Design and Implementation of a Multipass-Compiler Generator

MASATAKA SASSA*, JUNKO TOKUDA*, TSUYOSHI SHINOGI**, and KENZO INOUE*

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

The concept of multipass partial grammar parsing is presented, and relevant new features for its realization in a compiler generator are shown.

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 a modified attribute-grammar, and run-time prelude.

To facilitate a long term effort for compiler construction, the compiler generator is organized so that it can incrementally or partially generate or regenerate a compiler.

1. Introduction

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

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

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

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

Another commonly arising difficulty in block structured languages is the identification of variables in procedure bodies, as follows (cf. Algol 60).

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

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

Similar difficulties arise in processing mutually recursive procedures, or in processing declarations and uses of operator (priority) and mode in Algol 68 [1].

As the above mentioned difficulties cannot be dealt with efficiently in a single-pass compiler, we have proposed in a previous note [2] a compiler generator (often called compiler-compiler or translator writing system) which generates compilers allowing multipass parsing and optimization. (see also [3, 4, 5, 6])

As far as we know, compiler generators for multipass parsing do not exist. Most of the existing compiler generators are just able to generate single-pass compilers [7–14]. Although there have been some attempts at automatic generation of multipass compilers, they do not realize multipass *parsing*. For example, in MUG2 [15], the compiler generated by the compiler generator creates at the first pass a (somewhat optimized) parse tree, and at later passes it gives semantic attributes to nodes of the tree, evaluates them, and makes optimizing transformation on the tree. Since this tree construction should be completed during the first pass, the parsing is essentially single-pass. This cannot settle difficulties such as above mentioned ones for Algol 68 compilers. For example, the complete analysis of an "expression" could not be made during the first pass in case their operator priorities are defined later in the source text.

In contrast to these systems, we have adopted a real multipass parsing without creating the parse tree. In this note, we present the concept of multipass parsing, and we show relevant new features for its realization in our compiler generator.

To describe our system, we first present design philosophy, and then give an outline of multipass parsing and the compiler generator. Next, lexical analyzer, multipass parsing and semantic processing will be described in order.

*Department of Information Sciences, Tokyo Institute of Technology, Ookayama, Meguro-ku, Tokyo 152, Japan.
**Fujitsu Ltd., Kami-odanaka, Nakahara-ku, Kawasaki 211, Japan.

2. Design Philosophy

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

(1) Complete, readable, easily modifiable compiler description: —

Compiler generators are not only designed for saving labor, but are also a tool for complete and formal description of compilers which are really complex objects. To make the description complete and integrated, our description includes (i) user programs and data, (ii) the run-time prelude, and (iii) the usual compiler description. In other words, our system unifies compiler generator and compiler. To make the description formal, readable and easily modifiable, an appropriate description language is supplied for each description unit.

(2) An efficient usable compiler generator which can flexibly and partially regenerate a compiler whenever part of the description is changed in future: —

The criterion (1) could not be guaranteed by a system which requires the regeneration of the whole process, on account of a single bug in the compiler description.

(3) Machine-independence considerations: —

The portability of a compiler generator is also closely related to its products and descendants, i.e., generated compilers and object codes generated by them. We have taken care to logically discriminate machine-dependent parts from other parts in all of the above three levels of systems.

3. Overview of Multipass Partial Grammar Parsing and of the Compiler Generator

Our objective is to automatically generate a compiler which analyzes source text by multipass parsing. Brief explanation of multipass parsing will be shown using a schematic view in Fig. 1. The parser of each pass usually reads input text of the pass (blank portion in the figure) and copies it 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. Usually plural analyzed portions are interspersed in input and intermediate texts. Thus, our scheme should be more strictly called "multipass partial grammar parsing". (The simple term "multipass parsing" will also be used in this paper.) For reader's convenience, we have filled up Fig. 1 with an example of the compilation process for Algol 68, although multipass parsing is by no means special to that language as can be seen in sequential and concurrent Pascal [16, 17]. In the example of Fig. 1, user-defined mode indicators and user-defined operator symbols are collected at pass 1, declaration parts are parsed at pass 2, then at final pass the rest of source text is parsed and the complete object code corresponding to source text is generated.

Next, a schematic view of our multipass-compiler generator is shown in Fig. 2.

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

(1) Lexical analyzer (of the first pass): regular expressions (to be presented later).

(2) Syntax and semantics for each pass: modified

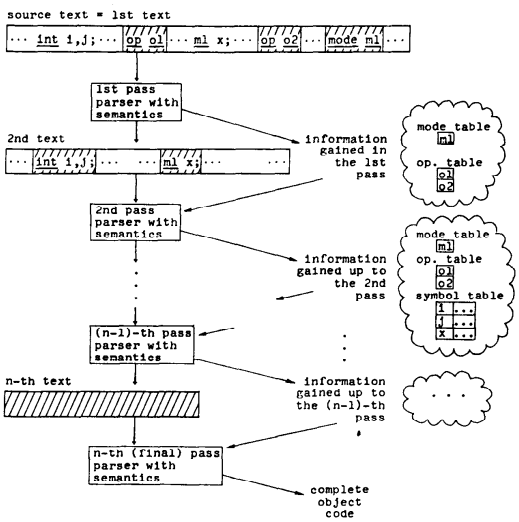

Fig. 1 Schematic view of multipass partial grammar parsing.

