suited to Algol-like stack-oriented

languages. Any number of base registers may be used. The code generator has been made with special care for generality and portability. Further details are available in a report [Nak79].

8.3. Optimizer

An optimizer is now under development. Its input text and output text are both in the intermediate language IL. IL is designed so that additional information necessary for optimization can be included in the IL text.

9. Evaluation of the generator system

A two-pass compiler for Algol 68 subset is under development using our compiler-generator [Nag79]. It collects mode-, operator-, variable- declarations and label definitions at the first pass, and the rest is parsed at the second pass.

Although not all characteristic features of our compiler-generator are fully utilized until now, we evaluate our design philosophy with some results gained so far, including the above one.

9.1. Power of compiler description language

The use of appropriate description languages, especially the attribute grammar-style language for semantics has shortened the description and has realized a good document of compiling process.

9.2. Easiness in compiler construction process regeneration

The realization of the generator allowing efficient partial

regeneration along with unification of compiler and user program in the description made the task of developing, testing, debugging and revising a compiler very simple. It will be invaluable in the course of compiler construction and maintenance.

9.3. Machine independence considerations

As suggested in section 2.3, machine-dependent parts should be logically well discriminated from other parts. Although we have no experience of transportation until now, we can evaluate this criterion in the design of our generator, generated compiler and object code, as follows.

(i) generator

OS-dependent parts, e.g., generation of commands in a job control language, have been confined to the master control program. Each sub-generator corresponding to each generation step has been written in Fortran, mostly in an Algol 68-style macro-extended Fortran (as stated before), aiming at future rewriting in Algol 68. Thus, the generator has achieved considerable machine independence.

(ii) generated compiler

Since the present generator outputs each phase of a compiler in macro-extended Fortran, the compiler is highly machine independent. Here again OS-dependent parts such as overlay commands to the linkage editor have been confined to the "pragmat" part of the description or to the master control program of the generator.

(iii) object code

A compiler generated by our generator translates source

texts into codes in a machine-independent intermediate language IL. IL codes are then translated into machine codes by the code generator. As was discussed in section 8.2, machine-dependent parts such as machine instruction set are confined to this code generator, so that object codes for another computer can be easily produced by merely modifying machine-dependent modules of this processor.

9.4 Multipass parsing

The adoption of multipass parsing not only has solved substantial problems of single-pass parsing discussed in chapter 1, but also has simplified compiler description which may present a rather congested semantics in single-pass compilers. For example, a two-pass processing of the famous "labels with block structure" is reduced in about half lines of program compared to the single-pass processing where the troublesome handling of links of labels is required.

With respect to time or space of generated compilers, one may imagine the multipass parsing to be more time-consuming compared with single-pass parsing, although space would be reduced using overlay. However, Table 9.1 shows that for a two-pass parser of XPL [Mac70] (not including semantic evaluation) increasing time is less than 5%, and space is much reduced in about 20% using overlay, compared to one-pass parser.

10. Concluding remarks

Starting from difficulties in processing inverted sequences of definitions and uses of entities in programming languages, we

introduced the concept of multipass parsing. Then, we described a compiler generator which generates compilers allowing multipass scan.

Its design philosophy was presented, and from experience gained so far we believe that the philosophy was successfully realized as follows:

(1) Compact description of compiler was designed, which is a complete and readable documentation of user program's environment. In particular, an attribute grammar-style description was devised which increases compile-time efficiency compared to original attribute grammars.

(2) The generator was organized so that it can efficiently and incrementally regenerate compilers in case a part of compiler description is modified.

(3) Clear separation of machine-dependent part which can realize high portability was achieved in the generator, generated compilers, and object codes.

Moreover, the adoption of multipass parsing has simplified compiler description compared to single-pass parsing, and has reduced space requirement of generated compilers without so much increasing the time requirement.

However, there are problems for future studies. First, the compiler-writer must describe partial grammar for each pass in the present system. One would note that, given the whole grammar G and the goal symbol S_i of an i-th pass, G_i, $Prec_i$ and $Succ_i$ are automatically computable. Thus, we are studying a method for automatically generating the description of each pass from these informations. However, note that even if it is realized, the description of semantics may still have to reflect partial

grammar of each pass. Secondly, the relation between partial grammar and additional features (for catching range structures and label definitions) is still vague in the present system. Their unification together with more clear description style for each description unit is investigated. Thirdly, It would be convenient if the compiler-writer can select grammar class for each pass. We are now developing an LALR(1) parser with disambiguating rules to be added in the repertoire of parsers. Lastly, consolidation of utility features such as error handling would be necessary to make the system a truly utilizable software tool.

In order to make such improvements in future versions, we are required to gain further experiences using our generator.

Acknowledgements

The authors wish to thank Sanya Uehara, Hideaki Tazaki and Yukio Nagasawa for their contributions to the design of the system.

References

[Wij76] Wijngaarden,A.V., et al., Revised report on the algorithmic language ALGOL 68, Springer- Verlag, 1976.

[Ino77] Inoue,K., et al. , A generation-method of multiphase-compilers, 18th Proc. IPSJ 302, (1977).

[Gan77] Ganzinger,H., et al., Automatic generation of optimizing multipass compilers, Information Processing 77, pp. 535 - 540 (1977).

[Les75] Lesk,M.E., LEX- a lexical analyzer generator, CSTR 39, Bell Laboratories, 1975.

[Sas79] Sassa,M., A pattern matching macro processor, to appear in Software- Practice and Experience, Vol.9 (1979).

