CHAPTER 3

Scanning & Parsing

3.1 SCANNING

Scanning is the process of recognizing the lexical components in a source string. As stated in Section 1.4.1.1, the lexical features of a language can be specified using Type-3 or regular grammars. This facilitates automatic construction of efficient recognizers for the lexical features of the language. In fact the scanner generator LEX generates such recognizers from the string specifications input to it (see Section 1.5.1).

In the early days of compilers, it was not possible to generate scanners automatically because the theory of PL grammars was not sufficiently understood and practiced. Specifications of lexical strings were excessively complex and scanners had to be hand coded to implement them. Fortran is notorious for very complex specifications of this kind.

Example 3.1 Successful scanning of some Fortran constructs requires interaction with the parser, e.g. consider the Fortran statements

```
DO 10 I = 1,2 and
DO 10 I = 1.2
```

The former is a DO statement while the latter is an assignment to a variable named `DO10I` (note that blanks are ignored in Fortran). Thus, scanning can only be performed after presence of the ',' identifies the former as a DO statement and its absence identifies the latter as an assignment statement. Fortunately, modern PL's do not contain such constructs.

Before proceeding with the details of scanning, it is important to understand the reasons for separating scanning from parsing. From Section 1.4.1.1, it is clear that each Type-3 production specifying a lexical component is also a Type-2 production. Hence it is possible to write a single set of Type-2 productions which specifies both lexical and syntactic components of the source language. However, a recognizer for

Type-3 productions is simpler, easier to build and more efficient during execution than a recognizer for Type-2 productions. Hence it is better to handle the lexical and syntactic components of a source language separately.

Finite state automata

Definition 3.1 (Finite state automaton (FSA)) *A* finite state automaton *is a triple* (S, Σ, T) *where*

- S *is a finite set of states, one of which is the* initial state s_{init}, *and one or more of which are the* final states
- Σ *is the alphabet of source symbols*
- T *is a finite set of state transitions defining transitions out of each* $s_i \in S$ *on encountering the symbols of* Σ.

We label the transitions of FSA using the following convention: A transition out of $s_i \in S$ on encountering a symbol $symb \in \Sigma$ has the label *symb*. We say a symbol *symb* is racognized by an FSA when the FSA makes a transition labelled *symb*. The transitions in an FSA can be represented in the form of a state transition table (STT) which has one row for each state $s_i \in S$ and one column for each symbol $symb \in \Sigma$. An entry STT(s_i, *symb*) in the table indicates the id of the new state entered by the FSA if there exists a transition labelled *symb* in state s_i. If the FSA does not contain a transition out of state s_i for *symb*, we leave STT(s_i, *symb*) blank. A state transition can also be represented by a triple (old state, source symbol, new state). Thus, the entry STT(s_i, *symb*) = s_j and the triple (s_i, *symb*, s_j) are equivalent.

The operation of an FSA is determined by its current state s_c. The FSA actions are limited to the following: Given a source symbol x at its input, it checks to see if STT(s_c, x) is defined—that is, if STT(s_c, x) = s_j, for some s_j. If so, it makes a transition to s_j, else it indicates an error and stops.

Definition 3.2 (DFA) *A* deterministic finite state automaton *(DFA) is an FSA such that* $t_1 \in T$, $t_1 \equiv (s_i, symb, s_j)$ *implies* $\nexists\, t_2 \in T$, $t_2 \equiv (s_i, symb, s_k)$.

Transitions in a DFA are deterministic—that is, at most one transition exists in state s_i for a symbol *symb*. Figure 3.1 illustrates the operation of a DFA. At any point in time, the DFA would have recognized some *prefix* α of the source string, possibly

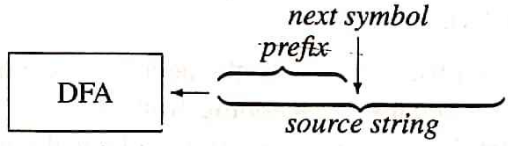


Fig. 3.1 Operation of a DFA

the null string, and would be poised to recognize the symbol pointed to by the pointer *next symbol*. The operation of a DFA is history-sensitive because its current state is a function of the prefix recognized by it. The DFA halts when all symbols in the source string are recognized, or an error condition is encountered. It can be seen that a DFA recognizes the longest valid prefix before stopping.

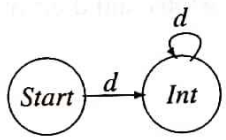
The validity of a string is determined by giving it at the input of a DFA in its initial state. The string is valid if and only if the DFA recognizes every symbol in the string and finds itself in a final state at the end of the string. This fact follows from the deterministic nature of transitions in the DFA.

Example 3.2 Figure 3.2 shows a DFA to recognize integer strings according to the Type-3 rule

$$<integer> ::= d \mid <integer>\ d$$

	Next Symbol
State	*d*
Start	*Int*
Int	*Int*

(a)



(b)

Fig. 3.2 DFA to recognise integer strings

where d represents a digit. Part (a) of the figure shows the STT for the DFA. Part (b) shows a diagrammatic representation of the states and state transitions. A transition from state s_i to state s_j on symbol *symb* is depicted by an arrow labelled *symb* from s_i to s_j. The initial and final states of the DFA are *Start* and *Int* respectively. Transitions during the recognition of string 539 are as given in the following table:

Current state	*Input symbol*	*New state*
Start	5	*Int*
Int	3	*Int*
Int	9	*Int*

The string leaves the DFA in state *Int* which is a final state, hence the string is a valid integer string. A string `5ab9` is invalid because no transition marked 'letter' exists in state *Int*.

Regular expressions

In Ex. 3.2 a single Type-3 rule was adequate to specify a lexical component. However, many Type-3 rules would be needed to specify complex lexical components like real constants. Hence we use a generalization of Type-3 productions called a *regular expression*.

Example 3.3 An organization uses an employee code which is obtained by concatenating the section id of an employee, which is alphabetic in nature, with a numeric code. The structure of the employee code can be specified as

$$\begin{aligned} &\langle \textit{section code} \rangle && ::= && l \mid \langle \textit{section code} \rangle\, l \\ &\langle \textit{numeric code} \rangle && ::= && d \mid \langle \textit{numeric code} \rangle\, d \\ &\langle \textit{employee code} \rangle && ::= && \langle \textit{section code} \rangle\, \langle \textit{numeric code} \rangle \end{aligned}$$

Note that a specification like

$$\langle s\_code \rangle \quad ::= \quad l \mid d \mid \langle s\_code \rangle\, l \mid \langle s\_code \rangle\, d$$

would be incorrect !

The regular expression generalizes on Type-3 rules by permitting multiple occurrences of a string form, and concatenation of strings. Table 3.1 shows regular expressions and their meanings.