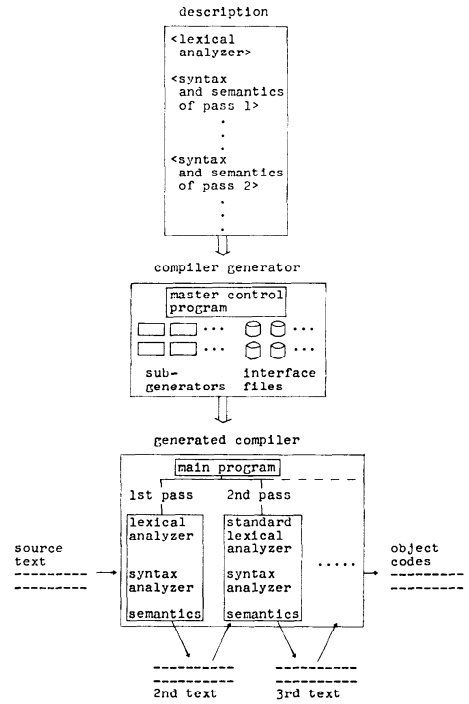

Fig. 2 Schematic view of multipass-compiler generator.

attribute-grammar description with procedure-oriented language (Algol 68-style macro-extended Fortran) (to be presented later).

(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 programs and data: the language acceptable by the generated compiler.

The compiler generator inputs the above description and outputs a multipass compiler. The compiler generator consists of a "master control program", several sub-generators, and interface files for passing information among these sub-generators.

A compiler generated by our compiler generator consists of modules corresponding to each description unit. For the analyzer of second to last passes, a standard lexical analyzer is supplied. The compiler analyzes source text according to multipass parsing and outputs object codes.

4. Lexical Analyzer Generator

There are several tools for automatically generating lexical analyzers, for example Lewi's [10] and LEX [18]. We adopt here a similar but extended approach using regular expressions.

Figure 3 is an example description of lexical analyzer for Algol 68. In general, the description consists of a set of "class" 's which are collections of characters to be used to discriminate terminal symbols (tokens), a set of "table"s which are the tables of keywords etc., and a set of "symbol"s which specify terminal symbols corresponding to each "class". "Symbol"s are described in regular expressions together with procedure names and terminal

```
,LEXICAL
,CLASS    SP  = " "@" ,
          LET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  :IDSE@,
          DIG = "0123456789"                  :DIGSE@,
          PERIOD = ".,"                       :PERSE@,
          @UOTE = """"                        :STRING,
          COM = "#"                           :COMMNT,
          SYM = "()[],@$"                     :SYMBOL,
          COL = ":"                           :COLSE@,
          BAR = "|"                           :BARSE@,
          MON = "+-¬%&"                       :OPSE@,
          NOM = "<>/@"                        :OPSE@,
          E@U = "="                           :E@SE@,
          OTHER = "?_!"                       :NEG!
,TABLE    TERM1 = "()[],@$" ,
          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   IDSE@  = LET < LET !DIG ! -SP>   ?IDENT !ID ,
          PERSE@ = -PERIOD(LET<LET!DIG> ,IN TERM2 ,OUT ?BOLDWD !BL
                   !/ ?NUMBER !RCON),
          STRING = -$ <-''@' ! -$ ! -$$ > -$    ?STORE !(CHARCON,STRCON),
          COMMNT = -$ <-¬$ > -$,
          SYMBOL = SYM ,IN TERM1,
          COLSE@ = '!'(':='(':'! !/! )!'/='(':'!!/?ERROR)!/!),
          BARSE@ = '!'(':'! !/! ),
          OPSE@  = $(NOM(':='!'='!/)!'='(':'('='!/)!'='!'!/)!'!'='!/)
                   ?OPRAT!OPR,
          E@SE@  = '='(NOM(':='  ?OPRAT !OPR!'=:'  ?OPRAT!OPR!/?OPRAT!OPR)
                   !'='(':'('='?OPRAT!OPR!/?OPRAT!OPR)!'='!'?OPRAT!OPR
                   !/?OPRAT!OPR)!':='?OPRAT!OPR!/!) ,
          DIGSE@ = DIG  ?NUMBER  !(ICON,RCON,BCON) ,
          NEG    = OTHER  !
```

Fig. 3 An example description of lexical analyzer using regular expressions (for Algol 68).

symbol names. Procedure names are prefixed by "?". These procedures are pieces of codes to be executed whenever a terminal symbol specified by the relevant regular expression is recognized. They can be written by compiler-writer using an Algol 68-style language which are translated into Fortran by a pattern matching macro processor [19]. Terminal symbol names are prefixed by "!". The names themselves can be used as terminal symbols in the description of syntax.

The description for lexical analyzer extends regular expressions in several aspects. Some of the main features are illustrated by the following examples using Fig. 3. (For further details, see [29].)

***Example of bold tag symbol*:**

Suppose for example that ".BEGIN" is given as input. (The preceding "." is for hardware representation.) The lexical analyzer scans ".", selects the "class" PERIOD, and jumps to "symbol" PERSEQ. It skips the first character which belongs to PERIED (in our description, "—" means skip, i.e., does not store in the string area) and scans input while the regular expression LET(LET|DIG)* is being recognized. Here, $\langle \cdots | \cdots | \cdots \rangle$ in the description means $(\cdots|\cdots|\cdots)^*$ in the usual mathematical notation for regular expressions. Scanned characters are usually stored in the string area, and now the string "BEGIN" is stored there. The lexical analyzer searches for "BEGIN" in "table" TERM2 (by .IN), succeeds in finding it, and returns BEGIN as the terminal symbol name. (If it failed, the .OUT part of the description would be selected.) Of course BEGIN is internally represented by an integer number.