[Sas78] Sassa, M., et al. , A framework for a multipass-compiler generator, 19th Proc. IPSJ 3C-6 (1978).

[Shi78] Shinogi,T., et al. , On generation of partial grammar parsers, 19th Proc. IPSJ 3C-5 (1978).

[Shi79] Shinogi,T., Research on automatic generation of multipass parsers (in Japanese), Master thesis, Tokyo Institute of Technology, Dept. of Information Sciences, 1979.

[Yon76] Yoneda,N., and Noshita,K., Lectures on Algol 60 (in Japanese), bit 8, 13 (Dec. 1976).

[Knu68] Knuth,D.E., Semantics of context-free languages, Mathematical systems theory, 2, 2, pp. 127 - 145 (1968).

[Kos74] Koster,C.H.A., Using the CDL compiler-compiler, in Compiler construction- an advanced course, Lecture Notes in Computer Science 21, Springer- Verlag, 1976.

[Ueh78] Uehara,S., et al., Design of an intermediate language for generation of optimized code, 19th Proc. IPSJ, 3C-8 (1978).

[Nak79] Nakamura,S., Code generator with machine-independence and portability considerations (in Japanese), Bachelor thesis, Tokyo Institute of Technology, Dept. of Information Sciences, 1979.

[Nag79] Nagasawa,Y., et al., A two-pass compiler for ALGOL68 using a multipass-compiler generator, to apperar in 20th Proc. IPSJ (1979).

[Mac70] McKeeman, W.M., et al., A compiler generator, Prentice-Hall, 1970.