Table 3.1 Regular expressions

Regular expression	*Meaning*
r	string r
s	string s
$r.s$ or rs	concatenation of r and s
(r)	same meaning as r
$r \mid s$ or $(r \mid s)$	alternation, i.e. string r or string s
$(r) \mid (s)$	alternation
$[r]$	an optional occurrence of string r
$(r)^*$	≥ 0 occurrences of string r
$(r)^+$	≥ 1 occurrences of string r

Example 3.4 The employee codes of Ex. 3.3 can be specified by the regular expression

$$(l)^+(d)^+$$

Some other examples of regular expressions are

integer	$[+ \mid -](d)^+$
real number	$[+ \mid -](d)^+.(d)^+$
real number with optional fraction	$[+ \mid -](d)^+.(d)^*$
identifier	$l\,(l \mid d)^*$

Building DFAs

A DFA for a Type-3 specification can be built using some simple rules. Building a DFA for a regular expression can be achieved by repeated application of the same simple rules. However it is a tedious process, hence it is best to automate the process of building DFAs. In the following, we shall not discuss the building of DFAs but assume that they can be built by a procedure described in the literature cited at the end of the chapter.

The lexical components of a source language can be specified by a set of regular expressions. Since an input string may contain any one of these lexical components, it is necessary to use a single DFA as a recognizer for valid lexical strings in the language. Such a DFA would have a single initial state and one or more final states for each lexical component.

Example 3.5 Figure 3.3 shows a DFA for recognizing identifiers, unsigned integers and unsigned real numbers with fractions. The DFA has 3 final states – *Id*, *Int* and *Real* corresponding to identifier, unsigned integer and unsigned real respectively. Note that a string like '25.' is invalid because it leaves the DFA in state s_2 which is not a final state.

State	*Next Symbol*		
	l	*d*	.
Start	*Id*	*Int*	
Id	*Id*	*Id*	
Int		*Int*	s_2
s_2		*Real*	
Real		*Real*	

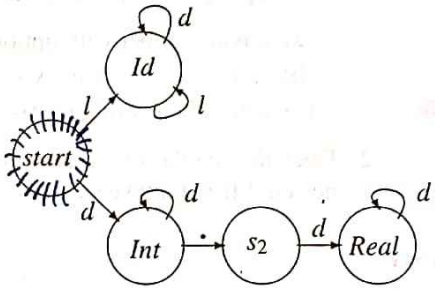


Fig. 3.3 A combined DFA for integers, real numbers and identifiers

Performing semantic actions

Semantic actions during scanning concern table building and construction of tokens for lexical components. These actions are associated with the final states of a DFA. The semantic actions associated with a final state s_f are performed after the DFA recognizes the longest valid prefix of the source string corresponding to s_f.

Writing a scanner

We will use a notation analogous to the LEX notation (see Section 1.5.1) to specify a scanner. A scanner for integer and real numbers, identifiers and reserved words of a language is given in Table 3.1.

Table 3.2 Specification of a scanner

Regular expression	*Semantic actions*
$[+ \mid -](d)^+$	{Enter the string in the table of integer constants, say in entry *n*. Return the token $\boxed{Int\ \#n}$ }
$[+ \mid -]((d)^+.(d)^* \mid (d)^*.(d)^+)$	{Enter in the table of real constants. Return the token $\boxed{Real\ \#m}$ }
$l(l \mid d)^*$	{Compare with reserved words. If a match is found, return the token $\boxed{Kw\ \#k}$, else enter in symbol table and return the token $\boxed{Id\ \#i}$ }

EXERCISE 3.1

1. Develop regular expressions and DFAs for the following
   (a) a real number with optional integer and fraction parts,
   (b) a real number with exponential part,
   (c) a comment string in Pascal or C language.
2. Does the regular expression developed by you in problem 1 permit comments to be nested ? If not, make suitable changes.

3.2 PARSING

The goals of parsing are to check the validity of a source string, and to determine its syntactic structure. For an invalid string the parser issues diagnostic messages reporting the cause and nature of error(s) in the string. For a valid string it builds a parse tree to reflect the sequence of derivations or reductions performed during parsing. The parse tree is passed on to the subsequent phases of the compiler.

As described in Section 1.4.1, the fundamental step in parsing is to derive a string from an NT, or reduce a string to an NT. This gives rise to two fundamental approaches to parsing—*top down parsing* and *bottom up parsing*, respectively.

Parse trees and abstract syntax trees

A parse tree depicts the steps in parsing, hence it is useful for understanding the process of parsing. However, it is a poor intermediate representation for a source string because it contains too much information as far as subsequent processing in the compiler is concerned. An *abstract syntax tree* (AST) represents the structure of a source string in a more economical manner. The word 'abstract' implies that it is a representation designed by a compiler designer for his own purposes. Thus the

designer has total control over the information represented in an AST. It thus follows that an AST for a source string is not unique, whereas a parse tree is.

Example 3.6 Figure 3.4(a) shows a parse tree for the source string `a+b*c` according to Grammar (1.3). Figure 3.4(b) shows an AST for the same string. It contains sufficient information to represent the structure of the string. Note its economy compared to the parse tree.

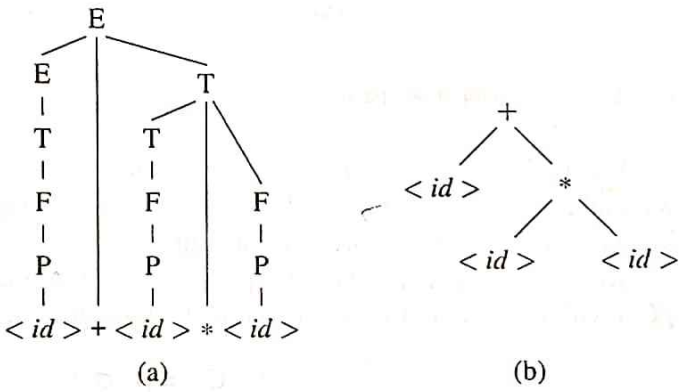


Fig. 3.4 Parse tree and AST

3.2.1 Top Down Parsing

Top down parsing according to a grammar G attempts to derive a string matching a source string through a sequence of derivations starting with the distinguished symbol of G. For a valid source string α, a top down parse thus determines a derivation sequence

$$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \alpha.$$

We shall identify the important issues in top down parsing by trying to develop a naive algorithm for it.

Algorithm 3.1 (Naive top down parsing)

1. *Current sentential form* (CSF) := 'S';
2. Let CSF be of the form $\beta A \pi$, such that β is a string of Ts (note that β may be null), and A is the leftmost NT in CSF. Exit with success if CSF = α.
3. Make a derivation $A \Rightarrow \beta_1 B \delta$ according to a production $A ::= \beta_1 B \delta$ of G such that β_1 is a string of Ts (again, β_1 may be null). This makes CSF = $\beta\beta_1 B \delta\pi$.
4. Go to Step 2.

