stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 157/81 JANUARI

G. FLORIJN & G. ROLF

PGEN - A GENERAL PURPOSE PARSER GENERATOR

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B99, 68F25

ACM-Computing Reviews category: 4.12, 4.22, 4.42

PGEN - A general purpose parser generator

by

Gert Florijn *)
Geert Rolf *)

ABSTRACT

This paper describes the use and implementation of a parser generator, that generates recursive descent parsers with automatic error recovery. The input for PGEN is a LL(1) grammar written in a powerful, BNF-like meta-language. This grammar can be augmented with user-defined actions for building parse-trees, generating code etc. The generator checks the correctness of the grammar, and generates a recursive descent parser, which contains suitable error recovery operations and error messages. The resulting parser is a SUMMER program. The user should be familiar with SUMMER, since the actions that can be associated with parts of a grammar, must be written in this language.

KEY WORDS & PHRASES: parser generator, error-recovery, SUMMER

---

*) Hogere Informatica Opleiding,
I.H.B.O. 'De Maere',
Enschede, The Netherlands.

INTRODUCTION AND BACKGROUND

The problem of implementing a parser-generator, which generates parsers with automatic error-recovery came up during the SUMMER project at the Mathematical Centre. SUMMER [1] is a string manipulation language designed and implemented by Paul Klint and Marleen Sint.

It turned out to be difficult and error-prone to update the parser of a language-in-development, while using an error-recovery method that depends on information (first-symbols) related to the complete grammar.

Hence a parser generator was needed, that generates the information for the error-recovery method automatically. The meta-language chosen is derived from the one Paul had developed for his description of SUMMER. It differs slightly from the well known Backus-Naur Form, but gives more concise grammars.

The generated parsers use an error-recovery scheme described in [2] and [3]. This method, which for instance has been used in the Pascal compiler from the ETH in Zurich (Switzerland), requires that the grammar obeys the LL(1) restrictions.

This paper is divided into two parts, a user manual and an implementation manual. The user manual describes all matters that are important when using PGEN. This part includes a description of the meta-language, a discussion of the LL(1) restrictions and a detailed illustration of the features and limitations. The second part, i.e. the implementation manual, gives an overall description of the parser-generator itself.

ACKNOWLEDGEMENT

This paper describes the result of our stay at the Mathematical Centre from August ´80 until January ´81.

We like to express our appreciation to Paul Klint and Marleen Sint for supporting our project and to Paul Verhelst and Dick Grune for testing the program. We thank Paul Klint and Dick Grune for correcting this document. Furthermore, we like to thank all the other members of the ´afdeling informatica´ (computer science department) for their cooperation and support.

Our project greatly benefited from the lessons in ´compiler construction´ by Theo de Ridder. He and Siep van der Wal, who supervised our stay, showed interest in the project for which we are grateful.

1. USER MANUAL

1.1. Description of the syntactic meta-language

1.1.1. Introduction.

A parser-generator needs a description of the syntax of the language for which a parser has to be created. Such a description can be given in different ways, but two methods are widely used. The first method gives a description by means of graphs called syntax diagrams. These diagrams give a good picture of the syntax of the language. Unfortunately, it is quite difficult to use diagrams as input for a program. Therefore a meta-language is often used to describe the syntax.

A syntax-description in a meta-language consists of terminal symbols, non-terminals and production-rules, which define the non-terminals. Sometimes special constructions can be used to describe optional or repeating parts of a syntax rule.

One of the best-known syntactic meta-languages is BNF. The meta-language used by PGEN, an extended form of BNF, was introduced in [1]. It offers some extra facilities to deal easily with repetition and nesting.

In the following section we give a syntactic description of the major part of the meta-language. Next we discuss some lexical and semantic aspects and some additions to the meta-language.

1.1.2. Syntax and semantics of the meta-language

1.1.2.1. Syntax of the meta-language

The syntax of the meta-language in its own notation is as follows.

```
<grammar>          ::= <rule>*.

<rule>             ::= <rule-name> ´::=´ <rule-body> ´.´ .

<rule-body>        ::= { <alternative> ´|´ }* .

<alternative>      ::= <primary>+ .

<primary>          ::= ( <terminal-symbol> | <rule-name>
                       | <compound> )
                       [ ´+´ | ´*´ ]
                   | <list> | <option>.

<option>           ::= ´[´ <rule-body> ´]´ .

<list>             ::= ´{´ <primary> <terminal-symbol> ´}´
                       (´+´ | ´*´).

<compound>         ::= ´(´ <rule-body> ´)´ .

<terminal-symbol>  ::= <keyword> | <string> .

<rule-name>        ::= ´<´ <identifier> ´>´ .
```

This definition describes only the basic part of the meta-language. Some extensions have been made, to deal with lexical and semantic matters. These additions will be described in sections 1.1.3 and 1.1.4. We kept them out of the description here, to avoid confusion. A complete syntax of the meta-language is given in appendix I. The next section describes the meta-language in detail.

1.1.2.2. The meaning of the syntactic constructions.

A non-terminal or rule-name is an identifier surrounded by ´<´ and ´>´. An identifier is defined as:

```
<identifier> ::= <l-c-l> ( <l-c-l> | <digit> | ´-´ )*.
```

where <l-c-l> means lower-case letter. Normally, every non-terminal has to be defined, but we will make an exception to this rule in the next section. Of all the rules in a grammar, exactly one should not be referred to by any other rule. This specific rule is called the start symbol of the grammar.

A terminal-symbol can have two different forms:

1) A <keyword>. This is a sequence of upper-case letters. This form can be used to denote keywords like IF, THEN etc. The keyword may not be INIT, EXIT or LEXICAL, because these are reserved words for PGEN.

2) A <string>. This is a nonempty sequence of characters surrounded by single quotes. The quote character within a string is denoted by writing ´´; a backslash is denoted by ´\\´. The string may not contain a newline . This construction can be used to describe constant character sequences in the object language, e.g.: ´:=´, ´==´, etc.

An <option> indicates that one of the enclosed alternatives may or may not occur. It is clear that repetition of an <option> makes no sense, and therefore this possibility has been excluded.

Before we describe the remaining forms of a <primary>, it should be noted that every <primary>, except an <option>, can be followed by a ´*´ or a ´+´. These indicators imply possible repetition of the preceding syntactical notion. Note that a list construction must be followed by an indicator.

A <primary> followed by ´+´ means that the part of the syntax defined by the <primary> can occur several times, but must occur at least once. A <primary> followed by ´*´ can occur zero or more times. For example :

<statement>*

implies zero or more occurrences of a <statement>. A <compound> can be used to group several alternatives of which exactly one must occur. It can also be used to denote a possible repetition for a sequence of constructions, for example:

(<statement> ´;´)*.

describes a sequence consisting of zero or more repetitions of a <stat> followed by ´;´.

A <list> is used to describe an occurrence of a list of <primary>s separated by <terminal-symbol>s. Its semantics can be described using a <compound>:

<primary> (<terminal-symbol> <primary>)*.

Note that this only defines the case { ... }+. When the ´*´ indicator occurs, the whole construction is optional:

[<primary> (<terminal-symbol> <primary>)*].

The <list> is merely a shorthand and one could expect that the parser generator expand the <list> into one of the two forms given above, but this is not true. The <list> is treated as a separate case, just like <compound> and <option>.

The <list> is mostly used to describe parameter-lists and similar sequences. An example of the definition of a procedure call, where zero or more parameters separated by commas can occur, is:

```
<procedure-call> ::=
    <procedure-name> '(' { <parameter> ',' }* ')'.
```

A feature which is not described in the syntax is the usage of comment in a syntax description. A comment starts and ends with a '#'.

1.1.3. Lexical considerations

The meta-language offers a lot of features to describe the syntax of a language. It is also easy to convert a description in syntax-diagrams into a definition in meta-language-constructions. But this does not mean that everything should or could be defined in the syntax-definition.

In general, it is good practice to separate the syntax definition from the definition of lexical units like identifiers and integer constants. The reasons for keeping these lexical definitions out of the syntactical definition are threefold:

1) Separating lexical and syntactical definitions leads to a parser with a clear structure.

2) Consecutive constructions in a syntax rule can, and sometimes must, be separated by layout-symbols (spaces, tabs etc.). This is not the case for the constructs occurring in a rule that describes a lexical unit. For example, when an integer-number is defined as:

   ```
   <integer-number> ::= <digit>+.
   ```

   what would the input-string "23 45" mean? Do we find two different <integer-number>s, 23 and 45, or only one <integer-number> 2345, because the layout will be skipped?