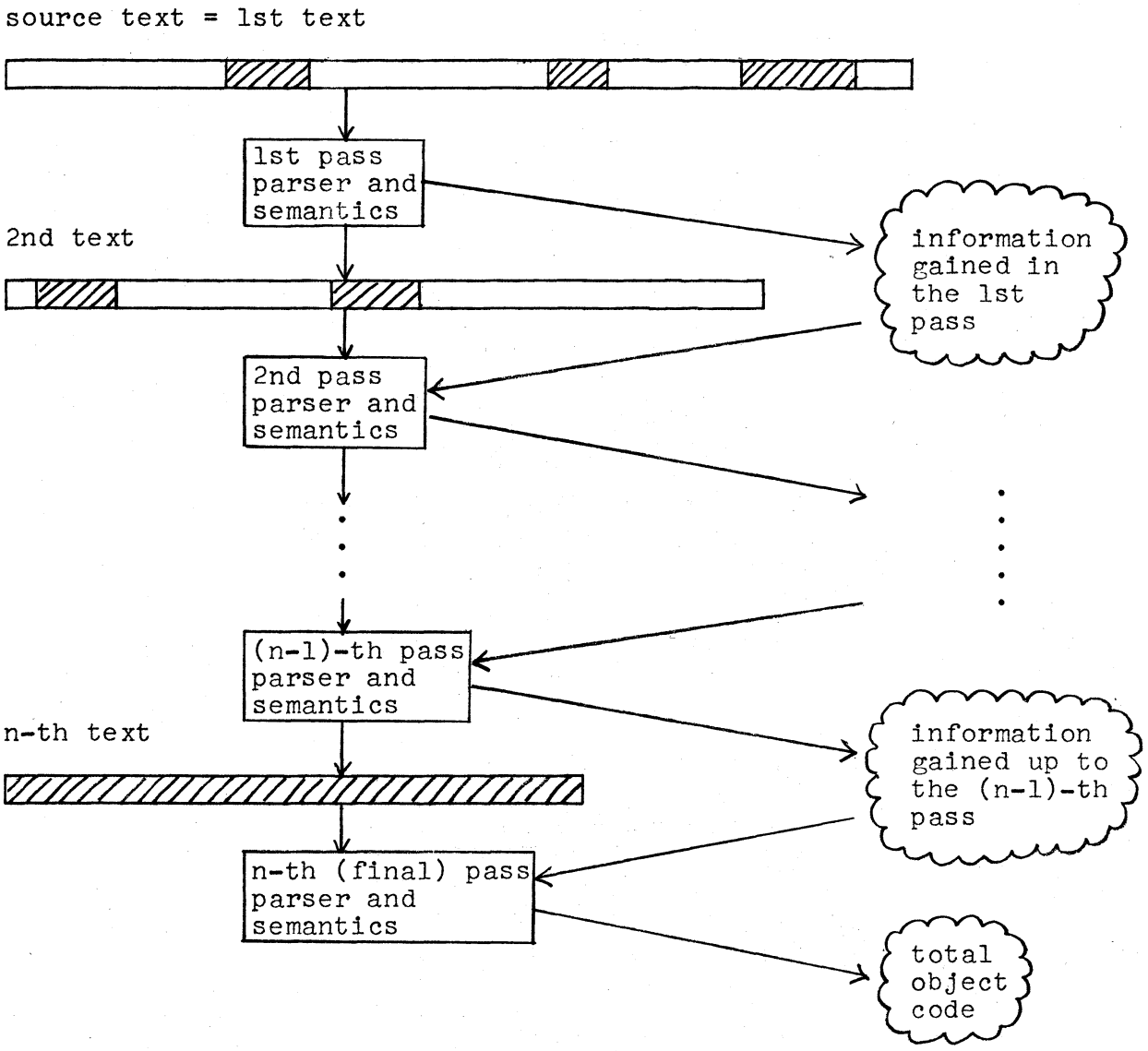


Fig. 3.1 Schematic view of multipass parsing

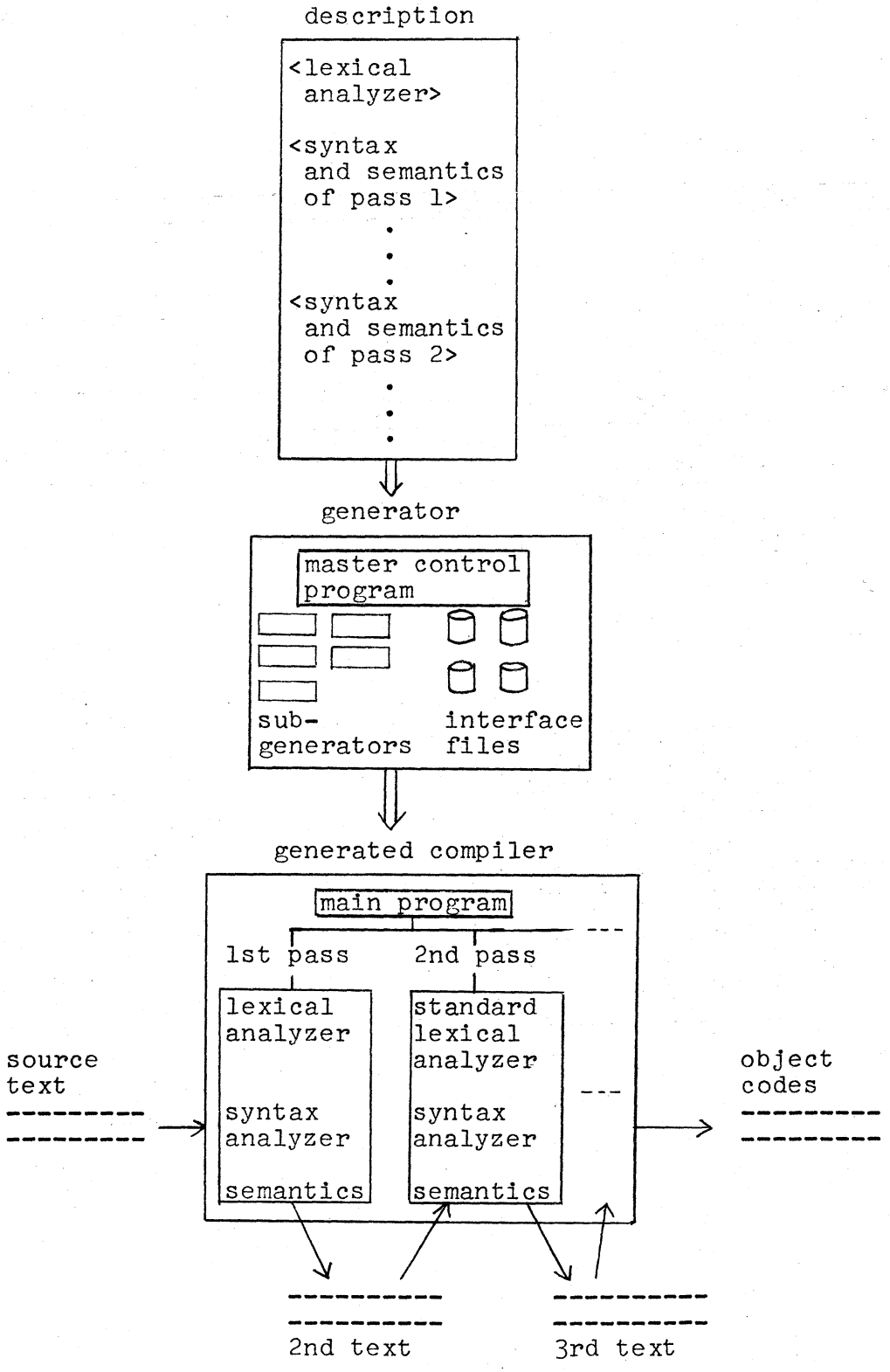


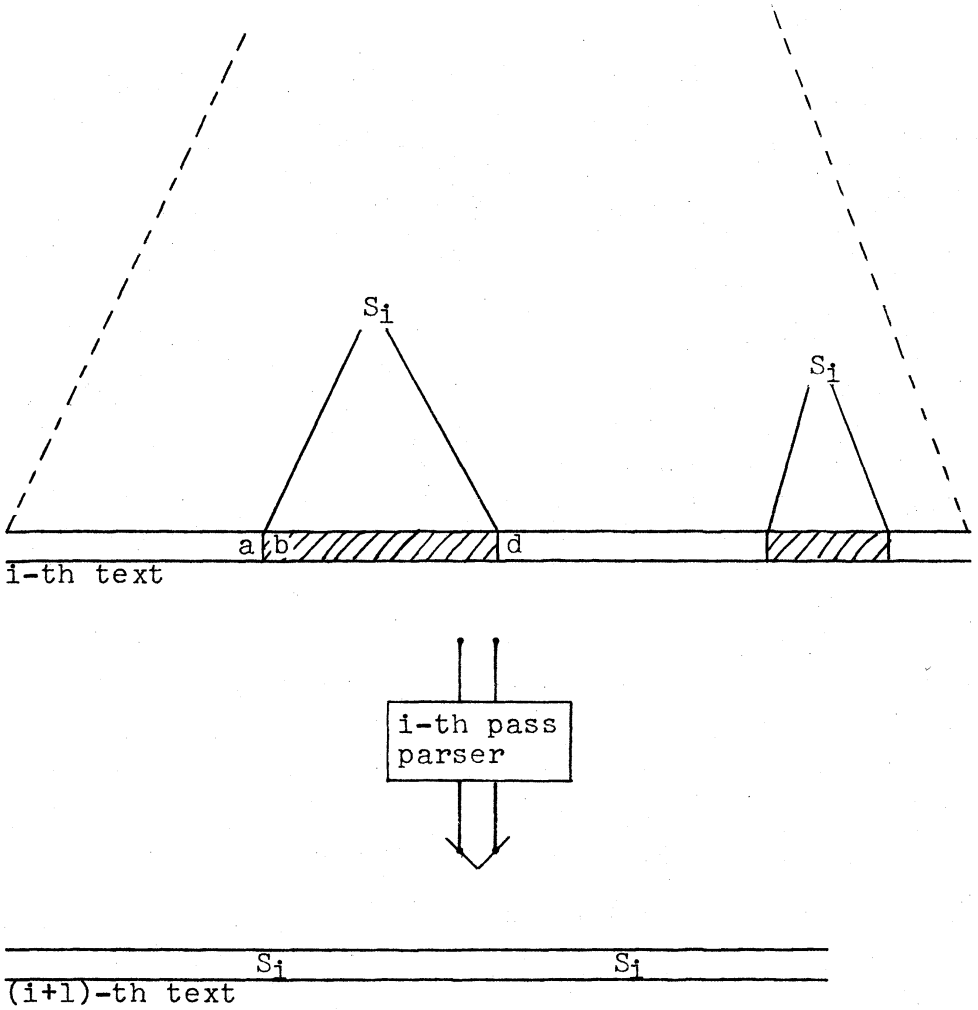
Fig. 3.2 Schematic view of generator

```
.LEXICAL
.CLASS    SP    = " " ,
          LET   = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  :IDSEQ,
          DIG   = "0123456789"                  :DIGSEQ,
          PERIOD = "."                          :PERSEQ,
          QUOTE = """'"                         :STRING,
          COM   = "#"                           :COMMNT,
          SYM   = "();,@$"                      :SYMBOL,
          COL   = ":"                           :COLSEQ,
          BAR   = "|"                           :BARSEQ,
          MON   = "+-¬%&"                       :OPSEQ,
          NOM   = "<>/*"                        :OPSEQ,
          EQU   = "="                           :EQSEQ,
          OTHER = "?_!"                         :NEG;