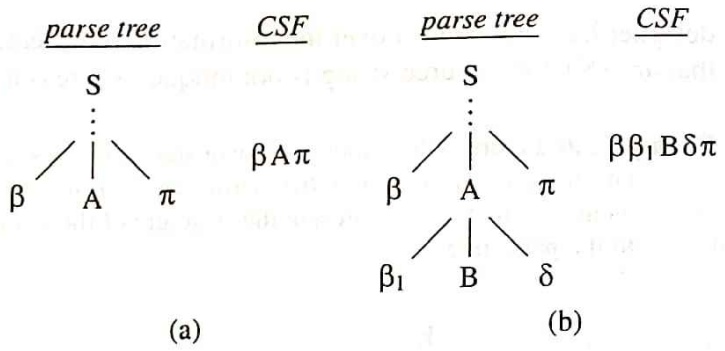


Fig. 3.5 Derivation $A \Rightarrow \beta_1 B\delta$ in top down parsing

Figure 3.5 depicts a step in top down parsing according to Algorithm 3.1. Since we make a derivation for the leftmost NT at any stage, top down parsing is also known as *left-to-left parsing* (LL Parsing).

Algorithm 3.1 lacks one vital provision from a practical viewpoint. Let CSF $\equiv \gamma C \delta$ with C as the leftmost NT in it and let the grammar production for C be

$$C ::= \rho \mid \sigma$$

where each of ρ, σ is a string of terminal and non-terminal symbols. Which RHS alternative should the parser choose for the next derivation ? The alternative we choose may lead us to a string of Ts which does not match with the source string α. In such a case, other alternatives would have to be tried out until we derive a sentence that matches the source string (i.e., a successful parse), or until we have systematically generated all possible sentences without obtaining a sentence that matches the source string (i.e., an unsuccessful parse). A naive approach to top down parsing would be to generate complete sentences of the source language and compare them with α to check if a match exists. However, this approach is inefficient for obvious reasons.

We introduce a check, called a *continuation check*, to determine whether the current sequence of derivations may be able to find a successful parse of α. This check is performed as follows: Let CSF be of the form $\beta A \pi$, where β is a string of n Ts. All sentential forms derived from CSF would have the form $\beta \ldots$. Hence, for a successful parse β must match the first n symbols of α. We can apply this check at every parsing step, and abandon the current sequence of derivations any time this condition is violated. The continuation check may be applied incrementally as follows: Let CSF $\equiv \beta A \pi$, then the source string must be $\beta \ldots$ (else we would have abandoned this sequence of derivations earlier). If the prediction for A is $A \Rightarrow \beta_1 B \delta$, where β_1 is a string of m terminal symbols, then the string β_1 must match m symbols following β in the source string (see Fig. 3.5). Hence we compare β_1 with m symbols to the right of β in the source string. This incremental check is more economical than a continuation check which compares the string $\beta\beta_1$ with $(n+m)$ symbols in the source string.

Predictions and backtracking

A typical stage in top down parsing can be depicted as follows:

$$\begin{array}{rcl} \text{CSF} & \equiv & \beta A \pi \\ & & \text{SSM} \\ & & \downarrow \\ \text{Source string} & : & \beta \ \texttt{t} \end{array}$$

where CSF $\equiv \beta A \pi$ implies $S \stackrel{*}{\Rightarrow} \beta A \pi$ and the *source string marker* (SSM) points at the first symbol following β in the source string, i.e. at the terminal symbol '`t`'. We have already seen that if CSF = $\beta A \pi$, the source string must have the form $\beta \ldots$.

Parsing proceeds as follows: We identify the leftmost nonterminal in CSF, i.e. A. We now select an alternative on the RHS of the production for A. Since we do not know whether the derived string will satisfy the continuation check, we call this choice a *prediction*. The continuation check is applied incrementally to the terminal symbol(s), if any, occurring in the leftmost position(s) of the prediction. SSM is incremented if the check succeeds and parsing continues. If the check fails, one or more predictions are discarded and SSM is reset to its value before the rejected prediction(s) was made. This is called *backtracking*. Parsing is now resumed.

Implementing top down parsing

The following features are needed to implement top down parsing:

1. *Source string marker* (SSM): SSM points to the first unmatched symbol in the source string.
2. *Prediction making mechanism:* This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string in L_G can be derived from S.
3. *Matching and backtracking mechanism:* This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to by SSM. (This implements the incremental continuation check.) Backtracking is performed if the match fails. This involves resetting CSF and SSM to earlier values.

Continuation check and backtracking is performed in Step 3 of Algorithm 3.1. A complete algorithm incorporating these features can be found in (Dhamdhere, 1983).

Example 3.7 Lexically analysed version of the source string `a+b*c`, viz. $< id > + < id > * < id >$ is to be parsed according to the grammar

$$\begin{array}{rcl} \text{E} & ::= & \text{T} + E \mid \text{T} \\ \text{T} & ::= & \text{V} * T \mid \text{V} \\ \text{V} & ::= & < id > \end{array} \qquad (3.1)$$

The prediction making mechanism selects the RHS alternatives of a production in a left-to-right manner. First few steps in the parse are:

1. SSM := 1; CSF := E;
2. Make the prediction E ⇒ T + E. Now, CSF = T + E.
3. Make the prediction T ⇒ V * T. CSF = V * T + E.
4. Make the prediction V ⇒ < *id* >. CSF = < *id* > * T + E. < *id* > matches with the first symbol of the source string. Hence, SSM := SSM + 1;
5. Match the second symbol of the prediction in Step 3, viz. '*'. This match fails, hence reject the prediction T ⇒ V * T. SSM and CSF are reset to 1 and T + E, respectively. (The situation now resembles that at the end of Step 2.)
6. Make a new prediction for T, viz. T ⇒ V. CSF = V + E.
7. Make the prediction V ⇒ < *id* >. CSF = < *id* > + E. < *id* > matches with the first symbol of the source string. Hence, SSM := SSM + 1;
8. Match the second symbol of the prediction in Step 2, viz. '+'. Match succeeds. SSM := SSM + 1;
9. Make the prediction E ⇒ T + E. CSF = < *id* > + T + E.
10. Make the prediction T ⇒ V * T. CSF = < *id* > + V * T + E.
11. ....

The predictions surviving at the end of the parse are listed in Table 3.3.

Table 3.3 Predictions in top sown parsing

Prediction	*Predicted Sentential Form*
E ⇒ T + E	T + E
T ⇒ V	V + E
V ⇒ < *id* >	< *id* > + E
E ⇒ T	< *id* > + T
T ⇒ V * T	< *id* > + V * T
V ⇒ < *id* >	< *id* > + < *id* > * T
T ⇒ V	< *id* > + < *id* > * V
V ⇒ < *id* >	< *id* > + < *id* > * < *id* >

Comments on top down parsing

Two problems arise due to the possibility of backtracking. First, semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a part of a successful parse. Second, precise error reporting is not possible. A mismatch merely triggers backtracking. A source string is known to be erroneous only after all predictions have failed. This makes it impossible to pinpoint the violations of PL specification.