3) Recognition of lexical units can be faster, when done by a dedicated procedure.

A feature was added to the meta-language to allow the user to separate the lexical and syntactic definitions. Before the start of the syntax-description the parser generator can be instructed which terminals will be used as lexical symbols, but are not defined. The syntax of this clause is :

<lexicals> ::= 'LEXICAL' { <identifier> ',' }+ '.'

Such a "lexical identifier" can be used in the same way as the name of a rule (i.e. surrounded by '<' and '>'), but it may not be redefined. Lexical identifiers are, in fact, terminal symbols, and may be used as such. This means that they can occur as separator in a <list>.

Only the types of the listed lexical identifiers are known to the generated parser. The (user-defined) lexical scanner has to find these lexical symbols in the input and return their types to parser.

Section 1.4 describes how a user-defined lexical scanner can be constructed.

1.1.4. Semantic considerations

A parser generator must provide some means to describe actions which have to be performed when parts of a syntax-rule have been recognized in a source-language program. These actions must be put on the right places in the generated parser, so that they are performed at the right time. To achieve this, the user must be able to indicate the actions in the grammar of the source-language.

The simplest method is to allow the user to put program pieces within the syntactic rules. This scheme is used by YACC[4]. This solution has the unpleasant consequence that it makes the grammar quite unreadable. It is much nicer to indicate only where an action has to be performed and to specify the action after the rule has been completely defined.

However, not all problems can be solved with this scheme. Most of these problems have to do with communication between parsing procedures. It may be essential to get information from a procedure which handles another syntax-rule. Only a return-value mechanism is provided.

To solve the communication problem we developed the following scheme:

1) Every rule-name occurring in the right part of a production-rule can be preceded by a tag, which is defined as:

   <tag> ::= <identifier> ':'.

   This tag specifies the name of a variable to which the return-value of the procedure associated with the rule-name is assigned. Every tag in a production rule corresponds with a local variable in the procedure which handles the currently defined rule. The declaration of the variables is automatically provided by the parser generator. Because the tag is used to deal with return values, it is not

allowed to put a tag before a lexical-id. How a procedure can return a value will be described later on.

2) Semantic actions which have to be performed when a certain part of a syntax-rule occurs can be indicated by placing a label after that part. These labels must be defined when the syntax-rule is completed (see 3). The labels are defined as follows:

```
<label> ::= '/' <identifier> '/'.
```

and may occur almost everywhere in the syntax-rule. There are two restrictions:

a) Two labels may not occur after each other within the same alternative; they must be separated by at least one <primary>.

b) An empty <rule-body>, whether it is a nested one or not, may not contain a label.

The same label may be used at various points in the syntax-rule.

3) The definition of a syntax-rule can be followed by a definition of the labels used in it. When a label is not specified, no action will be inserted at the point where the label occurs. Labels may be redefined, but only the last definition is used. The specification of the labels is defined as follows:

```
<label-specification> ::=
                [ 'INIT' ':' <program-piece> ]
                ( <label> ':' <program-piece> ) *
                [ 'EXIT' ':' <program-piece> ] .
```

A <program-piece> is a series of SUMMER-statements. This code is not checked for syntactical correctness. The init-part will be executed at the start of the procedure which handles the current production-rule. It can be used for initialization and declarations. The exit-part will be executed when the procedure has completed. It can be used to perform a return statement .

These tools are discussed and illustrated later on. Two questions remain to be answered. First, how about return-values? Second, at what moment are semantic actions associated with the labels performed by the generated parser?

The first question can be answered quite easily, since the user himself has to take care of returning values. We have considered some default return-values, like the matched input or a flag indicating whether errors occurred while executing the procedure, but we decided to leave this matter to the user. The exit-part in the label-specification can be used to specify a return-statement that returns an appropriate

value.

The moment that the actions are executed is also quite simply defined. Every action will be performed after the preceding <primary> has been recognized. If an action has to be performed at the start of an <alternative> (and no preceding <primary> exists), then the action will be performed when it is sure that this <alternative> has to be entered, but before recognition of the first <primary> is attempted.

The only exception is a <terminal-symbol>. It was decided that the action associated with a <terminal-symbol> (or lexical-id) will is performed immediately after the symbol is found, and before the next symbol is read. This implies that the action will not be performed if the terminal-symbol is not found.

The current input symbol is available as value of the global variable "sy". The following example illustrates this. We want a program which reads a list of identifiers separated by commas followed by ';', and prints the number of identifiers and their names. The following grammar defines this program, assuming that a scanning-procedure which can recognise an identifier (see 1.4) is available:

```
LEXICAL identifier.

<id-list> ::= { <identifier> /il/ ','}+ ';'.

INIT :  var count := 0;   # counts the identifiers #
        put(' The identifiers are :\n');

/il/ :  put( sy,'\n');
        count := count + 1;

EXIT :  put(' Number of identifiers : ',count,'\n');
```

1.2. The LL(1) Restrictions.

1.2.1. The general definition

As we have mentioned in the introduction, the structure of the generated parsers demands that the syntax of the language satisfies the LL(1)-restrictions. These (two) restrictions imply that a parser can determine its choice between alternatives in the syntax by means of the next input symbol. LL(1) grammars are a special form of LL(k) grammars, which need k input symbol to distinguish alternatives. This section of the LL(1) property starts with a general definition which is based on Wirth [3] and de Ridder [5]. As we shall see, this definition is not very well suited for the meta-language used by PGEN. First, the general definition is given. Next it is adjusted to the meta-language.

First, the two auxiliary functions ´first´ and ´follow´ are defined:

- first(v).
  This is the set of all terminal-symbols that can appear in the first position of a sentence derived from v. For example, when <a> is defined as

  <a> ::= A <c> B .

  then first(<a>) = { A } . When <a> is defined as

  <a> ::= A | B | C .

  then first(<a>) = { A,B,C }.

- follow(v).
  This is the set of all terminal symbols which can follow, directly or indirectly, any derivation of v in a grammar. For example, when <a> and <b> are defined as

  <a> ::= A <b> C .
  <b> ::= B .

  then follow(<b>) = { C }.

Now if for every rule in a grammar the following conditions hold, then the grammar is of type LL(1).

1) For every sequence of alternatives, e.g.

   alt 1 | alt 2 | ... | alt n.

   the following relation must hold.

   $$\text{first(alt i)} \cap \text{first(alt j)} = \emptyset \ , \ (i \neq j)$$

   This means that two alternatives may not start with the same symbol, like in

   <a> ::= A | A ´;´ .

2) For a construction C that can produce the empty sentence, the set of first-symbols must be disjoint from the set of follow-symbols, e.g.

   $$\text{first(C)} \cap \text{follow(C)} = \emptyset \ .$$

This restriction is violated by <b> in the following grammar.

```
<a> ::= <b> ';' .
<b> ::= [ ';' A ] .
```

A consequence of the combination of these two restrictions is that within a sequence of alternatives, at most one alternative may produce the empty sentence.

These restrictions exclude left-recursive grammars. Consider for instance the production:

```
<b> ::= B | <b> B .
```

This rule is invalidated by restriction 1, because

first(B) ∩ first(<b> B) = { B } ≠ ∅ .

If we attempt to rewrite it to

```
<b> ::= [ <b> B ] .
```

restriction 2 is violated, because

first(<b>) ∩ follow(<b>) = { B } ≠ ∅ .

1.2.2. Restrictions related to the meta-language

This general definition of the restrictions can also be applied to the meta-language used by PGEN, but then it becomes clear that the definition does not cover all the constructions allowed by the meta-language. Before we introduce a redefinition, we give some examples that will illustrate the trouble-spots in the meta-language.

If <a> is defined as

```
<a> ::= B | C ( '&' | '&' F ).
```

then restriction 1 is violated by the compound construction.

Restriction 2 causes a bit more trouble. Consider the following grammar:

```
<a> ::= A ( <b> D <c> )+ .
<b> ::= ';' E .
<c> ::= ( ';' C )+ .
```

As you may have guessed, this grammar is not LL(1). But how can we see

this? None of the given productions can produce the empty sentence and there seems to be no problem. However, this grammar can produce the following string:

A;ED;C; .....

Is the last semicolon the start of a new occurrence of <c>, or does it imply a new occurrence of "(<b> D <c>)+" ?