***Example of string*:**

Suppose for example that "'DON''T'" is given as input. The lexical analyzer scans single-quote, selects the "class" QUOTE, and jumps to "symbol" STRING. Then, the description of "symbol" STRING makes it possible to store "DON'T" in the string area, as follows. The lexical analyzer skips the first character, single-quote (by "—$"). Here, "$" specifies a one-character matching, which is made to match a single-quote in this "regular expression". The lexical analyzer continues to scan input while the input is (i) a newline which is skipped (by "—"@'" where "—" means skip and "'@" means newline), or (ii) not a single-quote (by "¬$"), or (iii) a succeeding pair of single-quotes of which the first is skipped (by "—$$"). Finally it ends scanning when the closing single-quote is found, which will be skipped (by "—$"). Now, the procedure STORE is called, which should return an integer flag 1 or 2. According to this flag, either CHARCON or STRCON is selected as terminal symbol name.

***Example of colon-sequence*:**

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

$$:=: \quad := \quad :/=: \quad :$$

Here, "/" in the description means empty, "!" not

followed by a name specifies that the token scanned until that time is itself the terminal symbol name.

Comparing with other lexical analyzer generators [10, 18], our system can be characterized in the treatment of "table"s, specification of skip ("—"), selection of terminal symbols by procedures ("!(···, ···, ···)") and automatic internal numbering of terminal symbols.

5. Multipass Partial Grammar Parsing

In this chapter, we shall present main features provided in our multipass partial grammar parsing scheme, drawing on an excerpt description of a compiler for Algol 68 subset (Fig. 4). We assume for simplicity that the compiler collects declarations at the first pass, and analyzes the rest at the second pass. In the research reported here, we have restricted ourselves to the application of SLR(1) parsing techniques in all passes, though, in general, our parsing scheme is independent of the grammar class. Detailed and formal description on our multipass parsing can be found in [5].

5.1 Starting and terminating partial grammar parsing

The partial grammar parsing of a pass is illustrated in Fig. 5. 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. The parsing is made only on partial portions

```
,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    '|D' '|'
          ,AFTER 'BEGIN'.'THEN'.'ELSE'.'OUT' .
                 'DO' . '|' . '|' . 'EXIT'
                                        !SEMLDE !LAB!        #(C)#
,SYNTAX
     DEC  ,BEFORE '|' . ')' .
     MVAR ,BEFORE ':=' . '|' . ',' . ')'
          ,RETURN ',','|'.                                   #(D')#
     CAST ,BEFORE '(',                                       #(D'')#
                    .
                    .
                    .
          ,AFTER    'BEGIN'.'IF'.'THEN'.'ELSE' .'ELIF'.'CASE'.
                    'OUSE'.'WHILE'.'DO'.'('.'|' .'|:'.'|'.')'.','.' !
                                                            #(D)#
     DEC            ->  S                                   #(E)#
     S              ->  MOP!MID!ROP!RID!RVAR!MVAR!PRI!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) *) !
                    .
                    .
                    .

,GRAMMAR 2

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

     PROGRAM    ->    .                                      #(G)#
                      .
                      .
                      .
```

Fig. 4 An example description of partial grammars for pass 1 and 2 (Algol 68 subset).

of the i-th text (shaded portion of the text in the figure) which correspond to sentences of $L(G_i)$, the language generated by G_i. The goal symbol S_i of the partial grammar G_i is output to the $(i+1)$-th text, which will be treated as a terminal symbol at the next pass. For the other portion of the text (blank portion of the text in the figure), the i-th pass parser merely copies i-th text into $(i+1)$-th text. (Note that in the case of the first pass, the parser copies the internal form of source text produced by the lexical analyzer into second text.) In order to catch the starting position(s) for i-th pass parsing, two terminal symbol sets must be given. The first is the set of terminal symbols preceding sentences of $L(G_i)$, which will be called $PREC_i$. The second is $\mathrm{FIRST}_1^{G_i}(S_i)$, the set of first terminal symbols of sentences 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 specified explicitly in the description of the syntax of each pass. The i-th pass parsing is "triggered" whenever a terminal symbol $a \in PREC_i$ is followed by a terminal symbol $b \in \mathrm{FIRST}_1^{G_i}(S_i)$, in the i-th text.

Similarly, to determine the terminating position of a partial grammar parsing, we must explicitly specify in the description the set of terminal symbols succeeding $L(G_i)$, which will be called $SUCC_i$.

In the description, these two sets are specified as

$$S_i \text{ .BEFORE } SUCC_i \text{ .AFTER } PREC_i \cdots$$

(see #(D)# in Fig. 4).

5.2 Extra goal symbol

The determination of starting position for partial grammar parsing by two adjacent input terminal symbols as mentioned above has a weak point. As an example, suppose that we want to collect declarations of an Algol 68 program at the first pass. We encounter a difficulty

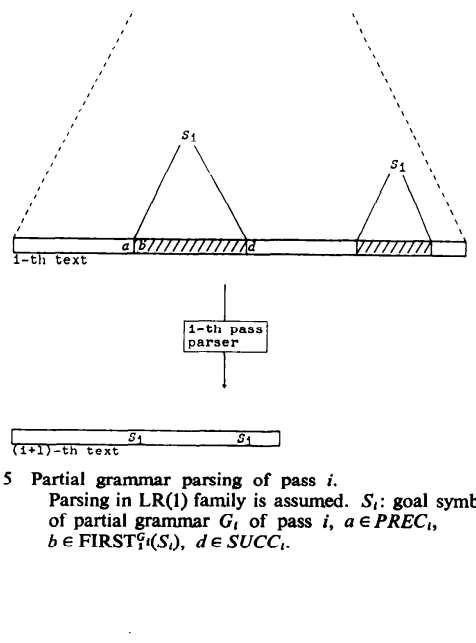

Fig. 5 Partial grammar parsing of pass i.
Parsing in LR(1) family is assumed. S_i: goal symbol of partial grammar G_i of pass i, $a \in PREC_i$, $b \in \mathrm{FIRST}_1^{G_i}(S_i)$, $d \in SUCC_i$.