Grammars containing left recursion are not amenable to top down parsing. For example, consider parsing of the string <*id*> + < *id* > * < *id* > according to the grammar

$$\begin{aligned} \mathrm{E} &::= \mathrm{E} + T \mid \mathrm{T} \\ \mathrm{T} &::= \mathrm{T} * V \mid \mathrm{V} \\ \mathrm{V} &::= < id > \end{aligned}$$

The first prediction would be

$$\mathrm{E} \Rightarrow \mathrm{E} + \mathrm{T}$$

which makes E the leftmost NT in CSF once again. Thus, the parser would enter an infinite loop of prediction making. To make top down parsing feasible, it is necessary to eliminate left recursion. This can be achieved by rewriting the grammar using right recursion as follows

$$\begin{aligned} \mathrm{E} &::= \mathrm{T} + E \mid \mathrm{T} \\ \mathrm{T} &::= \mathrm{V} * T \mid \mathrm{V} \\ \mathrm{V} &::= < id > \end{aligned} \tag{3.2}$$

However, this method is time-consuming and error-prone for large grammars. An alternative is to systematically eliminate left-recursion using the following rule: Rewrite a left recursive production

$$\mathrm{E} ::= \mathrm{E} + T \mid \mathrm{T}$$

as

$$\begin{aligned} \mathrm{E} &::= \mathrm{T}\, E' \\ \mathrm{E}' &::= +\, T\, E' \mid \varepsilon \end{aligned} \tag{3.3}$$

The rationale for the rewriting is as follows: From the original production, it is clear that E produces a string consisting of one or more Ts separated by '+' symbols. Hence we write the production E ::= T E' with the expectation that E' will produce zero or more occurrences of the string '+ T'. This is what the production of E' achieves. After all left-recursive rules are converted in this manner, the resulting grammar can be used for top down parsing.

Top down parsing without backtracking

Elimination of backtracking in top down parsing would have several advantages—parsing would become more efficient, and it would be possible to perform semantic actions and precise error reporting during parsing. Prediction making becomes very crucial when backtracking is eliminated. The parser must use some contextual information from the source string to decide which prediction to make for the leftmost NT.

This is achieved as follows: If the leftmost NT is A and the source symbol pointed to by SSM is 't', parser selects that RHS alternative of A which can produce 't' as its first terminal symbol. An error is signalled if no RHS alternative can produce 't' as the first terminal symbol.

For the above approach to work, it is essential that at most one RHS alternative should be able to produce the terminal symbol 't' in the first position. The grammar may have to be modified to ensure this. As an example, consider parsing of the string $< id > + < id > * < id >$ according to Grammar (3.2). The first prediction is to be made using the production

$$E ::= T + E \mid T$$

such that the first terminal symbol produced by it would be $< id >$, the first symbol in the source string. From the grammar, we find that $T \Rightarrow V\ldots$ and $V \Rightarrow < id >$. Thus, any RHS alternative starting with a T can produce $< id >$ in the first position. However, both alternatives of E start with a T, so which one should the parser choose? To overcome this dilemma, we use *left-factoring* to ensure that the RHS alternatives will produce unique terminal symbols in the first position. The production for E is now rewritten as

$$\begin{aligned} E &::= T\,E'' \\ E'' &::= +\,E \mid \varepsilon \end{aligned}$$

The first prediction according to this grammar is $E \Rightarrow T\ E''$ since the first source symbol is '<*id*>'. When E" becomes the leftmost NT in CSF, we would make the prediction E"$\Rightarrow$ + E if the next source symbol is '+' and the prediction E"$\Rightarrow$ E in all other cases. If the next source symbol is '+', we would make the prediction E" $\Rightarrow$ + E, else we would make the prediction E" $\Rightarrow \varepsilon$. Thus, parsing is no longer by trial-and-error. Such parsers are known as *predictive parsers*.

The complete rewritten form of Grammar (3.2) is

$$\begin{aligned} E &::= T\,E'' \\ E'' &::= +\,E \mid \varepsilon \\ T &::= V\,T'' \\ T'' &::= *\,T \mid \varepsilon \\ V &::= < id > \end{aligned} \qquad (3.4)$$

Note that grammar (3.3) does not need left factoring since its RHS alternatives produce unique terminal symbols in the first position.

Example 3.8 Parsing of $< id > + < id > * < id >$ according to Grammar (3.4) proceeds as shown in Table 3.4. Note that *Symbol* is the symbol pointed to by SSM.

To start with, CSF = E and SSM points to '<*id*>'. The first three steps are obvious. In the 4^{th} step, SSM points to '+' and the leftmost NT is T". This leads to the prediction T"$\Rightarrow \varepsilon$.

Table 3.4 Top down parsing without backtracking

Sr. No.	*CSF*	*Symbol*	*Prediction*
1.	E	$< id >$	$E \Rightarrow T\ E''$
2.	$T\ E''$	$< id >$	$T \Rightarrow V\ T''$
3.	$V\ T''E''$	$< id >$	$V \Rightarrow < id >$
4.	$< id > T''E''$	+	$T'' \Rightarrow \varepsilon$
5.	$< id > E''$	+	$E'' \Rightarrow + E$
6.	$< id > + E$	$< id >$	$E \Rightarrow T\ E''$
7.	$< id > + T\ E''$	$< id >$	$T \Rightarrow V\ T''$
8.	$< id > + V\ T''E''$	id	$V \Rightarrow < id >$
9.	$< id > + < id > T''E''$	$*$	$T'' \Rightarrow * T$
10.	$< id > + < id > * T\ E''$	$< id >$	$T \Rightarrow V\ T''$
11.	$< id > + < id > * V\ T''E''$	$< id >$	$V \Rightarrow < id >$
12.	$< id > + < id > * < id > T''E''$	–	$T'' \Rightarrow \varepsilon$
13.	$< id > + < id > * < id > E''$	–	$E'' \Rightarrow \varepsilon$
14.	$< id > + < id > * < id >$	–	–

3.2.1.1 Practical Top Down Parsing

A recursive descent parser

A recursive descend (RD) parser is a variant of top down parsing without backtracking. It uses a set of recursive procedures to perform parsing. Salient advantages of recursive descent parsing are its simplicity and generality. It can be implemented in any language supporting recursive procedures.

To implement recursive descent parsing, a left-factored grammar is modified to make repeated occurrences of strings more explicit. Grammar (3.4) is rewritten as

$$\begin{aligned} E &::= T\ \{+\ T\}^* \\ T &::= V\ \{*\ V\}^* \\ V &::= < id > \end{aligned} \tag{3.5}$$

where the notation $\{..\}^*$ indicates zero or more occurrences of the enclosed specification. A parser procedure is now written for each NT of G. It handles prediction making, matching and error reporting for that NT. The structure of a parser procedure is dictated by the grammar production for the NT. If A ::= ..B.. is a production of G, the parser procedure for A contains a call on the procedure for B.