.TABLE    TERM1 = "();,@$" ,
          TERM2 = 'BEGIN','END','EXIT','PAR','IF','THEN',
                  'ELIF','ELSE','FI','CASE','IN','OUT',
                  'OUSE','ESAC','AT','IS','ISNT','NIL','OF','GO',
                  'GOTO','SKIP','FOR','FROM','BY','TO','WHILE',
                  'DO','OD','COMMENT','CO','PRAGMAT','PR',
                  'LONG','SHORT','REF','LOC','HEAP','STRUCT',
                  'FLEX','PROC','UNION','OP','PRIO','MODE',
                  'INT','REAL','BOOL','CHAR','FORMAT','VOID',
                  'COMPL','BITS','BYTES','STRING','SEMA',
                  'FILE','CHANNEL','TRUE','FALSE','EMPTY' ;
.SYMBOL   IDSEQ  = LET < LET !DIG ! -SP>   ?IDENT !ID ,
          PERSEQ = -PERIOD(LET<LET!DIG> .IN TERM2 .OUT ?BOLDWD !BL
                    !/ ?NUMBER !RCON),
          STRING = -$ < ¬$ ! -$$ > -$   ?STORE !(CHARCON,STRCON),
          COMMNT = -$ <-¬$ > -$,
          SYMBOL = SYM .IN TERM1,