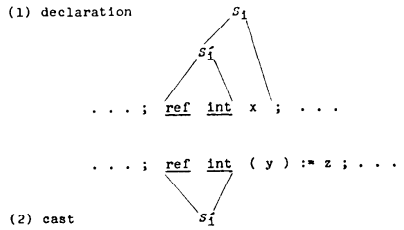

Fig. 6 Extra goal symbol.
S_i': extra goal symbol.

arising from the similarity between "declaration" and "cast" (Fig. 6). Since

$$; \in PREC_i \text{ and } \mathbf{ref} \in \mathrm{FIRST}_1^{G_1}(S_i)$$

hold, the parser inevitably enters the partial grammar parsing when catching ";" followed by "**ref**". However, in the "cast" case, this is an undesirable action. In order to escape from this difficulty, we adopt a strategy to exit from parsing when the parser catches "(", namely, as soon as the parser is aware that "cast" is being dealt with.

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

$$S_i, \underset{G_1}{\overset{*}{\Rightarrow}} S_i'\alpha$$

and a set of succeeding terminal symbols (which is a set including ")" in the above case). We have named S_i' an "extra goal symbol".

Using this feature, the above "cast" case can be specified by

S_i' .BEFORE '('

(cf. #(D")# in Fig. 4.)

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 at the first pass, the input

$$\cdots; \mathbf{int}\ i\ := n+1, j;$$

causes a problem that it contains an initialization portion ":=n+1" which we do not want to parse in this pass. For this purpose, a mechanism is supplied for saving parser state, restoring it and continuing partial grammar parsing thereafter. Schematically, it can work as follows:

$$\cdots; \mathbf{int}\ i \boxed{:=n+1} ,j;$$

save restore

Only underlined portions are parsed by the partial grammar.

Another example follows:

$$\mathbf{proc}\ p=(\mathbf{real}\ a)\ \mathbf{real} \boxed{: \cdots} ;$$

save restore

After saving parser state, the partial grammar parsing continues as usual. Thus, nested saving and restoring can be processed as shown below:

$$\mathbf{int}\ i \boxed{:=\mathbf{begin}\ \mathbf{int}\ k \boxed{:=20} , m; \cdots \mathbf{end}} , j \boxed{:=n+3} ;$$

save save restore restore save restore

An example description of this feature is

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

(cf. (D') in Fig. 4).

5.4 Replacing input terminal symbols

Recall the identification problem of variables in a procedure body ("x" and "g" at line (2) of Chapter 1). Such a problem can be treated in general in either of two ways. Namely, (i) using the same terminal symbols in the syntax for any case and distinguishing them in the semantics, or (ii) using different terminal symbols in the syntax by distinguishing them in the lexical analyzer. Method (i) requires no modification of the compiler generators, but since method (ii) conforms to multipass parsing, we have included a feature for replacing input terminal symbols by other terminal symbols in the lexical analyzer. Note that we have already presented in Chapter 4 the selection of terminal symbols in the lexical analyzer of the first pass. Here, we show a similar feature in the lexical analyzers of second to last passes which are usually system supplied.

We assume that the treatment of "$x+g$" in each pass will be

	$x + g$
(first pass)	ID+ID
(second pass)	ID+FID

Namely, in the first pass, ID is used for any entity which looks like an identifier; in the second pass, it is replaced by FID (function identifier) or ID (other identifier) in the lexical analyzer of that pass. To specify this, a description such as

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

should be included in the second pass description. The replacement is in fact performed according to the integer flag returned from the procedure IDFID which discriminates the two cases.

This feature is vital to the treatment of operators in Algol 68 where their priorities may be declared later in source text. (cf. #(F)# in Fig. 4).

5.5 Additional features for multipass partial grammar parsing

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

Catching range structures

In block structured languages, we would want to catch range structures of source text, such as

begin · · · **end**

do · · · **od**

in addition to the partial grammar parsing which, for example, collects declarations. Here, **begin** etc. are named "range opening symbols" and **end** etc. are named "range closing symbols".

However, recall that our parser can be given only one partial grammar for each pass. Since range opening and closing symbols such as "**begin**" are outside the sentence of partial grammar for collecting declarations, they are hard to catch in the same pass. Thus, we have added a mechanism to catch range structures simultaneously with the usual partial grammar parsing.

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

we can define

if · · · **then**

then · · · **else**

else · · · **fi**

as subdivisions of ranges, which are named "subranges".

The information collected in a subrange can be attached as an atrribute to the range opening and closing symbols, for utilization in later passes. For this purpose, semantic actions can be called when range opening and closing symbols are recognized (cf. SEML and SEMR at #(A)# in Fig. 4).

Catching label definitions

The treatment of label definitions in compilers has many similarities with the treatment of declarations. However, we realize again that the former is hard to handle in the same pass as the latter since label definitions are outside the sentence of partial grammar for declarations. On the other hand, 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 simultaneously with the usual partial grammar parsing. Thus, labels can be caught by specifying the two consecutive terminal symbols for label definitions (for example, ID and ":") and the set of terminal symbols preceding it. (cf. #(C)# in Fig. 4).