The conflict is immediately clear when we rewrite <c> as:

<c> ::= ';' C (';' C)* .

Now we see that <c> contains a construction which can produce the empty sentence, namely (';' C)* . If we try to determine the follow-symbols of this construction, the follow-symbols of <c> are needed. From rule <a> it can be seen that

follow(<c>) = first(<b>) = { ';' }.

and now the violation can be exposed.

A far more simple example is the following one.

<a> ::= { A ',' }+ ',' .

The terminal-symbol ',' can indicate a new occurrence of the keyword A, but it can also mean that the list is terminated. But how can we prove the violation of the LL(1) restrictions, using first- and follow-sets? Again, the solution can be obtained by rewriting a part of the production. This leads to:

<a> ::= A (',' A)* ','.

which is clearly a violation of the second restriction.

The rewriting trick could be applied to every construction, but the chance of not discovering a violation is rather big. Therefore, the restrictions will now be redefined in such a way that they can easily be applied to the meta-language.

1.2.3. Revised definition.

Since restriction 2 causes most troubles, we will concentrate on the redefinition of that restriction, and it must be determined to which construction in the meta-language it applies. First, every construction must be checked that can produce the empty sentence. This amounts to every option, and all constructions followed by a '*'. Furthermore every construction followed by a '+' should be checked, as we have seen in the previous examples. To deal with problems of the kind that were mentioned

in the second example above, we establish the convention that the terminal-symbol in a list construction can produce the empty sentence, and must therefore be checked.

A second aspect which needs redefinition is the determination of the follow-symbols. A first approach leads to:

follow(C) = first(RN(C)).

where C represents a construction whose follow-symbols must be determined and RN(C) (right neighbor) represents the construction which immediately follows C in the production-rule. For example in

<a> ::= <c>+ ´(´ .

RN(<c>+) = ´(´.

This approach must be refined on the following points.

1) RN(C) can produce the empty sequence. This implies that we must continue visiting right-neighbors, add their first-set to the follow-set of C, and stop when a construction is encountered which cannot produce the empty sentence.

2) C is the last construction within an alternative, e.g.

<a> ::= (A [´(´] | F) X .

where C is "[´(´]". In this case RN(C) is defined as the right-neighbor of the enclosing construction. Hence, RN([´(´]) = RN((A ....)), which is the terminal-symbol X. Of course if we have defined <a> as

<a> ::= [´(´].

then the follow-set of "[´(´]" equals the follow-set of <a>. This set is the union of the follow-sets of each reference to <a> in other (or the same) productions.

3) Let <a> be defined as

<a> ::= (´;´ C [´;´])+ VV.

If we want to determine the follow-set of "[´;´]", we must add the first-symbols of the enclosing construction to the follow-set. Otherwise we wouldn´t be able to show the ambiguity in rules like this. Of course this is only done when the enclosing construction is followed by a repetition indicator (´+´ or ´*´). There is one exception to this rule, related to the list construction. As noted before, the terminal-symbol in a list is considered to produce the

empty sentence. This means that the follow-set must be determined. To prevent that definitions like

<a> ::= .... { ´,´ ´,´ }+ ..... .

result in a violation, the first-symbols of the entire list construction are not added to the follow-set of the separating terminal-symbol.

These considerations lead to:

```
follow(C) = if RN(C) can produce the empty sentence
            then  first(RN(C)) U follow(RN(C))
            else  first(RN(C))
            fi


RN(C) = if C is last construction in alternative
        then    D := enclosing structure of C
                if D was followed by ´+´ or ´*´
                then    add first(D) to follow-set
                        of C.
                fi
                return (RN(D))
        else    return(construction on the right of C)
        fi
```

Of course RN must satisfy the properties mentioned in 2) and 3) above. Using the functions follow, first and RN, we can redefine the restrictions as follows:

- Restriction 1

  For every sequence of alternatives, e.g.

  alt 1 | alt 2 | .... | alt n.

  the following relation must be true.

  $$\text{first(alt i)} \cap \text{first(alt j)} = \emptyset \quad . \quad (i \neq j)$$

- Restriction 2

  For every construction C that can produce the empty sentence or is followed by a ´+´,

$$first(C) \cap follow(C) = \emptyset .$$

must be true.

1.3. Working with PGEN.

1.3.1. Useful variables and procedures

The generated parsers use a few global variables and procedures that can be used in user-defined actions in the grammar. The variable sy contains the current input-symbol. However, some caution is needed in using it. The only way to get a value from sy that makes sense, is by adding a semantic action to the grammar that will be executed when a specific terminal-symbol was found in the input. This is illustrated in the following example:

```
<greeting> ::= ( HI /a1/ | BYE /a1/ )  JOE .

INIT:   var word;

/a1/:   word := sy;
```

In the action associated with /a1/, sy has the value that you expect it to have, i.e. 'hi' or 'bye'. When none of these symbols is found, the action is not performed.

As mentioned in section 1.1.3, not all terminal-symbols have to be fixed, but a LEXICAL-clause can be used to define classes of lexical symbols that are treated as normal terminal symbols. For these cases, the availability of sy can be essential. For example:

```
LEXICAL name,number.

<group> ::= <person>+ .
<person>::= NAME <name> /a1/ PHONE <number> /a2/ .

INIT:   var pname,pnumber;

/a1/:   pname := sy;

/a2/:   pnumber := sy;

EXIT:   if pname ~= undefined & pnumber ~= undefined
        then    put(' call ',pname,' on number : ',pnumber,'\n');
        fi;
```

It is important to note that sy should be used exclusively in actions associated with a terminal-symbol. If it is used elsewhere, the value is certainly not the one you would expect. The reason for this is that

actions associated with terminal-symbols are performed after the string is matched, but before the next symbol is read in. Actions associated with other constructions are performed when the part of the syntax described by the construction has been recognized in the input, and when the next input symbol has already been read in. Such a wrong usage is illustrated in the following example:

```
<trash> ::= ( HI | BYE ) /a1/ LINDA.

INIT:   var word;

/a1/:   word := sy;
```

Now word does not have the value 'hi' or 'bye', as the author of this little grammar thought.

Another useful variable is lnr which always contains the current line number. This variable can for example be used in error-messages. Two standard procedures are available for the production of error-messages. They are called ermsg and error and should be called like this:

```
error( message , lnr);
ermsg( message , lnr);
```

Message is an error-message and lnr is the number of the line on which the error occurred. Both procedures write their output on standard-output. The difference between the two routines is that error appends the word "expected" to the message and ermsg simply prints the message. The message need not contain a newline-character, because it is added by the error-procedures.

The global variable errcnt indicates the number of errors found in the input until now. Both ermsg and error increment errcnt. This variable is useful to prevent actions from being performed when errors have occurred.

The generated parser simply executes the actions, and is not aware of the fact that some actions might depend on the result of other actions. The user has to check whether a certain action can be executed or not. Consider this simple example:

```
<person> ::= ( JOE /a1/ | MICHAEL /a1/ ) BROWN.

INIT:   var firstname;

/a1/:   firstname := sy;

EXIT:   put(' The name was :',firstname,' brown\n');
```

If the generated parser for this language would be fed with the input: "tom jones", then the statement mentioned in the exit-clause would cause a run-time error, since the value of firstname is not defined.

It is obvious that these situations can occur in many places, and therefore it is good practice not to rely too much on the availability of information supplied by other parts of the syntax-rule.

In appendix II all names of variables and constants used by the generated parsers are listed.

1.3.2. Contents of the actions

As mentioned before, the actions in the grammar are not checked for syntactical correctness. This means that it is not sure whether the generated parser will be accepted by the SUMMER-compiler. Of course, PGEN does not let you down completely, since all actions in the generated parser are prefixed with a comment containing the line number of the action in the syntax-file.

Even when the syntax of the actions is correct, it is not certain that the generated parser is a correct SUMMER-program. Problems can occur when variables are declared within the actions. Because SUMMER allows declarations at the beginning of a block, declarations in actions may only occur in the following cases:

1) The action is contained in an INIT-clause. This action will be inserted at the beginning of the procedure that is generated for the whole production rule. The variables declared in this action may be used within every action contained in this production rule.

2) The action is associated with a terminal-symbol. Declarations are allowed, but the declared variable may only be used within this action.

3) The action is associated with an alternative. These declarations can be used for variables that are needed within the actions associated with the syntactical constructions mentioned in the alternative.