          COLSEQ = ':'('='(':'! !/! )!'/='(':'!!/?ERROR)!/!),
          BARSEQ = '|'(':'! !/! ),
          OPSEQ  = $(NOM(':='!'=:'!/)!'='(':'('='!/)!'=:'!/)!':='!/)
                    ?OPRAT!OPR,
          EQSEQ  = '='(NOM(':=' ?OPRAT !OPR!'=:' ?OPRAT!OPR!/?OPRAT!OPR)
                    !'='(':'('='?OPRAT!OPR!/?OPRAT!OPR)!'=:'?OPRAT!OPR
                    !/?OPRAT!OPR)!':='?OPRAT!OPR!/!) ,
          DIGSEQ = DIG  ?NUMBER  !(ICON,RCON,BCON) ,
          NEG    = OTHER  ;
```

FIG. 4.1 AN EXAMPLE DESCRIPTION OF LEXICAL ANALYZER USING REGULAR EXPRESSIONS (CF. ALGOL 68)

```
.PROC IDENT = .VOID :
.BEGIN
   .INIT HASHX = 0 ;
   .IF  HASHX > LENH
      .THEN  PUT1('ERROR HASH OVERFLOW')
      .ELSE  PNTR := ENTRYS(S,HT,HASHX)
   .FI ;
.END ;
```

FIG. 4.2 AN EXAMPLE DESCRIPTION OF SUBORDINATE ROUTINE FOR LEXICAL ANALYZER AND FOR SEMANTICS (IN ALGOL68-STYLE MACRO-EXTENDED FORTRAN)



SLR(1) parsing is assumed.