6. Description and Evaluation of Semantics

6.1 Modified attribute grammar

There is not yet agreement on the best formal notation for semantics. A prominent candidate is the attribute grammar introduced by Knuth [20]. It has been widely known as a powerful description method for the specification of programming languages and their compilers. However, its use may result in a dilemma, that is, semantic evaluation speed is usually very slow and its power is rather restricted or insufficient compared with real compilers which use hand-coded routines.

Another attempt at formal notation for semantics can be found in the Affix grammar used in CDL compiler-compiler [8]. It relies intrinsically on top-down recursive descent 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 Wijngaarden grammar [1].

Considering these points, we have developed a modified type of attribute grammar as the description language for syntax and semantics. In our style, evaluation by semantic attributes can be processed efficiently and may be intermixed with evaluation by programs in procedure-oriented language which use 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 parsing without actually building the parse tree. Therefore, in a strict sense, inherited attributes cannot be accepted in our system. Instead, we introduce global entities, whose values can be determined using the information collected in previous passes. The introduction of global entities generally destroys the semantic locality of attribute grammars. However, we have confirmed from experience that using global entities makes it possible not only to replace inherited attributes, but also to facilitate natural description of semantics in a representative case, as will be shown later (Sec. 6.4). Moreover, use of global entities and avoidance of parse tree construction bring a considerable amount of speed-up of semantic evaluation, because the slow processing speed in the original attribute grammar results mainly from passing of local values of attributes from node to node within a parse tree. Comparisons with other compiler generator systems will be discussed later.

An example description ("if statement" in Algol 68 subset) can be seen in Fig. 7. It may be considered as a bottom-up version of Simonet's description [21] with code generation, and is comparable in simplicity. Strings enclosed by single-quotes, for example 'IF', are terminal symbols. Strings not enclosed by single-quotes, for example IF, are nonterminal symbols. Semantic synthesized attributes are enclosed by "⟨" and "⟩" in production rules, for example DECS and VALUE. CODEFILE is a special synthesized attribute for object code.

```
,GRAMMAR 2
                    .
                    .
                    .
#(1)#   IF<DECS>              ->  'IF'<DECS>
          <* ENV +:= DECS *) ,
#(2)#   ENQUIRY_CLAUSE<VALUE,CODEFILE> -> DEC_SERIES<VALUE1,CODEFILE1>
          <* COERCESTEP := COERCE(MODE(VALUE1),BOOLEAN) !
             ,IF  COERCESTEP = 0  ,THEN  ERROR  ,FI !
             VALUE := TEMP(RESULTMODE(COERCESTEP)) !
             CODEFILE := CODEFILE1 + CPROC3(COERCESTEP,VALUE,VALUE1) *) ,
#(3)#   THEN<DECS>            ->  'THEN'<DECS>
          <* ENV +:= DECS *) ,
#(4)#   THEN_PART<VALUE,CODEFILE>   -> SERIAL_CLAUSE<VALUE,CODEFILE> ,
#(5)#   THEN_CLAUSE<VALUE,CODEFILE> -> THEN<DECS>  THEN_PART<VALUE,CODEFILE>
          <* ENV -:= DECS *) ,
#(6)#   ELIF<DECS>            ->  'ELIF'<DECS>
          <* ENV +:= DECS *) ,
#(7)#   ELSE<DECS>            ->  'ELSE'<DECS>
          <* ,SAME *) ,
#(8)#   ELSE_PART<VALUE,CODEFILE>   -> SERIAL_CLAUSE<VALUE,CODEFILE> ,
#(9A)#  ELSE_CLAUSE<VALUE,CODEFILE> -> ELSE<DECS>  ELSE_PART<VALUE,CODEFILE>
          <* ENV -:= DECS *)  !
#(9B)#    ELIF<DECS>  BOOLEAN_CHOOSER_CLAUSE<VALUE,CODEFILE>
          <* ,SAME *) ,
#(10A)# ALTERNATE_B_CLAUSE<VALUE,CODEFILE,LABEL> -> THEN_CLAUSE<VALUE,CODEFILE1>
          <* LABEL := LNUM !
             CODEFILE := CODEFILE1 + CODE(LABEL,":") *)  !
#(10B)#   THEN_CLAUSE<VALUE1,CODEFILE1>  ELSE_CLAUSE<VALUE2,CODEFILE2>
          <* COERCESTEP := COMMONTO(MODE(VALUE1),MODE(VALUE2)) !
             ,IF  COERCESTEP = 0  ,THEN  ERROR  ,FI !
             VALUE := TEMP(RESULTMODE(COERCESTEP)) !
             LABEL := LNUM !
             LABEL2 := LNUM !
             CODEFILE := CODEFILE1 + CPROC3(COERCESTEP,VALUE,VALUE1)
                       + CODE("=>",LABEL2/ LABEL,":")
                       + CODEFILE2 + CPROC3(COERCESTEP,VALUE,VALUE2)
                       + CODE(LABEL2,":") *) ,
#(11)#  BOOLEAN_CHOOSER_CLAUSE<VALUE,CODEFILE>  ->
    ENQUIRY_CLAUSE<VALUE1,CODEFILE1> ALTERNATE_B_CLAUSE<VALUE,CODEFILE2,LABEL>
          <* CODEFILE := CODEFILE1 + CODE(VALUE1,"(=01=>",LABEL) + CODEFILE2 *),
#(12)#  BOOLEAN_CHOICE_CLAUSE<VALUE,CODEFILE>  ->
          IF<DECS>  BOOLEAN_CHOOSER_CLAUSE<VALUE,CODEFILE>  'FI'
          <* ENV -:= DECS *) ,
                    .
                    .
                    .
```

Fig. 7 An example description of syntax and semantics using a modified attribute grammar ("if statement" in Algol 68 subset).