Example 3.9 A skeletal recursive descent parser for Grammar (3.5) is shown in Fig. 3.6. The parser returns an AST for a valid source string, and reports an error for an invalid string. The procedures `proc_E`, `proc_T` and `proc_V` handle the parsing for E, T and

```
procedure proc_E : (tree_root);
   /* This procedure constructs an AST for 'E'
      and returns a pointer to its root */
   var
      a, b : pointer to a tree node;
   begin
      proc_T (a);
         /* Returns a pointer to the root of tree for T */
      while (nextsymb = '+') do
         match ('+');
         proc_T (b);
         a := treebuild ('+', a, b);
               /* Builds an AST and returns pointer
                  to its root */
      tree_root := a;
      return;
end proc_E;

procedure proc_T (tree_root);
   var
      a, b : pointer to a tree node;
   begin
      proc_V (a);
      while (nextsymb = '*') do;
         match ('*');
         proc_V (b);
         a := treebuild ('*', a, b);
      tree_root := a;
      return;
end proc_T;

procedure proc_V (tree_root);
   var
      a : pointer to a tree node;
   begin
      if (nextsymb = < id >) then
         tree_root := treebuild (< id >, -, -);
         else print "Error !";
      return;
end proc_V;
```

Fig. 3.6 A recursive descent parser

V, respectively, and build ASTs for these NTs using the procedure `treebuild`. Procedure `match` increments SSM. Procedure `proc_E`, the procedure for E, always calls `proc_T` to perform parsing and AST building for a T. If the next source symbol (which is assumed to be contained in the parser variable `nextsymb`) is a '+', then another T is expected. Hence `proc_T` is called again. This process is repeated until the symbol following a T is not a '+'. At this point, `proc_E` returns to the parser control routine. The control routine should now check whether the entire source string has been successfully parsed, else indicate an error. Error detection and reporting is restricted to those parser procedures which produce terminal symbol(s) in the first position. For grammar (3.5) only the routine for V would declare error if anything but an $< id >$ is encountered. The reader can verify that error indication would be precise.

Note that the parser procedures do not call themselves recursively. However, indirect recursion would result if the grammar specification contains indirect recursion—for example, if a rule like

$$V ::= < id > \mid (E)$$

existed in grammar (3.5).

An LL(1) parser

An LL(1) parser is a table driven parser for left-to-left parsing. The '1' in LL(1) indicates that the grammar uses a look-ahead of one source symbol—that is, the prediction to be made is determined by the next source symbol. A major advantage of LL(1) parsing is its amenability to automatic construction by a parser generator.

Figure 3.7 shows the parser table for an LL(1) parser for Grammar (3.6).

$$\begin{array}{lcl} E & ::= & T\ E' \\ E' & ::= & +\ T\ E' \mid \varepsilon \\ T & ::= & V\ T' \\ T' & ::= & *\ V\ T' \mid \varepsilon \\ V & ::= & < id > \end{array} \qquad (3.6)$$

The parsing table (PT) has a row for each NT $\in$ SNT and a column for each T $\in \Sigma$. A parsing table entry PT(nt_i, t_j) indicates what prediction should be made if nt_i is the leftmost NT in a sentential form and t_j is the next source symbol. A blank entry in PT indicates an error situation. A source string is assumed to be enclosed between the symbols '$\vdash$' and '$\dashv$'. Hence the parser starts with the sentential form $\vdash$ E $\dashv$.

Example 3.10 The sequence of predictions made by the parser of Fig. 3.7 for the source string

$$\vdash < id > + < id > * < id > \dashv$$

can be seen in Table 3.5 where *symbol* is the next source symbol. At every stage the parser refers to the table entry PT(nt_i, t_j) where nt_i is the leftmost nonterminal in the string and t_j is the next source symbol. Note that these steps are equivalent to the steps in Ex. 3.8.

Non-terminal	Source symbol			
	<id>	+	*	⊣
E	E ⇒ TE′			
E′		E′ ⇒ +TE′		E′ ⇒ ε
T	T ⇒ VT′			
T′		T′ ⇒ ε	T′ ⇒ *VT′	T′ ⇒ ε
V	V ⇒ <id>			

Fig. 3.7 LL(1) parser table

Table 3.5 LL parsing

Current sentential form	*Symbol*	*Prediction*
⊢ E ⊣	< id >	E ⇒ TE′
⊢ TE′ ⊣	< id >	T ⇒ VT′
⊢ VT′E′ ⊣	< id >	V ⇒ < id >
⊢ < id > T′E′ ⊣	+	T′ ⇒ ε
⊢ < id > E′ ⊣	+	E′ ⇒ + TE′
⊢ < id > + TE′ ⊣	< id >	T ⇒ VT′
⊢ < id > + VT′E′ ⊣	< id >	V ⇒ < id >
⊢ < id > + < id > T′E′ ⊣	*	T′ ⇒ *VT′
⊢ < id > + < id > * VT′E′ ⊣	< id >	V ⇒ < id >
⊢ < id > + < id > * < id >T′E′ ⊣	⊣	T′ ⇒ ε
⊢ < id > + < id > * < id >E′ ⊣	⊣	E′ ⇒ ε
⊢ < id > + < id > * < id > ⊣	–	–

The procedure for the construction of the parser table is described in the literature cited at the end of the chapter.

EXERCISE 3.2.1

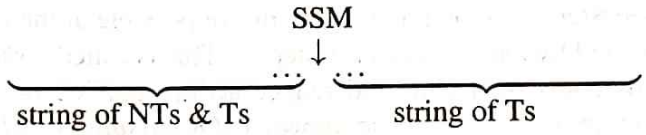
1. Study the operation of the recursive descent parser described in this section for the following input strings:
   (a) < id > + < id > * < id >
   (b) < id > * < id >< id >
   Verify that precise error indication is provided by the parser.
2. A recursive descent parser is to be developed for Grammar (1.3).

(a) What changes will you make in the grammar to make it suitable for recursive descent parsing?
(b) Code the parser procedures.
(c) Identify the procedures that perform error indication.

3. Compare and contrast recursive descent parsing with LL(1) parsing. What grammar forms are acceptable to each of these?

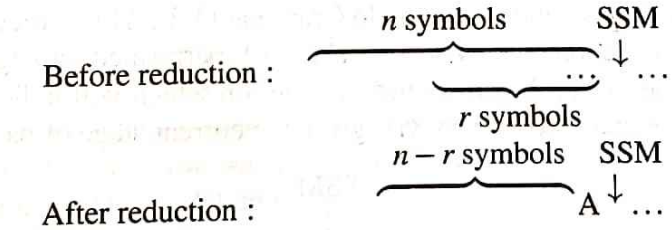
3.2.2 Bottom Up Parsing

A bottom up parser constructs a parse tree for a source string through a sequence of reductions. The source string is valid if it can be reduced to S, the distinguished symbol of G. If not, an error is to be detected and reported during the process of reduction. Bottom up parsing proceeds in a left-to-right manner, i.e. attempts at reduction start with the first symbol(s) in the string and proceed to the right. A typical stage during bottom up parsing is depicted as follows:



Reductions have been applied to the string on the left of SSM. Hence this string is composed of NTs and Ts. Remainder of the string, yet to be processed by the parser, consists of Ts alone.

We try the following naive approach to bottom up parsing: Let there be n symbols to the left of SSM in the current string form. We try to reduce some part of the string to the immediate left of SSM. Let this part have r symbols in it, and let it be reduced to the NT A. This reduction can be depicted as follows:



Since we do not know the value of r, we try the values $r = n$, $r = n-1$, ... $r = 1$. This approach is summarized in Algorithm 3.2.