S_i : goal symbol of partial grammar G_i of pass i

$a \in Prec_i$, $b \in FIRST1_{G_i}(S_i)$

$d \in Succ_i$

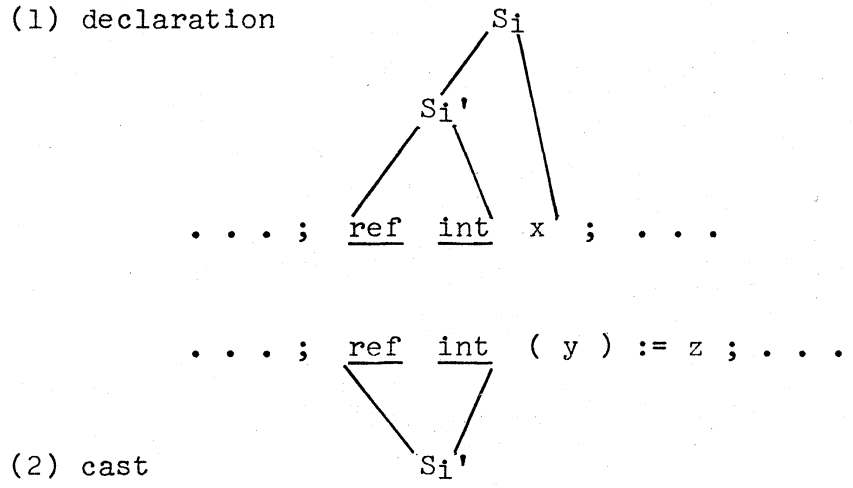
Fig. 5.1 Partial grammar parsing of pass i

```
,GRAMMAR 1

,SUBRANGE
                'BEGIN':'END','IF':'THEN','ELIF':'THEN' ,
                'THEN':'ELSE','THEN':'ELIF','THEN':'FI' ,
                'ELSE':'FI','CASE':'IN','OUSE':'IN',
                'IN':'OUT','IN':'ESAC','OUT':'ESAC' ,
                'WHILE':'DO','DO':'OD','(':')','(':'|' ,
                '|':'|','|':'|','|':'|:','|':')'
                ?SEML:SEMR;                                          #(A)#

,MARK           ','?M1 ;                                             #(B)#

,LABEL          'ID' ':'
                ,AFTER 'BEGIN','THEN','ELSE','OUT' ,
                       'DO' , '|' , ';' , 'EXIT'
                              ?SEMLDE !LAB;                          #(C)#

,SYNTAX
DEC  ,BEFORE ';' , ')' ,

MOP  ,BEFORE  '(','GO','GOTO','SKIP','ID','LOC','HEAP' ,
       'NIL' ,
       '=' , 'OPR' , 'BL' ,
        'ICON','RCON','TRUE','FALSE','STRCON','CHARCON' ,
        'BCON','EMPTY',
        'REF','PROC','UNION','BYTES' ,
        'STRUCT','BOOL','CHAR','STRING','FORMAT','SEMA' ,
        'FILE' ,
        'CHANNEL','INT','LONG','SHORT','REAL','COMPL' ,
        'BITS' ,
        'BEGIN','IF','CASE' ,'(','PAR','FOR','FROM','BY' ,
        'TO','WHILE','DO'
     ,RETURN ',',';',                                                #(D')#

MID  ,BEFORE  '(' ,'GO','GOTO','SKIP','ID','LOC' , 'HEAP' ,
       'NIL' ,
      '=','OPR','BL',
        'ICON','RCON','TRUE','FALSE','STRCON','CHARCON' ,
        'BCON','EMPTY',
        'REF','PROC','UNION','BYTES' ,
        'STRUCT','BOOL','CHAR','STRING','FORMAT','SEMA',
        'FILE' ,
        'CHANNEL','INT','LONG','SHORT','REAL','COMPL' ,
        'BITS' ,
        'BEGIN','IF','CASE','(','PAR','FOR','FROM','BY' ,
        'TO','WHILE','DO'
    ,RETURN ',',';',

ROP ,BEFORE  ':'
    ,RETURN ',',';',

RID ,BEFORE  ':'
    ,RETURN ',',';',

RVAR ,BEFORE  ':'
     ,RETURN ',',';',

MVAR ,BEFORE ':=' , ';' , ',' , ')'
     ,RETURN ',',';',

PRI ,BEFORE  ',' , ';'
    ,RETURN ',',';',
```

```
MOD ,BEFORE  ',',';'
    ,RETURN ',',';',

MOPO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
MIDO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
ROPO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
RIDO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
RVARO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
MVARO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
PRIO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',
MODO ,BEFORE 'OP','PRIO','MODE','(','FLEX','STRUCT','LOC','HEAP',

STRCT ,BEFORE '('
      ,RETURN 'ID',

VAF ,BEFORE  '(','BEGIN','IF','CASE','FOR','PAR',':',')' ,
             'FROM','BY','TO','WHILE','DO',

VOID_D ,BEFORE  '(','BEGIN','IF','CASE','FOR','PAR',':',')' ,
                'FROM','BY','TO','WHILE','DO',

SAM_GF ,BEFORE  '('
       ,RETURN 'BL','ID','FLEX','STRING','REF','PROC',
               'UNION','BOOL','CHAR','STRING','FORMAT',
               'SEMA','FILE','CHANNEL','LONG','SHORT',
               'INT','REAL','COMPL','BITS','BYTES',

FLX    ,BEFORE '('
       ,RETURN 'BL','ID','FLEX','STRING','REF','PROC',
               'UNION','BOOL','CHAR','STRING','FORMAT',
               'SEMA','FILE','CHANNEL','LONG','SHORT',
               'INT','REAL','COMPL','BITS','BYTES',

FLST ,BEFORE '('
     ,RETURN 'ID',

GFLST ,BEFORE '('
      ,RETURN 'ID',

MOD_H ,BEFORE '('
      ,RETURN 'BL','ID','FLEX','STRING','REF','PROC',
              'UNION','BOOL','CHAR','STRING','FORMAT',
              'SEMA','FILE','CHANNEL','LONG','SHORT',
              'INT','REAL','COMPL','BITS','BYTES',

MOD_T ,BEFORE  '('

,AFTER          'BEGIN','IF','THEN','ELSE' ,'ELIF','CASE',
                'OUSE','WHILE','DO','(','|' ,'|:',';',')',',' ;
                                                        #(D)#


DEC                 ->   S                                  ,  #(E)#
S                   ->   MOP|MID|ROP|RID|RVAR|MVAR|PRI|MOD     ,

MOPO                ->   MOP   ','                              ,
MIDO                ->   MID   ','                              ,
ROPO                ->   ROP   ','                              ,
RIDO                ->   RID   ','                              ,
RVARO               ->   RVAR  ','                              ,
MVARO               ->   MVAR  ','                              ,
PRIO                ->   PRI   ','                              ,
MODO                ->   MOD   ','                              ,
```

```
MOP<ENTRY>          ->    'OP' FOR_PROC_PLAN<MP> OPRA<SYM> '='
                           <* ,CASE FECM2(MP)
                                ,IN ATL:=NIL;
                                 ATR:=FECM2(MP+1);
                                 ATT:=MP+2 ,
                                 ATL:=FECM2(MP+1);
                                 ATR:=FECM2(MP+2);
                                 ATT:=MP+3
                                ,OUT PUTF(6,(*'MOP',10X,
                                     'PARAMETER NOT BETWEEN 1 AND 9'*,));
                                     CALL DUMP1;
                                     STOP
                                ,ESAC;
                            ENTRY := ADOP(SYM,ATL,ATR,ATT,UNDEF,1) *> |


                          'OP' FOR_PROC_PLAN<MP> 'BL'<SYM> '='
                           <* ,CASE FECM2(MP)
                                ,IN ATL:=NIL;
                                  ATR:=FECM2(MP+1);
                                  ATT:=MP+2 ,
                                  ATL:=FECM2(MP+1);
                                  ATR:=FECM2(MP+2);
                                  ATT:=MP+3
                                ,OUT PUTF(6,(*10X,
                                 'PARAM NOT BETWEEN 1 AND 9 ---> MOP'*,));
                                 CALL DUMP1;
                                 STOP
                                ,ESAC;
                        ENTRY:=ADOP(-SYM,ATL,ATR,ATT,UNDEF,1) *>    |

                        # 'OP' 'BL' ( OPRA | 'BL' ) '='  |#

                          MOP0       OPRA<SYM>  '='
                           <* ENTRY:=CPYOP(SYM,1) *>              |

                          MOP0       'BL'<SYM>  '='
                            <* ENTRY:=CPYOP(SYM,1) *>             ,

                              ,
                              ,
                              .

,GRAMMAR 2

,REPLACE   'OP' ?ADDLEX
           !(OP0,OP1,OP2,OP3,OP4,OP5,OP6,OP7,OP8,OP9);        #(F)#

,SYNTAX PROGRAM ;
    PROGRAM      ->     ,       ,                              #(G)#
                        ,
                        ,
```

