MODULE -4

Syntax Analysis

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation,

Grammars offer significant benefits for both
language designers and compiler writers.
• A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
• From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program.
As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.
• The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
• A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

The Role of the Parser

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names
can be generated by the grammar for the source language

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parse

There are three general types of parsers for grammars: universal, top-down, and bottom-up. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar (see the bibliographic notes). These general methods are, however, too inefficient to use in production compilers.
The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, LL and LR grammars,

are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars;

Syntax Error Handling

Common programming errors can occur at many different levels.

• *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier e l i p s e S i z e instead of e l l i p s e S i z e — and missing quotes around text intended as a string.

• *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, "{" or " } . " As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

• *Semantic* errors include type mismatches between operators and operands. An example is a r e t u r n statement in a Java method with result type void.

• *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect

an error as soon as possible; that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the *viable-prefix property,* meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue.

Error-Recovery Strategies

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability. The simplest approach is for the parser to quit with an informative error message when it detects the first error. Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information. If errors pile up, it is better for the compiler to give up after exceeding some error limit than to produce an annoying avalanche of "spurious" errors

Panic-Mode Recovery

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of *synchronizing tokens* is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. The compiler designer

must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking

it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop

Phrase-Level Recovery

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer. Of course, we must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.

Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Productions

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

Global Correction

Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string *x* and grammar *G,* these algorithms will find a parse tree for a related string *y,* such that the number of insertions, deletions, and changes of tokens required to transform *x* into *y* is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.

Context-Free Grammars

Grammars were introduced in Section 2.2 to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable *stmt* to denote statements and variable *expr* to denote expressions, the production

stmt —> if (*expr*) *stmt* else *stmt*

The Formal Definition of a Context-Free Grammar

context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. *Terminals* are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just

the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. In (4.4), the terminals are the keywords if and else and the symbols "(" and ") ."
2. *Nonterminals* are syntactic variables that denote sets of strings. In (4.4), *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the *start symbol,* and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each *production* consists of:
(a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.
(b) The symbol -K Sometimes : : = has been used in place of the arrow.
(c) A *body* or *right side* consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Derivations
The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.

Derivation Example

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

OR

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen.

2. In *rightmost* derivations, the rightmost nonterminal is always chosen;

Analogous definitions hold for rightmost derivations. Rightmost derivations are sometimes called *canonical* derivations

Left-Most and Right-Most Derivations

Left-Most Derivation

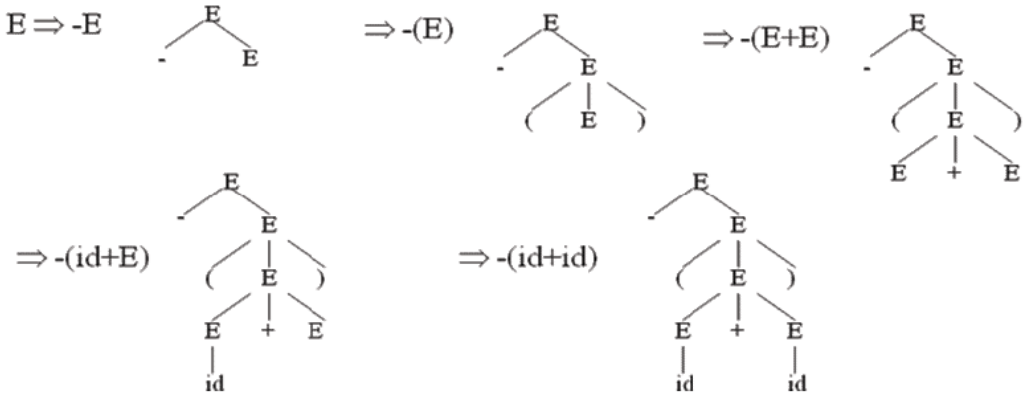
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$
$$\text{lm} \qquad \text{lm} \qquad \text{lm} \qquad \text{lm} \qquad \text{lm}$$

Right-Most Derivation

$$E \xrightarrow[rm]{} -E \xrightarrow[rm]{} -(E) \xrightarrow[rm]{} -(E+E) \xrightarrow[rm]{} -(E+id) \xrightarrow[rm]{} -(id+id)$$

Parse Trees and Derivations

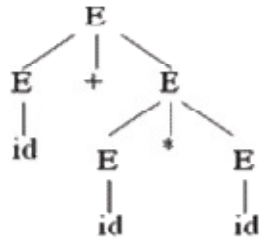
A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal *A* in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this *A* was replaced during the derivation.