Algorithm 3.2 (Naive bottom up parsing)

1. SSM := 2; n := 0;
2. $r := n$;
3. Compare the string of r symbols to the left of SSM with *all* RHS alternatives in G which have a length of r symbols.

4. If a match is found with a production A ::= α, then
   reduce the string of r symbols to the NT A.
   $n := n - r + 1$;
   Go to Step 2;
5. $r := r - 1$;
   If $r > 0$, go to Step 3;
6. If no more symbols exist to the right of SSM then
   if current string form = 'S' then
   exit with success
   else report error and exit with failure
7. SSM := SSM + 1;
   $n := n + 1$;
   Go to Step 2;

Thus the parser makes as many reductions as possible at the current position of SSM (see Step 5). When no reductions are possible at the current position, SSM is incremented by one symbol (see Step 7). This is called a *shift* action. The parsing process thus consists of shift and reduce actions applied in a left-to-right manner. Hence bottom up parsing is also known as *LR parsing* or *shift-reduce parsing*.

Algorithm 3.2 is unsatisfactory for two reasons. First, it is inefficient due to the large number of comparisons made in Step 3. Second, it performs reductions in a manner which may conflict with operator priorities. For example, consider Grammar (1.3) and the input string $< id > + < id > * < id >$. The correct parse tree is as shown in Fig. 3.8(b), where the symbols E, T, F and P stand for <*exp*>, <*term*>, <*factor*> and <*primary*>, respectively. The parser of Algorithm 3.2 builds the parse tree of Fig. 3.8(a) by erroneously reducing $< id > + < id >$ to an E. Further, since E * T does not appear on the RHS of any production, the parser indicates an error. However, the string is valid according to Grammar (1.3). This difficulty arises due to the premature reduction of $< id > + < id > \rightarrow$ E performed by Algorithm 3.2.

To overcome this problem, we need a criterion which will indicate when to perform a reduction and when not to, viz. given the current stage of parsing

SSM
↓
... αtγ ...

the criterion should indicate which of the following should be performed:

1. Apply a reduction involving α, and possibly some symbols to its left, to obtain an NT A.
2. Perform a shift action and now apply a reduction to some string δαt or σt to obtain an NT B. (Here σ is a substring of α.)
3. Perform one or more shift actions and apply a reduction to the string t... to obtain an NT C. Now apply a reduction to some string δαC or σC.

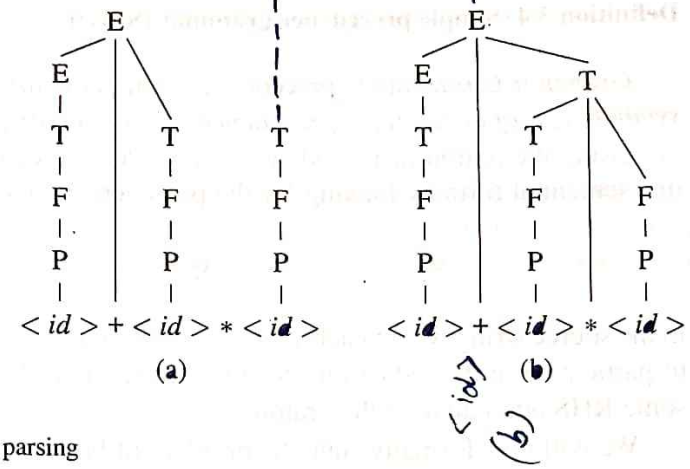


Fig. 3.8 Parse trees in bottom up parsing

In other words, given the sentential form ... αtγ..., the criterion should indicate whether to reduce (a substring of) α before, along with, or after some string t... appearing to its right. Such a criterion is provided by the notion of *precedence* of grammar symbols.

Simple precedence

Definition 3.3 (Simple precedence) *A grammar symbol* a precedes *symbol* b, *where each of* a, b *is a T or NT of G, if in a sentential form* ... a b ..., a *should be reduced prior to* b *in a bottom up parse.*

We use the notation a $\cdot>$ b for the words 'a precedes b'. Two possibilities arise if a does not precede b,

1. b precedes a, (i.e. b should be reduced prior to a), represented as b $\cdot>$ a, or
2. a and b have equal precedence (i.e. a and b should be reduced in the same step), represented as a $\doteq$ b.

Note that a precedence relation is defined between grammar symbols a and b only if a and b can occur side by side in a sentential form. We use this fact to obtain the precedence relation between a and b as follows:

1. Consider some sentential form ... a b ... .
2. Determine the sequence of derivations S $\stackrel{*}{\Rightarrow}$ ... a b ... such that the last derivation derives a string containing a or b or both. Number the derivations in this sequence from 1 to q.
3. Consider the derivation numbered q. This is the last derivation for obtaining ... a b ... . Let this be of the form A $\Rightarrow \beta$. Then $\beta \rightarrow$ A must be the first reduction in the bottom up parse of the string ... a b ... . Now
   (a) a $\cdot>$ b if $\beta \equiv$ ...a
   (b) a $<\cdot$ b if $\beta \equiv$ b...
   (c) a $\doteq$ b if $\beta \equiv$ ...a b... .

(3.7)

Definition 3.4 (Simple precedence grammar (SPG))

Grammar G is a simple precedence grammar *if for all terminal and nonterminal symbols* a, b *of G, a unique precedence relation exists for* a, b.

Using the notion of precedence in an SPG, we can identify a unit for reduction in a sentential form by looking for the precedence relations

$$<\cdot\, s_1 \doteq s_2 \doteq \ldots \doteq s_r \,\cdot>$$

in the source string where each s_i is a T or NT. Here s_1, s_r are the first and last symbols to participate in the reduction. Needless to say that the string $s_1 s_2 \ldots s_r$ would form some RHS alternative in the grammar.

We will now formally state the problem of bottom up parsing as follows:

Definition 3.5 (Simple phrase) α *is a* simple phrase *of the sentential form* $\ldots\alpha\beta\ldots$ *if there exists a production of the grammar* $A ::= \alpha$ *and* $\alpha \rightarrow A$ *is a reduction in the sequence of reductions* $\ldots\alpha\beta\ldots \rightarrow \ldots \rightarrow S$.

Definition 3.6 (Handle) *A* handle *of a sentential form is the left-most simple phrase in it.*

Example 3.11 E + T is not a simple phrase of the sentential form E + T * F according to grammar (1.3) since its reduction to E does not lead to the distinguished symbol. However, T * F is a simple phrase of the sentential form (see Fig. 3.8).

The deficiency of Algorithm 3.2 lies in its failure to identify simple phrases and handles. Algorithm 3.3 rectifies this deficiency.

Algorithm 3.3 (Bottom up parsing)

1. Identify the handle of the current string form.
2. If a handle exists, reduce it. Go to Step 1.
3. If the current string form = 'S' then exit with success
   else report error and exit with failure.

In practice, most grammars are not SPGs. Hence we will illustrate bottom up parsing for the class of operator grammars.