FIG. 5.2 AN EXAMPLE DESCRIPTION OF PARTIAL GRAMMARS FOR PASS 1 AND 2

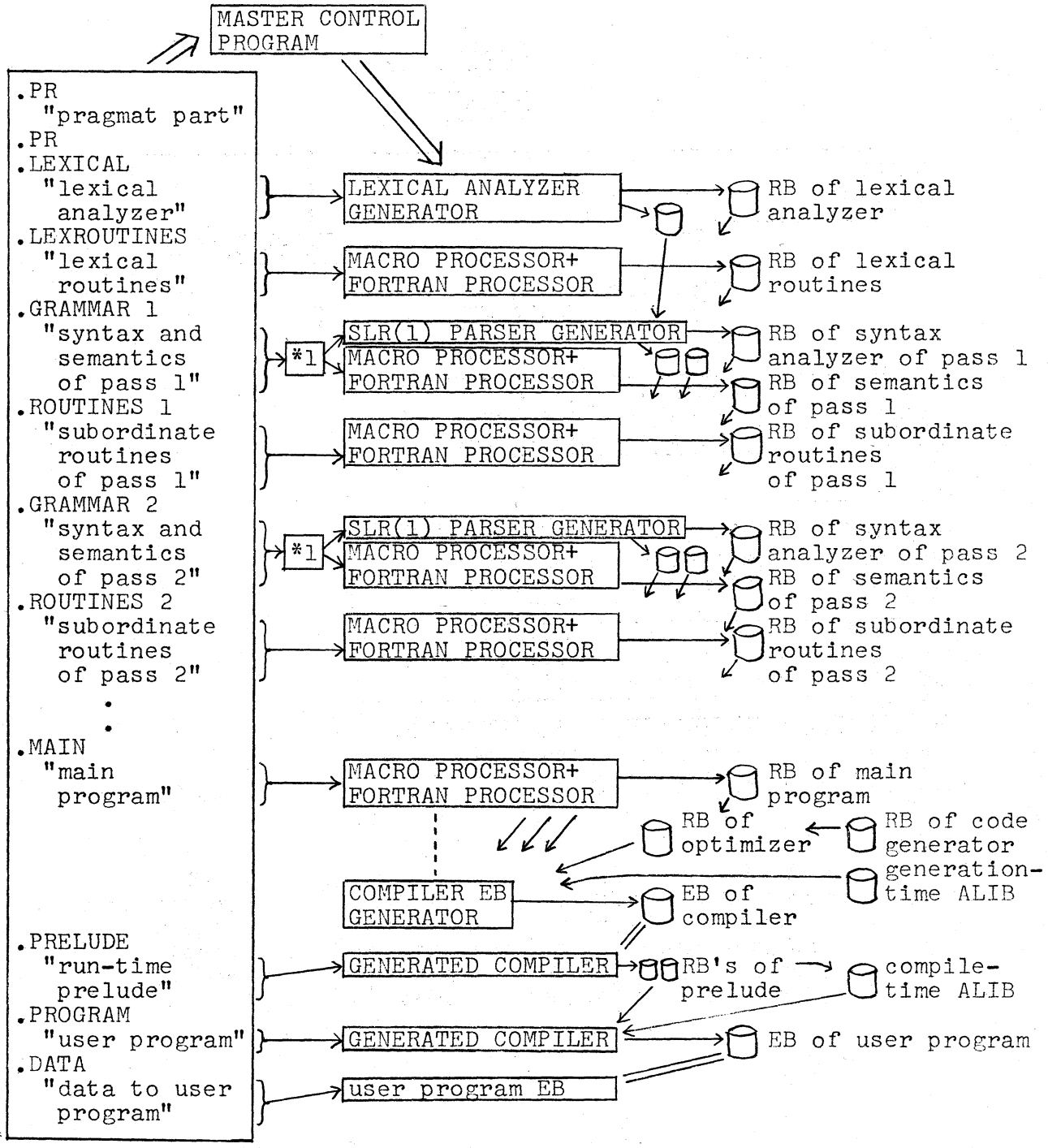


In order to exit from the parser, specify
a nonterminal symbol S_i' and
a set of succeeding terminal symbols
which contains "(" in this case.

Fig. 5.3 Extra goal symbol

```
.GRAMMAR 2
            .
            .
            .
#(1)#   IF<DECS>            ->  "IF"<DECS>  # ATTRIBUTE GRAMMAR STYLE #
          <* ENV +:= DECS *>                                    ,
#(2)#   ENQUIRY-CLAUSE<VALUE> -> DEC-SERIES<VALUE1>
          <* P := COERCE(MODE(VALUE1),BOOLEAN) ;
             ,IF  P = 0  ,THEN  ERROR  ,FI ;
             VALUE := TEMP(RESULTMODE(P)) ;
             CODEFILE +:= CPROC3(P,VALUE,VALUE1) ;
             CODEFILE +:= CODE(VALUE<=0|->LNUM) ;
             LPUSH(LNUM) *>                          # PROCEDURE-ORIENTED
                                                       LANGUAGE STYLE #  ,
#(3)#   THEN<DECS>            ->  "THEN"<DECS>
          <* ENV +:= DECS *>                                    ,
#(4)#   THEN-PART<VALUE>     ->  SERIAL-CLAUSE<VALUE>          ,
#(5)#   THEN-CLAUSE<VALUE>  -> THEN<DECS>  THEN-PART<VALUE>
          <* ENV -:= DECS *>                                    ,
#(6)#   ELIF<DECS>           ->  "ELIF"<DECS>
          <* ENV +:= DECS ;
             CODEFILE +:= CODE(MNUM)+CODE(->LNUM)+CODE(LSTACK(LTOP):) ;
             LPOP ;  LPUSH(LNUM) ;  MPUSH(MNUM) *> ,
#(7)#   ELSE<DECS>           ->  "ELSE"<DECS>
          <* ,SAME *>                                           ,
#(8)#   ELSE-PART<VALUE>     ->  SERIAL-CLAUSE<VALUE>,
#(9A)#  ELSE-CLAUSE<VALUE>  -> ELSE<DECS>  ELSE-PART<VALUE>
          <* ENV -:= DECS ;
             CODEFILE +:= CODE(MNUM) ;
             MPUSH(MNUM) *>                                     |
#(9B)#    ELIF<DECS>  BOOLEAN-CHOOSER-CLAUSE<VALUE>
          <* ,SAME *>                                           ,
#(10A)# ALTERNATE-BOOLEAN-CLAUSE<VALUE>  ->  THEN-CLAUSE<VALUE>
          <* CODEFILE +:= CODE(LSTACK(LTOP):) ;
             LPOP *>                                            |
#(10B)#   THEN-CLAUSE<VALUE1>  ELSE-CLAUSE<VALUE2>
          <* P := COMMONTO(MODE(VALUE1),MODE(VALUE2)) ;
             ,IF  P = 0  ,THEN  ERROR  ,FI ;
             VALUE := TEMP(RESULTMODE(P)) ;
             CODEFILE2 +:= CPROC2(P,VALUE,VALUE1,VALUE2,
                                  MSTACK(MTOP),MSTACK(MTOP-1)) ;
             CODEFILE +:= CODE(LSTACK(LTOP):) ;
             LPOP *>                                            ,
#(11)#  BOOLEAN-CHOOSER-CLAUSE<VALUE>  ->
          ENQUIRY-CLAUSE   ALTERNATE-BOOLEAN-CLAUSE <VALUE> ,
#(12)#  BOOLEAN-CHOICE-CLAUSE<VALUE>   ->
          IF<DECS>  BOOLEAN-CHOOSER-CLAUSE<VALUE>  "FI"
          <* ENV -:= DECS *>                                    ,
            .
            .
            .
```