It is always possible to declare variables by surrounding the action with parenthesis. This transforms the action in a block, and declarations are possible.

The declaration of global variables and procedures can be achieved by placing a dedicated file with the suffix ´.ud´. When PGEN is called with the ´-u´ option, it inserts that file in the generated parser just after the global declarations made by PGEN (See appendix III). Again we refer to appendix II, where all variable-names used by the generated parsers are listed.

1.3.3. Return values

PGEN offers a means to create one-way communication between the procedures that corresponds to the production-rules. This is examplified by:

```
<hello> ::= HI who : <name> .

<name>  ::= COR /a1/ | LINDA /a1/.

INIT:   var person;

/a1/:   person := sy;

EXIT:   return(person);
```

The declaration of the variable 'who' is generated by PGEN. This variable can be used in the whole procedure-body, and initially its value is undefined.

In this example, a return-expression is part of the EXIT-clause. This is the right place to use a return expression, since writing it anywhere else might disturb the parsing-process. Note that the procedures do not have a default return value and that a return expression must occur in rules that are supposed to return a value, i.e. rules that are preceded by a <tag> in some rule.

1.3.4. Argument passing

All the production rules of a grammar are transformed into procedures that parse the piece of the syntax defined by the production. The production-rule for the start symbol of the grammar however, is transformed to a program-declaration. This program-declaration has an argument which allows the generated parser to communicate with the outside-world. The argument is called args and can be used to set flags in the parser, or to get the name of the input file. As usual in SUMMER, args is an array of strings corresponding to the arguments of the program. For example:

```
<start> ::= <program> '.' .

INIT:   if args.size = 0
        then    put(' No file name\n');
                stop(1);
        elif    infile := file (args[0],'r') fails
        then    put(' Cannot open : '||args[0]||'\n');
                stop(1);
        fi;
```

This example first checks whether any arguments were given, and then

checks whether the file that is passed as an argument can be opened for read-access. The variable 'infile' is assumed to be declared in the '.ud' file and to be known to the lexical routines.

The generated parsers also return an exit status. When errcnt is larger than zero, the SUMMER-statement stop(1) is performed. This exit-status can be used by another process that controls the parser.

1.4. Lexical aspects: a scanner for the generated parser

PGEN assumes the existence of a lexical scanner. Although there is a default scanner available, you will be forced to write one yourself in most cases. The generated parser has only a small interface with the scanner and no detailed knowledge about the working of the parser is needed to create a lexical scanner. This section discusses the interface between the parser and the scanner, and the tasks of the scanning-routine. A brief description is given of the default scanner and the conditions it needs to function properly.

1.4.1. The parser-scanner interface

The scanner for parsers generated by PGEN must be called nextsym without parameters. Its purpose is to recognise the lexical symbols in the input. Two variables are used to pass these to the parser. The variable sy gets the value of the symbol. The variable t sy contains the type-value of the symbol, stored in sy. This type-value is a unique number, used by the parser.

PGEN defines a type value for every terminal symbol in the grammar. This information is included in the generated parser, and can be used by the nextsym-routine. It depends on the type of the terminal-symbol how the type-value can be obtained. As explained in 1.1 there are three cases:

1) The input symbol was defined as a keyword. PGEN creates a table, called keytab, which contains all symbols of the grammar that were defined as keywords. The keywords are mapped to lower-case. Thus, when the grammar contained a keyword IF, the associated type-value can be obtained by

```
t_sy := keytab['if'];
```

2) The input symbol was defined as a literal constant in the grammar. The character constants from the grammar are stored in the table kartab. So, if the grammar contained something like ':=', then the scanner can access the type-value via

```
t_sy := kartab[':='];
```

3) The input symbol was defined in a LEXICAL-clause. The names that were mentioned in the LEXICAL-clause, are entered in the table predef. The associated type-value can be found by indexing this table with the name of the lexical symbol. So when when the grammar contained:

```
LEXICAL identifier.
```

and sy contains the identifier, then the type can be obtained by writing

```
t_sy := predef['identifier'].
```

Note that hyphens in lexical names are changed to underscores, so

```
LEXICAL string-const.
```

leads to

```
predef['string_const'].
```

Predef also contains the type-value that must be returned when the end of the input is reached by the scanner. This type-value can be found in the predef-table by:

```
t_sy := predef['EOF'];
```

The type-value is used by the parser to identify the symbol it is dealing with. The type-value of a symbol must be unique, meaning that a symbol may not have different meanings at different places. It is obvious that lexical ambiguities should be removed from the grammar, to prevent a lot of trouble.

PGEN defines a few (string) constants in the generated parser that may be used by the scanner. The names and their meanings are listed below.

- lower	All lower case letters
- upper	All upper case letters
- digit	All digits
- ASCII	All ascii-characters in ascending order.

Furthermore PGEN declares the variable line that may be used to read lines from the input file.

The tasks of the scanner can be formulated as follows:

1) Find in the input lexical symbols as defined in the grammar. Store the value of the symbol in the variable sy, and store the type in t_sy,

2) Skip layout symbols and comment.

3) Read new lines, if necessary, and increment the variable lnr.

4) When the end of the input is reached, sy must be set to 'EOF' and t_sy must be set to the corresponding type-value.

PGEN can be instructed to use a hand-made scanner by calling it with the '-n' option (See appendix III).

1.4.2. The default scanner.

If PGEN is not called with the '-n' option, then the scanner on the file pglib.ns in the PGEN directory is used. This scanner is capable of handling all the symbols defined in the grammar. The scanner is based on the SUMMER-definitions for strings, numbers and identifiers. It can be used for testing parsers, but not for production usage, since the generality of this scanner causes some inefficiency.

Apart from literal constants and keywords in the grammar, the standard nextsym recognises four classes of lexical symbols.

string_const — The string_const is a string of any characters opened and closed by a single quote. If a ' should appear in the string, a '' should be used. The string may not cross the end of the line.

ident — An identifier must start with a lower or upper case letter and may continue with zero or more underscores, letters or digits.

simple_integer — A simple_integer is defined as a sequence of digits.

simple_real — A simple_real is a simple integer containing a dot '.' somewhere. It may be followed by an 'e' and a simple_integer.

These lexical symbols have to be declared in the grammar. The "declaration" looks like:

LEXICAL string-const, ident, simple-integer, simple-real.

Other lexical symbols may not occur in the LEXICAL-clause, unless you write your own scanner. It is also important to list all of the symbols

that the scanner can recognise, since otherwise some of the symbol-types returned by the scanner will not be known to the parser.

The global operations performed by the default scanner procedures are described first, followed by a complete listing of those procedures.

After skipping blanks and tabs, nextsym tries to read one symbol. Five different cases exist:

1 The character read was a lower- or upper-case letter. Nextsym tries to read as many characters as possible, without violating the definition of an identifier. If this identifier does not occur in the table ´symtab´, it is assumed to be an identifier and the identifier_type is returned as value of t_sy. If the identifier did occur in symtab, it is a keyword defined in the grammar and its type is returned.

2 If the character is not a letter, then kartab is searched in reverse alphabetic order for character_strings that might appear at the cursor position. This reverse order is necessary, because the longest string should be preferred over smaller ones. If, for instance, ´:=´ occurs in the input and the symbols ´:´, ´:=´ and ´=´ are contained in kartab, then a ´:=´ should be read instead of ´:´ followed by ´=´.

3 If the character is a single quote, the procedure get_str is invoked and a string constant is returned.

4 If the character is a digit, the procedure get_number is called which tries to read a simple_integer or a simple_real. Get_number will read as many digits as possible and return the type of the symbol.

5 The end of the input line is reached. Now a new line is read in from standard-input, and the process is started again, unless end of file is reached, and sy and t_sy are set to the correct values.

When the grammar doesn´t contain keywords or literal-constants, this scanner falls short, because keytab and kartab will not declared. These deficiences restrict the usefulness of the default scanners and in most cases a user-defined scanner will have to be provided.