Example 3.12 Grammar (1.3) is not an SPG because of the following: In the string E + T * F, + $<\cdot$ T due to the production T ::= T * F. However, in E + T, + $\doteq$ T due to the production E ::= E + T.

Operator precedence grammars

In Section 1.4.1.1, we defined an operator grammar as a grammar none of whose productions contain two NTs appearing side-by-side. By induction, NTs cannot appear side-by-side in any sentential form of an operator grammar. Hence it is possible to ignore the presence of NTs and define precedence relations between operators for making parsing decisions. An operator precedence grammar (OPG) is an operator grammar in which the precedences between operators are unique. Such grammars typically arise in expressions.

Operator precedence

Precedence between *operators* a and b appearing in a sentential form ...aPb... where P is an NT or a null string, is termed *operator precedence.* The rules of procedure (3.7) can be used to determine operator precedences by considering the sentential form ...aPb... instead of the form ...ab.... Alternatively, it is possible to determine precedence relations from the productions of G as follows

1. $a \doteq b$ iff there exists a grammar production
   $C ::= \beta ab\gamma$ or $C ::= \beta aAb\gamma$.
2. $a \cdot> b$ iff there exist grammar productions
   $C ::= \beta Ab\gamma$ and
   $A ::= \pi a \mid \pi aD$
3. $a <\cdot b$ iff there exist grammar productions
   $C ::= \beta aB\gamma$ and
   $B ::= b\delta \mid Db\delta$

(3.8)

An *operator precedence matrix* (OPM) represents operator precedence relations between pairs of operators. The entry OPM(a, b) represents the precedence of operator a with respect to operator b in a sentential form ...aPb... , where P may be a null string.

Example 3.13 The precedence relations for the grammar

$$\begin{array}{lll} E & ::= & E + T \mid T \\ T & ::= & T * V \mid V \\ V & ::= & < id > \end{array}$$

are shown in Fig. 3.9. The entry OPM(+, *) = <·, represents the precedence of '+' as a left operator and '*' as the right operator of a pair of adjoining operators in a string, e.g. '+' and '*' of the string $< id > + < id > * < id >$. This precedence relation is obtained by applying procedure (3.7) to the sequence of derivations $E \Rightarrow E + T$, $T \Rightarrow T * V$, or by applying rule 3 of the rules (3.8) to the same productions.

There is a simpler way to obtain the precedence relations which works well for most grammars. This is based on the notions of associativity and relative priority of

LHS operator	RHS operator	
	+	*
+	·>	<·
*	·>	·>

Fig. 3.9

operators. A high priority operator always precedes a low priority operator appearing to its left or right. When two occurrences of an operator occupy adjoining positions of a string, the left occurrence precedes the right occurrence if the operator is left associative, else the right occurrence precedes the left occurrence.

Example 3.14 Consider the following grammar

$$\begin{aligned} S &::= \ \vdash E \dashv \\ E &::= \ E + T \mid T \\ T &::= \ T * V \mid V \\ V &::= \ < id > \mid (\ E\ ) \end{aligned} \tag{3.9}$$

Figure 3.10 shows the operator precedence matrix for the grammar. The entry OPM('+', '+') = ·> follows from the fact that '+' is left associative. The entries for '(' and ')' can be explained as follows: Consider an expression string

... b * (c + d) * e ...

Since a parenthesized expression is to be evaluated before its left and right neighbours, any operator would have <· relation with '(' appearing to its right, viz. '* <· '(' in the above string, and '(' would have a <· relation with any operator appearing to its right, viz. '(' <· '+'. Similarly, any operator would have ·> relation with ')' appearing to its right, while ')' would have a ·> relation with any operator appearing to its right. This is seen from the fact that '+' ·> ')', while ')' ·> '* in the above string. When '(' and ')' occur side by side, they would have $\doteq$ precedence with one another. In fact, this is the only instance of equal precedence in expressions. Same precedence relations would be obtained by applying procedure (3.7) or rules (3.8).

Note that '⊣' cannot be the LHS operator of a pair. Similarly '⊢' cannot be the RHS operator of a pair. Hence no rows and columns exist for these operators in OPM, respectively. The entries OPM('(', '⊣') and OPM('⊢', ')') are blank. Both these entries represent erroneous combinations of operators.

Operator precedence parsing

Example 3.15 Consider parsing of the string

$$\vdash < id > + < id > * < id > \dashv$$

new input

Stack op

LHS operator	RHS operator				
	+	*	(	)	⊣
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	≐	
)	·>	·>		·>	·>
⊢	<·	<·	<·		≐

Fig. 3.10 Operator precedence matrix

according to Grammar (3.9). The precedence relations from Figure 3.10 are marked below the string

Initial string : ⊢ <*id*> + <*id*> * <*id*> ⊣
<· <· ·>

The first <· is the precedence relation ⊢ <· +, the second <· represents + <· * etc. The handle is the string consisting of '* and its operands. After its reduction, the reduced string is

Reduced string : ⊢ <*id*> + ... ⊣
<· ·>

which leads to reduction of '+'.

Since operator precedence parsing ignores the NTs in a string, it is not easy to build a parse tree for a source string. However, an AST can be built easily. To develop an algorithm for AST building, we begin with some terminology and observations. We call the operator pointed to by SSM as the *current operator* and the operator to its left as the *previous operator*. From Ex. 3.15 it is clear that the previous operator must be reduced if previous operator ·> current operator, else a shift action must be performed. Thus end of the handle is known when previous operator ·> current operator. To find the beginning of the handle, we use the fact that the only instance of ≐ is '(' ≐ ')'. Hence the handle merely consists of the previous operator and its operands unless the previous operator is '('. If the previous operator is '(', the handle consists of '(..)'.

From these observations it is clear that apart from the current operator, only the previous operator needs to be considered at any point. A stack is therefore an appropriate data structure to use. An operator is pushed on the stack during a shift action and popped off during a reduce action. We can accommodate the right operand of an operator in the stack entry of the operator. The left operand, if present, would exist in the previous stack entry. We refer to the stack entry below the TOS entry as (TOS-1) entry, or TOSM entry for short.

Algorithm 3.4 (Operator Precedence Parsing)
Data structures

Stack : Each stack entry is a record with two fields, *operator* and *operand_pointer*.

Node : A *node* is a record with three fields, *symbol*, *left_pointer*, and *right_pointer*.

Functions

newnode (*operator, l_operand_pointer, r_operand_pointer*) creates a *node* with appropriate pointer fields and returns a pointer to the node.

1. TOS := SB − 1; SSM := 0;
2. Push '⊢' on the stack.
3. SSM := SSM + 1;
   If current source symbol is an operator, then go to Step 5.
4. *x* := *newnode* (*source symbol, null, null*);
   TOS.*operand_pointer* := *x*;
   Go to Step 3;
5. While TOS operator $\dot{>}$ current operator
   x := *newnode* (TOS *operator*, TOSM.*operand_pointer*, TOS.*operand_pointer*);
   Pop an entry off the stack.
   TOS.*operand_pointer* := *x*;
6. If TOS operator $<\cdot$ current operator, then
   Push the current operator on the stack.
   Go to Step 3;
7. If TOS operator $\doteq$ current operator, then
   If TOS operator = '⊢', then exit successfully.
   If TOS operator = '(', then
   temp := TOS.*operand_pointer*;
   Pop an entry off the stack.
   TOS.*operand_pointer* := *temp*;
   Go to Step 3;
8. If no precedence defined between TOS operator and current operator then
   Report error and exit unsuccessfully.