Attributes differing only in their trailing digits, for example VALUE and VALUE1, are regarded to be of the same type. User-specified semantic actions are enclosed between "⟨*" and "*⟩".

6.2 Semantic evaluation

Our system supports two types of semantic evaluation. The first type of semantic evaluation proceeds in the same manner as in the usual attribute grammar. Namely, if same (synthesized) attributes appear in both sides of a production rule, such as DECS in Fig. 7 #(1)# or VALUE and CODEFILE in Fig. 7 #(5)#, then, assignments of the right-hand side attributes into left-hand side ones take place. (The direction of assignment corresponds to bottom-up parsing.) The second type of semantic evaluation proceeds according to the user-specified semantic action enclosed between "⟨*" and "*⟩". It can be written in a procedure-oriented language similar to the one used in subordinate routines for lexical analyzer. Further, the use of macros can make description much more readable, for example,

ENV+:=DECS

which is in fact no other than a procedure call.

6.3 Implementation of semantic attributes

In practical implementation, operations on semantic attributes are converted into operations among semantic stack elements, for both types of the above semantic evaluation. This realization method together with the introduction of user-specified semantic action in procedure-oriented language can achieve compile-time semantic evaluation speed as efficient as hand-coded compilers, still preserving readability of description.

Several implementational issues exist which need precaution. However as space is limited, we point out here only the "aliasing" problem. Observe, for example, VALUE associated with ALTERNATE-B-CLAUSE (Fig. 7 #(10A)#) and VALUE1 associated with THEN-CLAUSE (Fig. 7 #(10B)#). Since we realize the attributed tree implicitly on semantic stacks, the two attributes are mapped to the same position on the semantic stack VALUESTK. The problem is that assignment to VALUE may destroy VALUE1, or vice versa. The implementation handles this and other problems properly, and further makes some optimization which are all transparent to the users.

6.4 An example of semantic evaluation

As an example of semantic evaluation, we illustrate the treatment of "active" declarations according to the description of Fig. 7. Suppose that declarations have been collected at the first pass, and that the following text is given to the second pass.

```
                                        (ENV=empty is here assumed)
  if
⟨DECS⟩.....> (declarations in (i))
              (i)                       (ENV=(i))
    then
    ⟨DECS⟩.....> (declarations in (ii))
                 (ii)                   (ENV=(i)+(ii))
    else
    ⟨DECS⟩.....> (declarations in (iii))
                 (iii)                  (ENV=(i)+(iii))
  fi
                                        (ENV=empty)
```

Here, each semantic attribute DECS has been made to possess declarations appeared in the corresponding subrange. ENV is a global entity corresponding to the "environment" (collection of active declarations at each stage) which serves as a substitute for an inherited attribute. Now, in recognizing "if", the reduction by the production rule (1) takes place, and DECS (for declarations in subrange (i)) is added to ENV. Then, if no surrounding declarations had existed, ENV may be schematically expressed as "ENV=(i)". In recognizing "then", another reduction by (3) takes place, and DECS (for (ii)) is added to ENV resulting in "ENV=(i)+(ii)". At the moment immediately before recognizing "else", DECS (for (ii)) is subtracted from ENV using the reduction by (5), and in recognizing "else", DECS (for (iii)) is added to ENV using the reduction by (7). So, "ENV=(i)+(iii)". Thus, the range structure of "if statement" in Algol 68 can be naturally described using (synthesized) attribute, global entity and subrange feature.

6.5 Comparisons to other systems

In many compiler generators, such as attribute-grammar based systems [10, 15] and transformation-

grammar based one [12], generated compilers first build a kind of attributed parse tree using the syntax analyzer and then perform semantic evaluation on the tree. Compared to our system, they tend to be slow in creating the tree and in passing attributes through the tree.

There exist some systems [13, 14] where attributed parse trees are not built by realizing them implicitly within the framework of recursive descent compiling. Their generated compilers are essentially made of a set of procedures, and can handle global entities or tables as we did in our system. In contrast to them, our system has shown that a *bottom-up* implementation of implicit attributed tree is also attractive on the foundation of multipass parsing. Our realization of attribute grammar is comparable to the above systems in its power and efficiency, with removing delicate restrictions of top-down parsing as regards grammar class.

7. Generator Organization for Efficient Partial Regeneration

We have organized the compiler generator so that the task of generating a compiler can be incremental, by making the task divisible into efficient partial (re)generation steps corresponding to each of the description units.

Figure 8 shows the modular organization of the compiler generator. The input to the compiler generator consists of description units corresponding to each compiler module. The "pragmat part" of the description controls all generation steps by specifying, for example, number of passes, compiler name, generate/not-generate indication and parameter options for each description unit. Note that OS-dependent parts of the description are wholly confined in this "pragmat part".

The "master control program" activates only necessary generation steps according to the above generate/not-generate specification. In order to provide necessary information for partial (re)generation, all interfaces between generation steps, once generated, are automatically preserved or updated in files called "interface files".