A complete listing of the procedure "nextsym" and related procedures follows:

```
const TRUE := 1,
      OKE := 1;

var   karar := kartab.index,
      karsize := kartab.size;

proc get_str()
(  const quote := ´´´´;

  scan line
  for   var s := cursor-1;

    while break(quote) & lit(quote || quote) do od;
    if ~lit(quote)
    then        ermsg(´newline not allowed in string´, lnr);
                rtab(0)
    fi;
    if errcnt = 0
    then        s := cursor-s; sy := move(-s);  move(s)
    else        sy := ´´
    fi
  rof
);


proc get_number()
assert  scan line
  for   [sy, t_sy] := [sy || (span(digit) | empty),
                predef[´simple_integer´]] &
        if try sy := sy || lit(´.´) || span(digit) yrt
        then    t_sy := predef[´simple_real´]
        fi &
        if sy := sy || lit(´e´) ||
            (lit(´+´) | lit(´-´) | ´´) || span(digit)
        then t_sy := predef[´simple_real´]
        fi
  rof;

proc nextsym()
(
  var i;

  while TRUE do
        line.span(´ \t´)|OKE;
    if sy := line.any(upper || lower)
    then    (sy := sy||line.span(lower||$
                upper||digit||´_´))|OKE;
                t_sy := if keytab[sy] ~= undefined
                        then keytab[sy];
```

```
                        else predef[´ident´]
                        fi;

                return;
        fi;

        for i in interval(karsize - 1, 0, -1)
        do
                sy := karar[i];
                if line.lit(sy) succeeds
                then
                        t_sy := kartab[sy];
                        return;
                fi;
        od;

        if line.lit(´´´´)
        then
                get_str();
                t_sy := predef[´string_const´];
                return;
        elif sy := line.any(digit)
        then
                get_number;
                return;
        elif line.rpos(0)
        then
                line := scan_string(get()) & lnr := lnr + 1 |
                ( sy:=´EOF´ & t_sy := predef[´EOF´] & return;)
        else
                sy := line.move(1);
                ermsg(´illegal character: ´ || sy,lnr);
        fi;
  od;
);
```

1.5. Error messages from the parser generator

Several errors can occur while generating a parser. First, the syntax of the meta-language may be violated and PGEN will complain about this.

Second, the grammar must contain just one start symbol. PGEN checks that there is just one rule that is not called elsewhere. For instance, the grammar:

```
<a> ::= ´&´.
<b> ::= ´%´.
```

causes the error message:

```
More than one start symbol found.
Start symbols:  <a> <b>
```

Third, a left recursion may be discovered in the grammar. An error message with a list of involved rules will be printed. For instance:

```
<s> ::= <a>.
<a> ::= <a> '&'.
```

is a simple case of left-recursion. It causes the error message:

```
First sets of the following rules cannot be determined
** a **
** s **
```

The left recursion causes trouble for PGEN when it tries to determine the first-sets. Note that the first-set of rule <a> depends on itself.

Fourth, a rule may be redefined or simply undefined. Both cases cause an error message.

Fifth, an LL(1) error may occur. As explained in chapter 1.2, there are two LL(1) restrictions. Restriction 1 says that it must be possible to distinguish two alternatives by their first-symbols. Restriction 2 says that an option, or any optional construction in general, and the succeeding constructions must be distinguishable by their first-sets. In both cases the first-sets of the involved constructions must be disjoint.

A few examples will show some typical violations of the LL(1) restrictions and PGEN's diagnosis. In the rule

```
<a> ::= '%'* KEY.
```

the first-set for the whole rule is { '%', KEY }, since the '%'* construction may be empty. This means that the production of rule <a> may start with a '%' or a KEY symbol. As the '%'* can be empty its follow-set has to be calculated. This turns out to be { KEY }. In the next rule:

```
<a> ::= '%'* '+' | '@'* '%' | '@' .
```

a two-fold restriction 1 error occurs. The first and second alternative violate by the symbol '%', the second and third by the '@'. The rule

```
<a> ::= '%'* '%'.
```

obviously violates restriction 2, but it can be easily rewritten to:

```
<a> ::= '%'+.
```

A less trivial example is:

```
<s> ::= <a> '%'.
<a> ::= ('%' <a>)*.
```

which causes the following error messages:

```
Restriction 2, rule : a
In : <a>
Symbols : %
Restriction 2, rule : a
In : ( .. )*
Symbols : %
Restriction 2, rule : s
In : <a> '%'
Symbols : %
```

Here two errors occurred. It is obvious that rule <a> may be empty. Hence, the '%' character is part of the first-set of rule <a>. As the ('%' <a>)* construction may repeat, the second restriction is violated by the fact that the '%' in the compound can be followed by a '%' of the nonterminal <a> or by a '%' of the repeating compound itself. The second error is caused by the fact that the whole compound may be empty and its follow-symbols are determined by the follow-symbols of the non-terminal <a> in the compound, in fact a '%'.

The following example is a more or less hidden case of a restriction 2 error

```
<a> ::= <c> '@'.
<b> ::= '@'*.
<c> ::= <b> '&'*.
```

Two restriction 2 errors occur:

```
Restriction 2, rule : a
In : <c> '@'
Symbols : @
Restriction 2, rule : b
In : '@'*
Symbols : @
```

Note that rule <c> is empty, because <b> is empty. In rule <a> restriction 2 is violated, because '@' is a first-symbol of the rule <c> and <c> might be followed by a '@' in rule <a>. The second error has the same hidden cause. Rule <b> is empty, so its follow-symbols have to be determined. As <b> is called in <c> and <c> is empty the follow-symbols

of <c> are involved in the follow-set of <b>. Finally, the '@' in rule <a> causes this same error.

Here we see that the parser generator finds the same error twice, by looking from different points.

1.6. Example of PGEN usage

In this section we give an example of the usage of PGEN: a translator for a very simple language, henceforth called SL. The SL-compiler has to translate to an assembly language for a virtual machine (VIRMA)[5].

An SL-program consist of variable declarations followed by statements. There are only two kinds of statements in SL: assignment and conditional statement. SL only knows integers.

A SL-program is converted directly to an assembler-program for a stack-oriented virtual machine.

The parser uses a few global variables, that are declared in the '.ud' file:

- infile : The input-file.
- outfile : The output-file.
- label_cnt : Used to create unique label-names.
- symboltab : Used to check undeclared variables.

Here follows a list of the grammar and the actions, then the '.ud' file, and afterwards the nextsym-routine for this language.

```
# contents of the example.ud file       #

var label_cnt := 0,
    symboltab := table(5,undefined),
    infile,outfile;
```

```
        # contents of the example.syn file      #

LEXICAL         ident,integer.

<program> ::= [ <declarations> ] <statement-list>  .

INIT:   # Try to open the input and outputfile  #
        if args[0].size = 0
        then    put(´ No arguments.\n´); stop(1)
        elif infile := file(args[0],´r´) fails
        then    put(´ Cannot open ´,args[0],´.\n´); stop(1)
        elif outfile := file(args[0]||´.imc´,´r´) fails
        then    put(´ Cannot open intermediate file.\n´); stop(1)
        fi
EXIT :  outfile.put(´\tEXIT\n´);

<declarations> ::=  VAR  { <ident> /place/ ´,´ }+  ´;´.

/place/: symboltab[sy] := 1;

<statement-list> ::=  ( <statement> ´;´ )+.

<statement> ::= <if-statement> | <assignment> .

<if>    ::= IF <expression>
            THEN  /a1/  <statement-list>
            [ ELSE  /a2/ <statement-list> ]
            FI.

INIT :  # Generate a unique labelname.$
          This labelname is used to jump over the THEN-branch.
        #
        var l1 := lab_cnt,l2;
        lab_cnt := lab_cnt + 1;
/a1/ :  outfile.put(´\tJZER LAB´,l1,´\n´);
        # jump over then branch when top of stack is false #
/a2/ :  # Generate a jump over the else branch.   #$
        l2 := lab_cnt;
        lab_cnt := lab_cnt + 1;
        imc.put(´\tJMP LAB´,l2,´\n´);
        imc.put(´LAB´,l1,´ :\n´);
EXIT :  if l2 = undefined
        then    imc.put(´LAB´,l1,´ :\n´)
        else    imc.put(´LAB´,l2,´ :\n´)
        fi;

<assignment>  ::= <ident> /check/ ´:=´ <expression> .

INIT :  var idname;
```

```
/check/:if symboltab[sy] = undefined
        then    ermsg(sy||' undeclared.',lnr)
        else    idname := sy;
        fi;
EXIT :  if idname ~= undefined
        then    outfile.put('   POP ',idname,'0);
        fi;

<expression>  ::= <term> ( adop: <adding-operator>  <term> /add/ )*.

/add/ : if adop ~= undefined
        then    outfile.put('\t',adop,'\n')
        fi;

<term>  ::= <factor>  ( muop: <multiplying-operator>  <factor>  /mul/ )*.

/mul/ : if muop ~= undefined
        then    outfile.put('\t',muop,'\n')
        fi;

<factor>  ::= '(' <expression> ')' |  <variable> .

<variable>  ::=  <ident> /check/ |  <integer> /pushc/ .

/check/:if symboltab[sy] ~= undefined
        then    outfile.put('\tPUSH ',sy,'\n')
        else    ermsg(sy||' undeclared.',lnr)
        fi;
/pushc/:outfile.put('\tPUSHC ',sy,'\n');

<adding-operator> ::= '+' /add/ | '-' /sub/ .

INIT :  var sign;
/add/:  sign := 'ADD';
/sub/:  sign := 'SUB';
EXIT :  return(sign);

<multiplying-operator> ::= '*' /mul/ | '/' /div/ .

INIT :  var sign;
/mul/:  sign := 'MUL';
/div/:  sign := 'DIV';
EXIT :  return(sign);
```

```
        # The scanner for the SL-language. It resides
          on the file example.ns
        #

proc nextsym()
(
  const TRUE := 1;

  while TRUE
  do
        line.span(´ \t´) | TRUE;

        if sy := line.any(lower||upper)
        then    sy := sy || line.span(lower||upper||´_´) | ´´;
                t_sy := if keytab[sy] ~= undefined
                        then    keytab[sy]
                        else    predef[´ident´]
                        fi;
                return;

        elif sy := line.any(digit)
        then    sy := sy || line.span(digit) | ´´;
                t_sy := predef[´integer´];
                return;

        elif sy := line.lit(´:=´) | sy := line.any(´,;*/-+´)
        then    t_sy := kartab[sy];
                return;

        elif line.rtab(0)
        then    (line:=scan_string(infile.get()) & lnr:=lnr+1) |
                (sy := ´EOF´ & t_sy := predef[´EOF´] & return);

        else ermsg(´Illegal character:´||line.move(1),lnr);
        fi;
  od;
);
```