FIG 6,1 AN EXAMPLE DESCRIPTION OF
SYNTAX AND SEMANTICS
USING ATTRIBUTE GRAMMAR-LIKE DESCRIPTION +
PROCEDURE-ORIENTED LANGUAGE
(BOOLEAN-CHOICE-CLAUSE IN ALGOL 68)



(a)description (b) generation steps (c) interface files

*1 = syntax/semantics separator + attribute conversion translator
RB=relocatable binary, EB=execution binary,
ALIB=automatic call library
Inclusion of the description in files is possible using
".FILE = <file name and vol> ;"
or ".INCLUDE <i> = <file name and vol> ;".
Master control program executes only necessary generation steps.
Generated compiler is shown in Fig. 8.1.

Fig. 7.1 Modular structure of generator

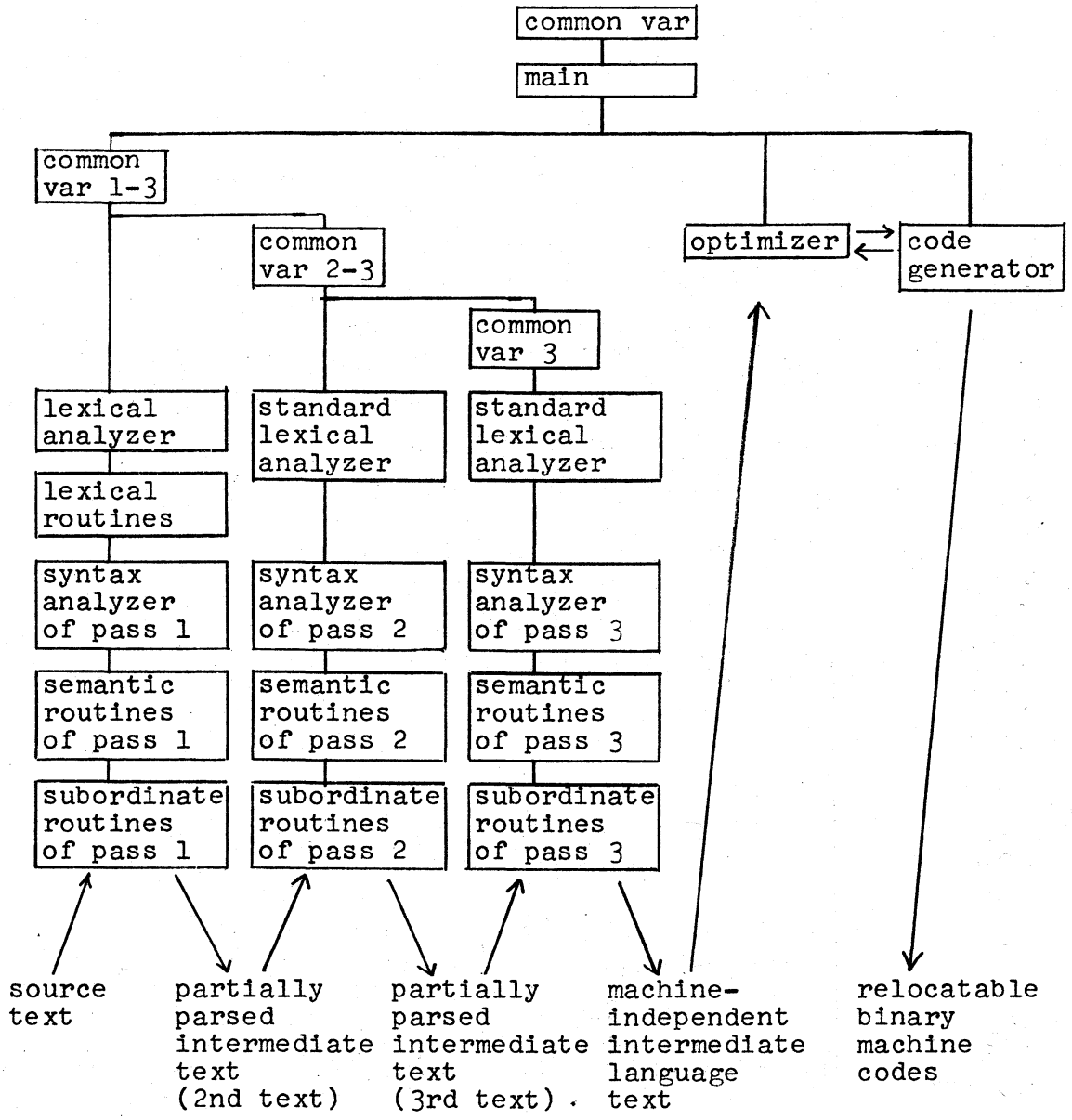


Fig. 8.1 Structure of a generated compiler (three-pass case)

	one-pass XPL	two-pass XPL with overlay	rate (two-pass/one-pass)
parsing table size (byte)	2392	1st pass 668 2nd pass 1924	0.804
parsing time of example program 1 (sec)	3.187	3.325	1.043
parsing time of example program 2 (sec)	5.214	5.341	1.024

Example program 1 : about 60 lines
Example program 2 : about 80 lines

Parsing time does not include semantic evaluation time.

Table 9.1 A comparison of one-pass and two-pass parsers for XPL