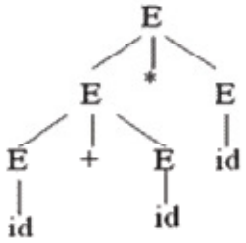
Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous.* Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



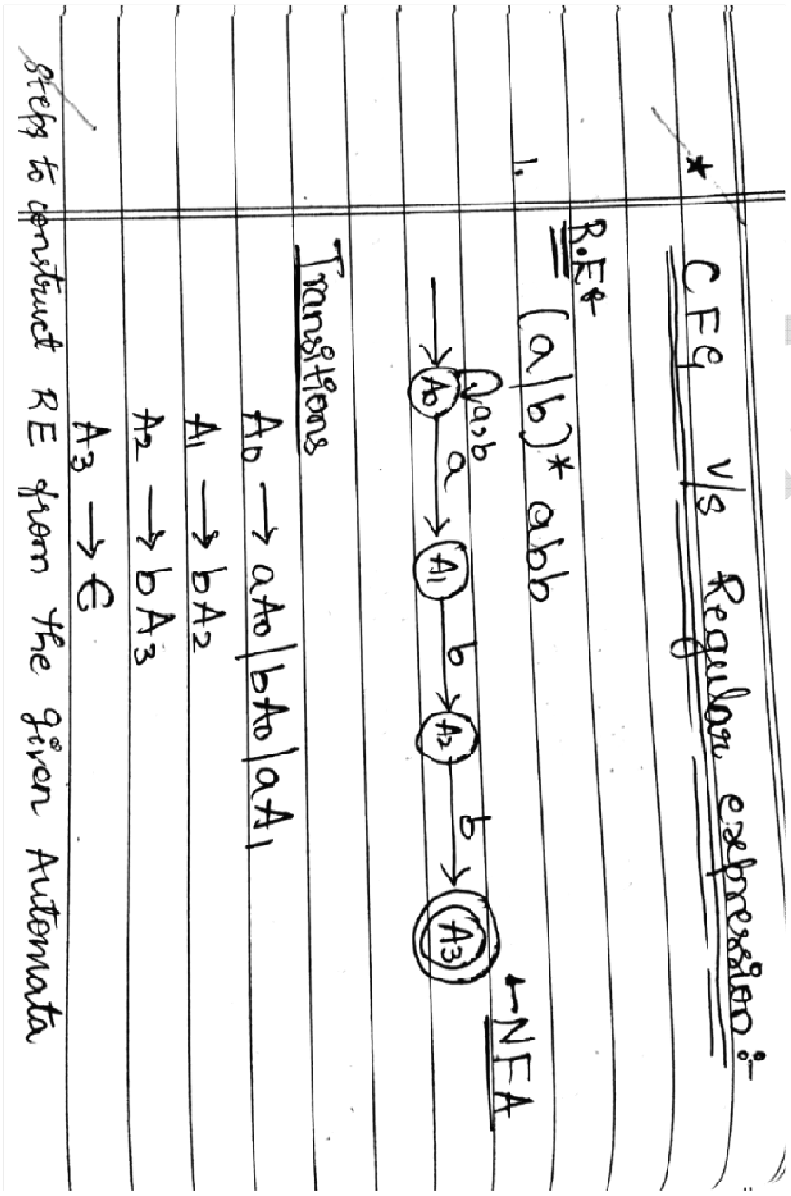
$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



Context-Free Grammars Versus Regular Expressions

Before leaving this section on grammars and their properties, we establish that grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.

For example, the regular expression **(a|b)*abb** and the grammar



describe the same language, the set of strings of a's and 6's ending in *abb.*

We can construct mechanically a grammar to recognize the same language as a nondeterministic finite automaton (NFA). The grammar above was constructed from the NFA , using the following construction:

1. For each state *i* of the NFA, create a nonterminal *Ai.*
2. If state *i* has a transition to state *j* on input a, add the production *Ai* —>

aAj. If state *i* goes to state *j* on input *e,* add the production *Ai* —• *Aj.*
3. *Hi* is an accepting state, add *Ai* —> *e.*
4. If *i* is the start state, make *Ai* be the start symbol of the grammar.

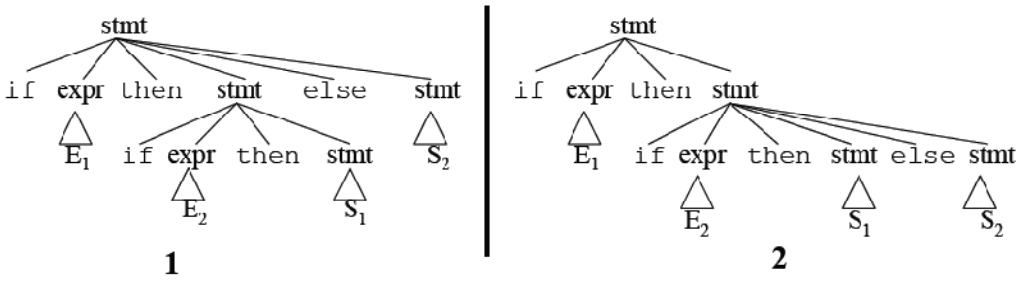
Lexical Versus Syntactic Analysis
E everything that can be described by a regular
expression can also be described by a grammar. We may therefore reasonably ask: "Why use regular expressions to define the lexical syntax of a language?" There are several reasons.