2. IMPLEMENTATION MANUAL

2.1. Introduction

SUMMER-statements in actions are not checked for syntactical correctness. They are simply read in one line after another, and remembered (see next section). The procedure that performs this task only stops when a new input line starts with the keyword 'EXIT', a new label specification (a '/'), the beginning of a new production (a '<') or end-of-file. This procedure is only called when the label (or 'INIT' or 'EXIT') is followed by a colon. When this colon is not found, the statements are assumed to be a part of the grammar.

2.2. Internal representation.

In PGEN, every production-rule is represented by a doubly threaded tree-structure. The nodes of the tree represent the constructions used in the definition part of the rule. The trees are created in such a way, that nesting is reflected in a clear and simple manner. This is done by distinguishing two layers within one level. This is illustrated in fig. 1 and 2.

The trees are constructed during the parsing of the input. All nodes (except root-nodes) also establish a link with the father-nodes.

PGEN uses five kinds of nodes: alt, term, nont, body and rule-root. These have the following properties.

alt — This node represents an alternative. The subtrees represent the constructions of which the alternative is composed.

term — Represents a terminal-symbol.

nont — Represents a call of a nonterminal.

body — Represents those constructions that can contain a sequence of alternatives, e.g. the compound and optional construction. As can be seen in fig. 2, the list is also represented by a body-node. To make this work, the list is supposed to consist of one alternative. Some advantages of this will be encountered later on.

rule-root — This node represents the complete right part of a production. It is used as root node of the tree, and has the same properties as a body-node, except a link with a parent-node.

Every node contains some information concerning the syntactic construction it represents. Some pieces of information are stored more than once in the tree to avoid a certain amount of tree-walking.

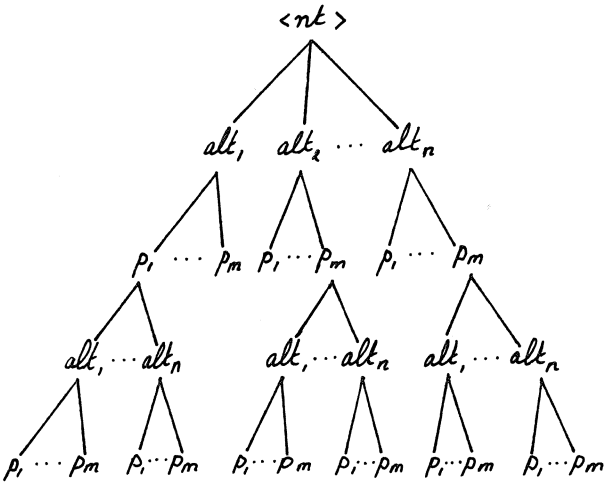


fig. 1 Tree representation of a production.
The leaves of the tree represent terminal-symbols and calls of nonterminals.

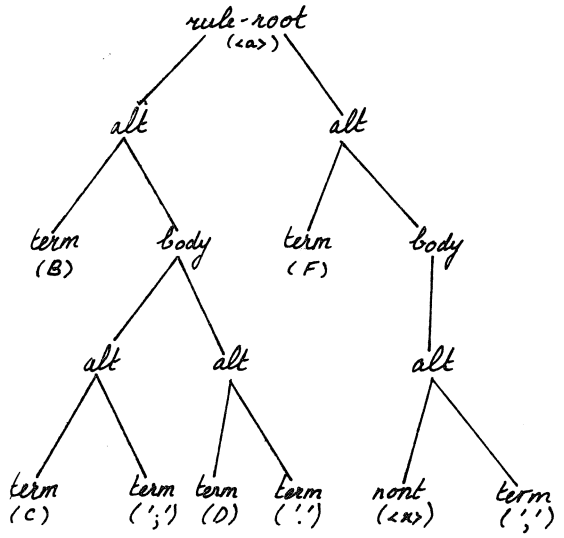


fig 2. Tree representation for

<a> ::= B (C ´;´ | D ´.´) | F { <x> ´,´ }+.

Every node contains a field which indicates whether the represented construction can produce the empty sentence. This field, called state, can have three values: EMPTY, indicating that the represented construction can produce the empty sentence, NONEMPTY, indicating that it can not, and UNDECIDED, when it is not known yet.

The first-symbols of a construction are stored in the node that represents it. First-symbols are not stored in nodes that refer to a non-terminal (nont-nodes).

Every node, except "alt" nodes, has a field containing print information of the represented construction.

Besides these fields, every node contains links to subtrees (children-nodes) and a link back to the parent-node. Every node also contains a field in which the semantic action related to the represented construction can be stored.

2.2.1. Relations between nodes.

Relations must be specified that define the flow of information through the tree. This information has to do with the state and first-symbols of a node. Other fields, such as a description of exterior aspects, can be derived directly from the input, and do not depend on subtrees.

The state of some nodes does not depend on subtrees. The state of a node which represents an option or a construction that may be repeated zero or more times, is always EMPTY. The state of a terminal-symbol that is used as separator in a list construction, is also EMPTY (see section 1.2). Normally, the state of a terminal-node is NONEMPTY.

The state of a body-node, depends on the states of its alternatives. If one alternative can produce the empty sentence, then the state of the node is EMPTY, otherwise it is NONEMPTY. *) In an alternative the state is EMPTY if the state of every subtree is EMPTY. If the state can not yet be determined, the state is UNDECIDED.

The rules for the propagation of first-set information are also straightforward. In an alternative the first-set is determined from the first-sets of the first X subtrees if the state of the X-th subtree is NONEMPTY and the state of all preceding subtrees is EMPTY. If the state of the alternative is EMPTY, the first-set is determined from the first-sets of all subtrees. For a body-node, the first-set equals the union of the first-sets of the alternatives (see fig. II).

---

*) Note that the representation chosen for list constructions allows this general formulation of these relations.

2.2.2. Propagation.

These relations must be applied to every node in every tree. Our only task is to provide an algorithm which visits every node in the tree, applying these relations. In other terms, the algorithm must store the information we need in every node of every tree.

The simplest method seems to be the following. The process starts by visiting the root of the tree representing the start symbol of the grammar, and visits every subtree in prefix order. When a call of another nonterminal is encountered, the corresponding tree is visited using the same process. When returning to a node all necessary information can be obtained, by referring to the subtrees.

This method provides a simple manner to discover left-recursion. Every time a new tree is entered, the name of the tree is stored, and if the name was already stored, a set of left-recursive rules has been discovered.