Example 3.16 Consider parsing of the string

$$\vdash < id >_a + < id >_b * < id >_c \dashv$$

according to grammar (3.9), where $< id >_a$ represents a. Figure 3.11 shows steps in its parsing. Figures 3.11(a)-3.11(c) show the stack and the AST when current operator is '+', '*' and '⊣', respectively. In Fig. 3.11(c), TOS operator $\dot{>}$ current operator.

This leads to reduction of '*' (see Step 5 of Algorithm 3.4). Figure 3.11(d) shows the situation after the reduction. The new TOS operator, i.e. '+', -> current operator. This leads to reduction of '+' as shown in Fig. 3.11(e).

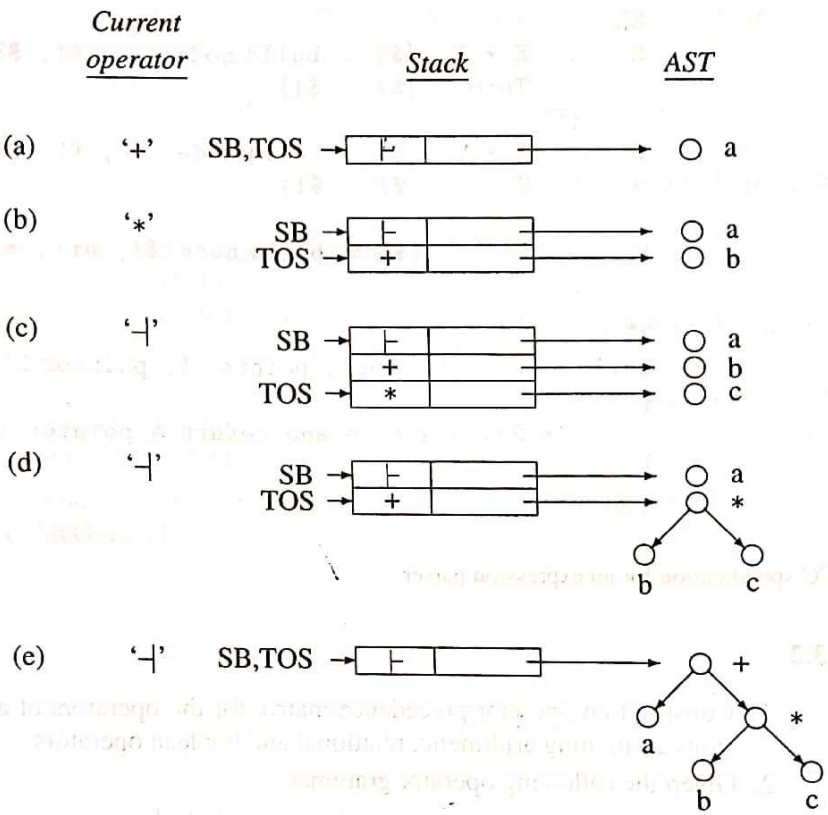


Fig. 3.11 Operator precedence parsing using a stack

LALR parsing

Practical LR parsers are table driven in nature, i.e. the parsing actions are governed by the table entry corresponding to the current state of the parser and the next few source symbols. The parser tables are typically generated using a parser generator. The information in the LR parsing tables and the operation of the LR parsers are described in Dhamdhere(1983) and Aho, Sethi, Ullman (1986). Here, we shall see a simple example containing specification of grammar productions and semantic actions in a YACC like manner. Note that YACC generates a parser which uses the LALR(1) variant of LR parsing.

Example 3.17 Figure 3.12 shows the YACC like specification for an expression parser. The parser builds an AST analogous to the parser of Ex. 3.16. As described in Section 1.5.2, the semantic action specified for a production is executed when a reduction is performed according to that production. The `build_node` routine builds a node

and returns a pointer to it. The node description consists of three parts—node label and pointers to its left and right child nodes. The pointer returned by `build_node` is remembered as the attribute of the left hand side symbol of a production.

```
%%
E   :  E + T   {$$ = build_node('+', $1, $3)}
    |  Term    {$$ = $1}
    ;
T   :  T * V   {$$ = build_node('*', $1, $3)}
    |  V       {$$ = $1}
    ;
V   :  id      {$$ = build_node($1, nil, nil)}
    ;

%%
build_node(node_label, pointer_1, pointer_2);
{
     /* Build a node and return a pointer to it */
}
```

Fig. 3.12 YACC specification for an expression parser

EXERCISE 3.2

1. Construct an operator precedence matrix for the operators of a grammar for expressions containing arithmetic, relational and boolean operators.
2. Given the following operator grammar

$$\begin{array}{lll} S & ::= & \vdash A \dashv \\ A & ::= & VaB \mid \varepsilon \\ B & ::= & VaC \\ C & ::= & VbA \\ V & ::= & <id> \end{array}$$

   (a) Construct an operator precedence matrix for the operators of the grammar.
   (b) Give a bottom up parse for a string containing nine <*id*> symbols.
3. An **if** statement may be written with or without an **else** part. Given the following operator grammar for **if**

$$\begin{array}{rcl} <\textit{if\_stmt}> & ::= & \textbf{if} <\textit{exp}> \textbf{ then} <\textit{stmt}> \textbf{ else} <\textit{stmt}> \\ & & \mid \textbf{if} <\textit{exp}> \textbf{ then} <\textit{stmt}> \\ <\textit{assignment}> & ::= & <\textit{var}> := <\textit{exp}> \\ <\textit{stmt}> & ::= & <\textit{assignment}> \mid <\textit{if\_stmt}> \end{array}$$

   where **if**, **then** and **else** are operators, find whether the operator precedence relations in this grammar are unique.

BIBLIOGRAPHY

Aho, Sethi and Ullman (1986) discuss automatic construction of scanners. Lewis, Rosenkrantz and Stearns (1976), Dhamdhere (1983) and Aho, Sethi and Ullman (1986) discuss parsing techniques in detail.

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.
2. Barrett, W.A. and J.D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.
3. Dhamdhere, D.M. (1983): *Compiler Construction – Principles & Practice*, Macmillan India, New Delhi.
4. Fischer, C.N. and R.J. LeBlanc (1988): *Crafting a Compiler*, Benjamin/Cummings, Menlo Park, California.
5. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.
6. Lewis, P.M., D.J. Rosenkrantz, and R.E. Stearns (1976): *Compiler Design Theory*, Addison-Wesley, Reading.
7. Tremblay, J.P. and P.G. Sorenson (1984): *The Theory and Practice of Compiler Writing*, McGraw-Hill.