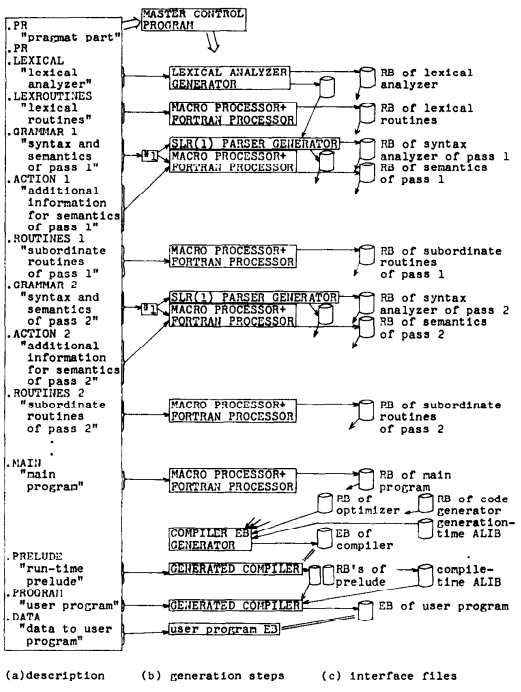

Fig. 8 Modular organization of compiler generator.
*1=modified attribute grammar handler (syntax/semantics separator+attribute to semantic stack translator), RB=relocatable binary, EB=execution binary, ALIB=automatic call library.
Inclusion of the description in files is possible using ".FILE=⟨file name and vol⟩;" or ".INCLUDE ⟨i⟩=⟨file name and vol⟩;".
Master control program activates only necessary generation steps, although it activates all generation steps in this figure.
Generated compiler is shown in Fig. 9.

8. Generated Compilers

8.1 Organization of generated compilers

Figure 9 shows an example organization of a compiler generated by our compiler generator. It consists of modules corresponding to each description unit. A natural overlay structure between each pass, optimizer and code generator is realized as a default unless otherwise specified.

8.2 Intermediate language, code generator and optimizer

Although these subjects are described elsewhere, we shall briefly present their outlines.

A machine-independent intermediate language, named IL, aiming at production of optimized codes has been designed [22]. The intermediate language to be output

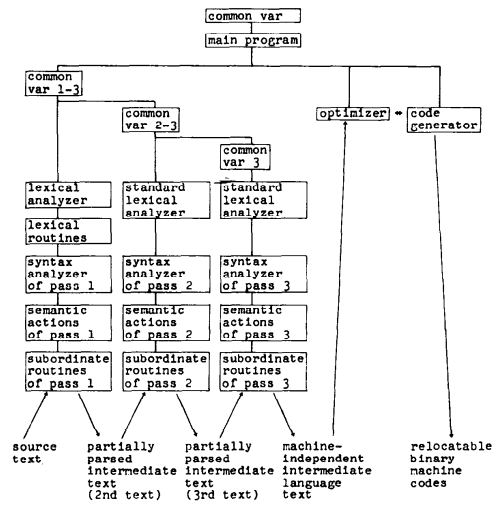

Fig. 9 Organization of a generated compiler (three-pass case).

by multipass parsing is not limited in any sense, but we support IL as a standard for user's convenience. The language level of IL is kept low enough with excluding peculiarities of particular machines. An IL statement is made of a quadruple with one operator field and three operand fields.

A code generator for a Japanese machine (FACOM 230–45S series) has been made with special care for generality and portability [23].

An optimizer is under development [24, 25]. Its input and output text are both chosen to be the intermediate language IL so that the optimizer can be included or omitted without affecting other phases.

9. Concluding Remarks

Starting from difficulties in processing inverted appearances of definitions and uses of language constructs, we introduced the concept of multipass partial grammar parsing. On this basis, we described a compiler generator which generates compilers allowing multipass parsing.

9.1 Remarks on multipass parsing

The adoption of multipass parsing not only has solved substantial problems of single-pass parsing presented in the introduction, but also has simplified compiler description which may present rather congested semantics in single-pass compilers. For example, processing of the famous "labels with block structures" in two passes has reduced its program size to about half compared with the single-pass processing where the troublesome handling of links of label is required [26].

With respect to time and space of generated compilers, one may imagine the multipass parsing to be more time-consuming compared with single-pass parsing. However, it has been shown that for a two-pass parser (not including semantic evaluation) of XPL [7], increased CPU time compared with one-pass parser is less than 5%, with about 20% of space reduced using overlay [5].

Next, we point out problems for future studies. First, we note that formalism of multipass partial grammar parsing is our original work, as far as we know. Khabbaz [27] discusses multipass precedence analysis, but his work is of theoretical interest making it impossible to apply to partial parsing such as for collecting declarations. However, our formalism seems to be of too practical nature. More strict and mathematical formalism should be further studied. Second, in the present system, the compiler-writer must describe partial grammar for each pass. 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 such description. However, note that even if it is realized, the description of semantics would still be dependent on partial grammar of each pass. Third, the relation between partial grammar and additional features (for catching range structures and label definitions) still lacks uniformity in the present system. Therefore, their unification is being investigated. Fourth, it would be convenient if the compiler-writer could 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.

9.2 Remarks on compiler generator

At the moment, we have two compilers currently being developed using our compiler generator. One is for a subset of Algol 68 [26] and the other is for a subset of Ada [28]. Although not all characteristic features of our compiler generator have been fully utilized, we believe from experience gained so far that the design philosophy was successfully realized as follows:

(1) Integrated and compact description of compilers was designed. Especially, a modified attribute grammar for syntax and semantics simplified the description, and at the same time increased compile-time efficiency compared with original attribute grammars.

(2) Compiler construction process was made flexible by the partial (re)generation feature of the compiler generator, and by the unification of compiler and user/test program in the description.