Unfortunately, it is possible (and easy) to construct grammars for which this method does not work, since the information of nonterminals is mutually dependent. We therefore decided to use another method. The algorithm used by PGEN is divided in two parts, each performed in a separate phase of the program.

The first part works in tune with the reading of the input and the construction of the trees. All nodes whose state and first-set are determined (i.e. 'determined nodes'), pass this information to their parent node, obeying the relations specified in the previous section. During this phase, state and first-set can be determined for every construction that consists only of terminal-symbols (see fig. 3). This process is described in algorithm 1.

```
if  state and first-set of current node are determined
then
        pass-on the information to the parent-node,
        using the relations specified in 2.2.1.
fi
```

Algorithm 1. First phase of the retrieval algorithm.

This first phase leaves us with a set of trees, containing 'holes', where the state and first-set of a node are not yet determined. The information needed by these nodes depends on nonterminals (see also fig. 3).

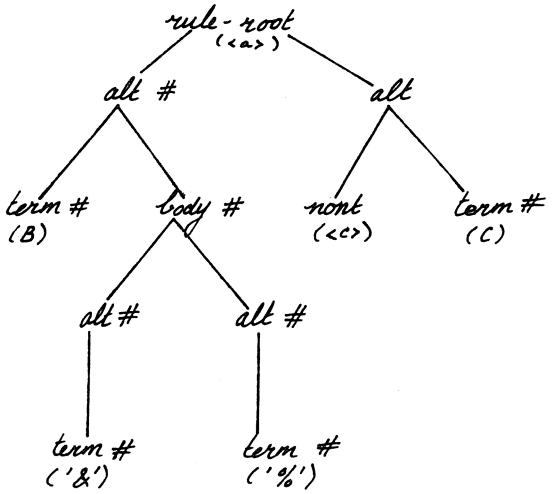


fig. 3 Tree representing the production

<a> ::= B ('&' | '%') | <c> C .

after the completion of the first phase of the algorithm. Nodes marked with '#' are determined.

The second phase attempts to determine the information in every tree, by passing on the information of already determined productions to the places where they are used in other productions. In order to find these places a table of uses is associated with each production-rule.

This process is based on the following property of context-free grammars: after the first phase at least one production exists whose state and first-set are determined. When this information is passed-on to every point of use, and is propagated to higher levels, again the state and first-set of at least one production must become determined. When no new productions are determined and not all rules have been determined, then the remaining set of rules contains a subset of productions that have a left-recursive relationship (see algorithm 2).

We shall not prove this property here, but only give a short motivation. When a production cannot be determined, the information which is needed depends on at least one other nonterminal. This implies that the specific nonterminal cannot be determined. When we have a set of rules that can not be determined, all these rules depend on nonterminals which are not determined either. But this means that the

nonterminals they depend on are in the set too. Hence, one or more of these rules are defined in a left-recursive way.

```
while  not all productions are determined
do
        count := 0
        for i in productions
        do
                if production determined and not yet
                   filled-in
                then
                        for j in references[i]
                        do      fill-in(j,i)
                        od
                        count := count + 1
                fi
        od
        if count = 0
        then    no rules found in this pass
                (partial) left recursion in
                remaining rules.
        fi
od
```

Algorithm 2. Second phase; filling in the points of use.
Note that the procedure fill-in passes the information on to the points of use. From there it can be 'rolled-up' to higher levels, using the relations specified in 2.2.1.

We shall now give two examples of this method. As first example we use the following part of a grammar for which the originally proposed method does not work.

```
<prog>  ::= <decl> BEGIN <stat> END.
<decl>  ::= <proc> | <var>.
<proc>  ::= PROC <ident> '(' ( <decl> | <stat> ) ')'.
<var>   ::= VAR <ident>.
<stat>  ::= ...
```

After phase one, productions <proc> and <var> are determined. In the first pass of the second phase, we can fill in the information of these productions in rule <decl>, which becomes determined now. This allows us to fill in <decl>. Assuming that <stat> is determined too, there are no problems to determine the compound in <proc>.

The second example illustrates how left-recursion is found. We define <a> as follows:

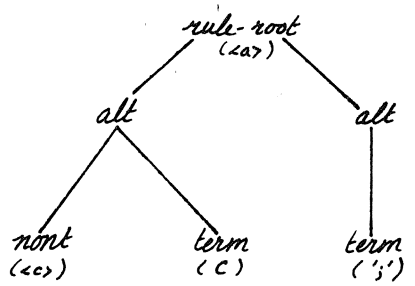
```
<a> ::= <b> | '.' .
<b> ::= [ <c> ] <d> | B .
<c> ::= C [ '&' ] .
<d> ::= [ <b> ] D .
```

After phase one, production <c> is determined. The state is NONEMPTY and the first-set consists of the symbol 'C'. Passing this information to the point of use in <b> does not determine any other production. This means that <a>, <b> and <d> can not be determined.

The various kinds of nodes contain some auxiliary fields to facilitate the above information retrieval process. Every alt-node contains a field called 'point' which at any time indicates the first subtree whose state is NONEMPTY, and whose first-symbols have been assembled. This field is modified when new information becomes available. When the state of the alternative itself is EMPTY, the 'point' is set to the number of subtrees. For example, if <a> is defined as:

```
<a> ::= <c> C | ';' .
```

then the tree representation after the first phase is:



The point of the first alternative is 2 (caused by the terminal-symbol 'C'), and the point of the second one is 1. If <c> appears to be NONEMPTY, then the point of the first alternative must be modified.

Second, every nont-, alternative and body-node contains a field called 'first_UND' (first_UNDECIDED). This field indicates the number of subtrees which (partially) determine the first-set of the node, but whose own first-symbols are not yet determined. If we consider the example given above, then 'first_UND' of the first alternative is 1, which is caused by the reference of <c>, and first_UND of <a> is also 1, which is caused by alternative 1.

2.3. LL(1)-check

The representation of production-rules allows a straightforward implementation of the check of the LL(1)-restrictions. The process is split up into two procedures; the first procedure checks restriction 1 for body-type nodes; the second procedure checks restriction 2 for all subtrees of an alternative node that satisfy the specified conditions, i.e. nodes whose state is EMPTY or that were followed by a '+'. The two procedures call each other recursively when nesting occurs.

The first procedure uses a short-cut to improve the speed of the checker. When the first-symbols of a body-node are assembled, the occurrences of each symbol are counted. If any symbol occurs more than once, then we have found a violation. An error-message is also given when two or more alternatives can produce the empty sentence.

The second procedure uses the same short-cut as the first one. Again the occurrences of the first-symbols are counted. A violation of restriction 2 within the subtrees that determine the first-set of the alternative can now be found immediately. For example, when <a> is defined as:

```
<a> ::= [ B ] B .
```

then the symbol B will be found twice and this violates restriction 2.

Subtrees that do not determine the first-set of an alternative must also be checked. Furthermore we must call the first procedure for all body-nodes that are subtrees of the alternative. For the subtrees that do not determine the first-set of an alternative, a follow-set must be assembled. This is only done for the nodes with state EMPTY, or with suffix '+' in the grammar. For these nodes, the follow-set and the first-set must be disjoint.

The procedure get-follow() reflects the relations concerning the assembly of the follow-symbols, as they were specified in section 1.2. The scheme is adapted at two points. First, the follow-symbols of a production are only assembled once, and stored in the root-node of the corresponding tree. This avoids a time-consuming search. Second, a modification was added to prevent endless searches.

2.4. Code generation

The last part of the program PGEN generates a parser. Declarations for parsing procedures and various sets are generated simultaneously on a number of intermediate files. The following intermediate files are created by the generator.

file.co This file contains the generated parsing procedures. One procedure is generated for each production rule.

file.fs Declarations of first-sets.

file.rs This file contains remainder-sets and miscellaneous sets.

file.tb All tables for the symbol-to-set representation.

By combining these intermediate files with the standard procedures in "pglib" and "pglib.ns", a complete parser is constructed, which can be compiled by the SUMMER compiler.

2.4.1. Generation of parser procedures

The generation of the parser is performed by scanning the tree representation of the grammar in a recursive way and generating one procedure for every rule. This is mainly done by the procedure ´gencode´, which is assisted by a few other procedures to perform actions like generating the user-defined SUMMER statements, generation of error-messages, declaration of several kinds of symbol-sets and generation of sufficient calls of the error-recovery procedure ´testsym´. The procedures ´gencode´ and ´body_code´ generate the parser-procedures.

We will discuss the most typical skeletons that are generated for the different types of nodes in the tree representing the grammar.

The procedure ´body_code´ closely cooperates with the procedure ´gencode´. Body_codes handles the body_construction. One of the alternatives in a body may be empty. As the alternatives are selected by their first_symbols in the parsing method, this is only possible when the empty alternative is the last one. This is achieved by deferring the empty alternative to the end of the body. It is included in the else clause in the if-then-elif ... else-fi construction that has to be generated for this body. Only one empty alternative may occur. Its position in the body is not important. The following skeleton is generated for a body-construction:

```
if symbol_in_first-set_alt1
then code_for_alt1
elif symbol_in_first-set_alt2
then code_for_alt2
else code_for_empty_alt
fi
```

For the list construction the following skeleton is used:

```
while first-set_of_list[t_sy] = 1
do
        actions_for_the_main_part_
         of_the_list_construction

        code_for_the_main_part_
         generated_by_a_recursive_
         call_of_gencode

        if t_sy = list_separator
        then
                nextsym;
                testsym_for_the_main_
                  part_of_the_list
        elif main_part[t_sy] = 1
        then error(separator missing)
        fi
od;
actions_for_the_whole_list
```

If the list is NONEMPTY (e.g. followed by a ´+´) then the generated code will be preceded by a call of the procedure ´testsym´, which demands that the list consist of at least one element.

There are two possibilities for the ´term´ construction. When the term-node is nonempty then the code looks like:

```
if t_sy = term_symbol
then    action
        nextsym;
        testsym(...);
else error(...);
fi
```

If the construction is followed by a ´*´ a while-loop is generated:

```
while t_sy = term_symbol
do
        action
        nextsym;
        testsym(...):
od
```

Note that the while-construction appears in the then-clause of the scheme for terms, in case the term was followed by a ´+´.

The next case handles the ´nont´ construction. For a nonterminal a procedure call has to be generated. Now there are three possibilities:

the '*' repetition, for which a single while loop is generated, the '+' repetition which causes an extra call before the while-loop, and the non-repeating one, for which a simple call is generated. Depending on the existence of a tag in the syntax rule, the call is generated as a function (var := ...) or as a plain procedure call.

The last case is the rule_root. It looks like:

```
proc p_name (dont_skip)
(
        declarations_and_init_code

        testsym(...);
        code_generated_by_body_code

        exit_code
);
```

Auto-declared variables (assignment vars for the nont_calls) are part of init_code.

2.5. Example of a generated parser

This paragraph shows an example of a generated procedure. The example from chapter 1.1.4 is extended to:

```
LEXICAL identifier.

<input> ::= ids: <id-list>.

  EXIT :  put('number of identifiers:',ids,'\n');

<id-list> ::= { <identifier> /il/ ',}+ ';'.

  INIT :  var count := 0;   # counts the identifiers #
          put(' The identifiers are :\n');
  /il/ :  put( sy,'\n');
          count := count + 1;
  EXIT :  return(count);
```

The generated procedures for these two production rules are displayed on the next page.

As can be seen in the generated procedures, some of the error messages are generated from the names of the production rules. A badly chosen name of a rule will give an unclear error message.

```
proc  p_id_list(dont_skip)
( # line: 9#
  var count := 0;    # counts the identifiers #
  put(´ The identifiers are :\n´);
  testsym(s_2,dont_skip,´<id_list>´);
  if t_sy = 2
  then
    while s_2[t_sy] = 1
    do# line: 11#
      put( sy,´\n´);
      count := count + 1;
      nextsym;
      testsym(UN2(dont_skip,r_id_list_0_0_0_0),EMPTY,´´);
      if t_sy = 1
      then
        nextsym;
        testsym(s_2,UN2(dont_skip,
                r_id_list_0_0_0_1),´"identifier"´);
      elif s_2[t_sy] = 1
      then error(´Separating ","´,lnr);
      fi;
    od;
    if t_sy = 0
    then
      nextsym;
      testsym(dont_skip,EMPTY,´´);
    else error(´";"´,lnr);
    fi;
  fi;
  # line: 13#
  return(count);
);

program  p_input(args)
( var dont_skip := s_3;
  var ids ;
  nextsym;
  testsym(s_2,dont_skip,´<input>´);
  if t_sy = 2
  then
    ids := p_id_list(dont_skip);
  fi;
  # line: 5#
  put(´number of identifiers:´,ids,´\n´);
  if errcnt > 0
  then stop(1)
  fi;
);
```

APPENDIX I.

The syntax of the meta-language.

```
LEXICAL         keyword,string,ident,prog.

<grammar>               ::= [ <lexicals> ] <rule>*.

<lexicals>              ::= 'LEXICAL' { <ident> ',' }+ '.' .

<rule>                  ::= <rule-def> [ <code-spec> ] .

<code-spec>             ::=
                              [ 'INIT' ':' <prog> ]
                              ( <label> ':' <prog> )*
                              [ 'EXIT' ':' <prog> ] .

<rule-def>              ::= <name> '::=' <rule-body> '.' .

<rule-body>             ::= { <alternative> '|' }* .

<alternative>           ::= [ <label> ] <primary>+ .

<primary>               ::= ( ( <terminal-symbol> |
                                <rule-call> | <compound> )
                                        [ '+' | '*' ]
                              | <list> | <option>
                            ) [ <label> ] .

<option>                ::= '[' <rule-body> ']' .

<list>                  ::= '{' [ <label> ] <primary>
                                ( <terminal-symbol> | <name> )
                                [ <label> ]
                            '}' ( '+' | '*' ).

<compound>              ::= '(' <rule-body> ')' .

<terminal-symbol>       ::= <keyword> | <string> .

<name>                  ::= '<' <ident> '>' .

<rule-call>             ::= [ <ident> ':' ] <name> .

<label>                 ::= '/' <ident> '/'.
```

APPENDIX II

Table of variables and procedures, used by the generated parser. These names should not be redefined. Names marked with '*)' may be useful to the user.

line	*)	can be used in nextsym
lnr	*)	current line number
errcnt	*)	number of errors
sy	*)	current input symbol
t_sy		type of sy
lower	*)	can be used in scanner
upper	*)	,,
ASCII	*)	,,
digit	*)	,,
predef	*)	table with symbol types
kartab	*)	,,
keytab	*)	,,
SETSIZE		used for set manipulation
empty		,,
args	*)	call arguments of generator
f_...		symbol sets
r_...		,,
s_...		,,
ermsg	*)	proc for error messages
error	*)	,,
nextsym		scanner procedure
testsym		recovery procedure
get_str		procedure used in default nextsym
conv_str		,,
get_number		,,
dont_skip		symbol set
SET		set handling procedure
UN2		,,
UN3		,,

APPENDIX III

PGEN(1) UNIX Programmer´s Manual PGEN(1)

NAME

pgen - generate a parser with error recovery.

SYNOPSIS

pgen [-n] [-i] [-u] file.syn

DESCRIPTION

pgen is a parser generator that reads a syntax description from ´file.syn´. After checking the correctness of the given grammar, it generates a parser written in SUMMER for the given grammar.

The filename ´file´ is meant to be the basename of the used ´.syn´ file.

pgen has three options:

-i pgen will not remove the intermediate files of the generator.

-n pgen inserts the file ´file.ns´ in order to enable the user to write his own lexical procedures. By default pgen would insert ´pglib.ns´, which contains the default scanner procedures.

-u pgen inserts user-supplied declarations of global procedures, classes and variables from the file ´file.ud´.

The generated parser is left on the file ´file.sm´ and can be translated by sumc

FILES

file.syn	syntax definition
file.sm	generated parser
file.ns	user defined nextsym procedure
file.ud	user defined declarations
file.syn.er	error messages from the generator.
file.syn.tb	generated tables
file.syn.fs	generated first-sets
file.syn.rs	generated remainder-sets
file.syn.co	generated code

REFERENCES

[1] P. Klint, SUMMER reference manual, Mathematisch Centrum, Amsterdam (1981).

[2] U. Ammann, "Error recovery in recursive descent parsers," Berichte des Instituts für Informatik Vol. 25 (1978).

[3] N. Wirth, Algorithms + Datastructures = Programs, Prentice Hall, Englewood Cliffs, N.J., U.S.A. (1973).

[4] S. C. Johnson, "Yacc - Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

[5] Th. F. de Ridder, Dictaat vertalerbouw, I.H.B.O. "De Maere" afd. H.I.O. , Enschede (1979).