(3) Careful considerations on machine-independence are paid in our compiler generator, generated compilers and object codes. The compiler generator and generated compilers are written or generated in Fortran, mostly in an Algol 68-style macro-extended Fortran. As for object codes, a machine-independent language IL was designed. At the same time, clear confinement of machine-dependent parts is achieved in each software. Namely, in the compiler generator, OS-dependent parts, such as generation of commands in a job control language, have been confined within the "master control program"; in generated compilers, OS-dependent parts, such as overlay commands to the linkage editor, have been confined to the "pragmat part" of the description and to the "master control program" of the compiler generator. Thus, our system affords the user fine possibility of moving either his language description or his generated compiler to a new machine.

Finally, we point out some problems for future studies. First, more clear and useful description for syntax and semantics should be further investigated. It correlates to the uniformity problem between partial grammar and additional features discussed before. Second, consolidation of rich utility features will be necessary. For example, unified handling of multipass error occurrences in generated compilers will be required to make the system a truly usable software tool. Lastly, we must have more experience including transportation of the system, for further improvements in future versions.

Acknowledgments

The authors wish to thank S. Uehara, H. Tazaki, S.

Nakamura, H. Yoshida and members of the laboratory for their contributions and helpful comments to the compiler generator system.

A preliminary version of this paper first appeared in [29].

References

1. WIJNGAARDEN, A. V. et al. *Revised report on the algorithmic language ALGOL 68*, Springer-Verlag, 1976.
2. INOUE, K. et al. A generation method of multiphase-compilers, **18***th Proc. IPSJ* 302 (1977).
3. SASSA, M. et al. A framework for a multipass-compiler generator, **19***th Proc. IPSJ*, 3C-6 (1978).
4. SHINOGI, T. et al. On generation of partial grammar parsers, **19***th Proc. IPSJ*, 3C-5 (1978).
5. SHINOGI, T. Research on automatic generation of multipass parsers (in Japanese), Master thesis, Tokyo Institute of Technology, Dept. of Information Sciences (1979).
6. TOKUDA, J. et al. A multipass-compiler generator and its semantic processing, **20***th Proc. IPSJ*, 1J-6 (1979).
7. MCKEEMAN, W. M. et al. *A compiler generator*, Prentice-Hall, 1970.
8. KOSTER, C. H. A. Using the CDL compiler-compiler, in *Compiler construction- an avanced course, Lecture Notes in Computer Science* **21**, Springer-Verlag, 1974 (revised 1976).
9. LECARME, O. and BOCHMANN, G. V. A (truly) usable and portable compiler writing system, in *Information Processing* **74** (1974) 218–221.
10. LEWI, J. et al. SLS/1: A translator writing system, in *Lecture Notes in Computer Science* **34**, Springer (1975) 627–641.
11. JOHNSON, S. C. YACC-Yet another compiler-compiler, CSTR 32, Bell Laboratories (July 1975).
12. PIERCE, R. H. and ROWELL, J. A transformation-directed compiling system, *The Computer Journal* **20**, 2 (1977) 109–115.
13. BOCHMANN, G. V. and WARD, P. Compiler writing system for attribute grammars, *The Computer Journal* **21**, 2 (1978) 144–148.
14. MILTON, D. R. et al. An ALL(1) compiler generator, SIGPLAN Notices **14**, 8 (Aug. 1979) 152–157.
15. GANZINGER, H. et al. Automatic generation of optimizing multipass compilers, in *Information Processing* **77** (1977) 535–540.
16. HARTMANN, A. C. A concurrent Pascal compiler for minicomputers, *Lecture Notes in Computer Science* **50**, Springer (1977).
17. KIMURA, T. et al. Logical partition of a Pascal compiler (in Japanese), *Proc. of* **20***th Programming Symposium*, 1 (1979).
18. LESK, M. E. LEX- a lexical analyzer generator, *CSTR* **39**, Bell Laboratories (Oct. 1975).
19. SASSA, M. A pattern matching macro processor, *Software-Practice and Experience* **9**, 6 (1979) 439–456.
20. KNUTH, D. E. Semantics of context-free languages, *Mathematical Systems Theory* **2**, 2 (1968) 127–145.
21. SIMONET, M. An attribute description of a subset of Algol 68, *Proc. of the Strathclyde Algol 68 Conference*, SIGPLAN Notices **12**, 6 (June 1977) 129–137.
22. UEHARA, S. et al. Design of an intermediate language for generation of optimized code, **19***th Proc. IPSJ*, 3C-8 (1978).
23. NAKAMURA, S. Code generator with machine-independence and portability considerations (in Japanese), Bachelor thesis, Tokyo Institute of Technology, Dept. of Information Sciences (1979).
24. UEHARA, S. et al. Optimized object code generation with global register assignment, **21***th Proc. IPSJ*, 5B-8 (1980).
25. TAZAKI, H. et al. On interprocedural data flow analysis for optimizing compiler, **21***th Proc. IPSJ*, 5B-9 (1980).
26. NAGASAWA, Y. et al. A two-pass compiler for ALGOL 68 using a multipass-compiler generator (in Japanese), **20***th Proc. IPSJ*, 1J-2 (1979).
27. KHABBAZ, N. A. Multipass Precedence Analysis, *Acta Informatica* 4 (1974) 77–85.
28. YOSHIDA, H. et al. Experimental implementation of Ada translator using a compiler generation system, **21***th Proc. IPSJ*, 2B-10 (1980).
29. SASSA, M. et al. Design and implementation of a multipass-compiler generator, *Research Reports on Information Science*, C-24, Tokyo Institute of Technology, Dept. of Information Sciences (1979).

(Received March 10, 1980; revised June 17, 1980)