1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Eliminating Ambiguity
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "danglingelse" grammar:

```
stmt → if expr then stmt |
       if expr then stmt else stmt  | otherstmts
```

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



We prefer the second parse tree (else matches with closest if).
• So, we have to disambiguate our grammar to reflect this choice.
• The unambiguous grammar will be:

stmt -> matchedstmt | unmatchedstmt

matchedstmt → if expr then matchedstmt else matchedstmt | otherstmts
unmatchedstmt → if expr then stmt |
if expr then matchedstmt else unmatchedstmt

Ambiguity – Operator Precedence

• Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

E → E+E | E*E | E^E | id | (E)

disambiguate the grammar

precedence: ^ (right to left)

* (left to right)

+ (left to right)

E → E+T | T

T → T*F | F

F → G^F | G

G → id | (E)

Left Recursion

• A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

$$A \overset{+}{\Rightarrow} A\alpha \qquad \text{for some string } \alpha$$

• Top-down parsing techniques **cannot** handle left-recursive grammars.

• So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

• The left-recursion may appear in a single step of the derivation (*immediate leftrecursion*), or may appear in more than one step of the derivation.

To eliminate left recursion use the equations

A→βA'

A'→αA'|ε

Eliminate Left-Recursion – Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
  - **for** i **from** 1 **to** n **do** {
    - **for** j **from** 1 **to** i-1 **do** {
      replace each production
      $A_i \rightarrow A_j \gamma$
      by
      $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$
      where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$
      }
    - eliminate immediate left-recursions among A_i productions
  }

Left-Factoring

A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the

A'-> $\beta_1 | \beta_2$

In general,

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

when processing α we cannot know whether expand

A to $\alpha\beta_1$ or

A to $\alpha\beta_2$

But, if we re-write the grammar as follows

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring – Algorithm

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Recursive-Descent Parsing

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

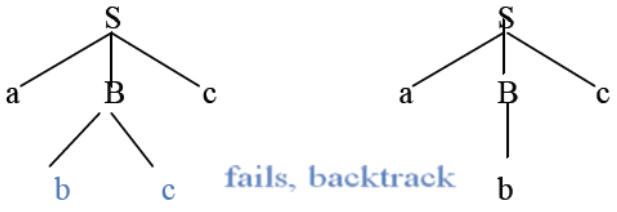
General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.

```
void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

- Backtracking is needed.
- It tries to find the left-most derivation.

S → aBc
B → bc | b

input: abc



FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW,** associated with a grammar *G.* During topdown parsing, **FIRST** and **FOLLOW** allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by **FOLLOW** can be used as synchronizing tokens.

Define ***FIRST(a),*** where ***a*** is any string of grammar symbols, to be the set of terminals that begin strings derived from ***a.***

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST$(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST$(Y_1), \ldots,$ FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST$(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

Define ***FOLLOW(A),*** for nonterminal *A,* to be the set of terminals *a* that can appear immediately to the right of *A* in some sentential form;

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Identify FIRST and FOLLOW by eliminating left recursion in the grammar

E → E + T | F ; T → T * F | F; F → (E) | id

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow id \mid (E)$$

$\Downarrow$ eliminate immediate left recursion

$$E \rightarrow T\,E'$$
$$E' \rightarrow +T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *F\,T' \mid \varepsilon$$
$$F \rightarrow id \mid (E)$$

	FIRST	FOLLOW
E	**id,(**	**$,)**
E'	**+,ε**	**$,)**
T	**id,(**	**+,$,)**
T'	***,ε**	**+,$,)**
F	**id,(**	***,+,$,)**

Predictive Parser

• When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

Non-Recursive Predictive Parsing -- LL(1) Parser

• Non-Recursive predictive parsing is a table-driven parser.
• It is a top-down parser.
• It is also known as LL(1) Parser.

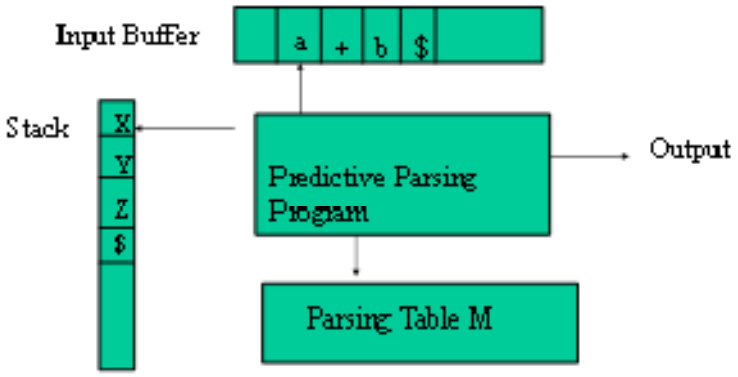


Fig: Model Of Non-Recursive predictive parsing

LL(1) grammar

The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets. The third condition is equivalent to stating that if ϵ is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ϵ is in FIRST(α).

LL(1) Parser
input buffer
– our string to be parsed. We will assume that its end is marked with a special symbol $.
output
– a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.
stack
– contains the grammar symbols
– at the bottom of the stack, there is a special end marker symbol $.
– initially the stack contains only the symbol $ and the starting symbol S. $S
 initial stack

– when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

– a two-dimensional array M[A, a]
– each row is a non-terminal symbol
– each column is a terminal symbol or the special symbol $
each entry holds a production rule.

Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following

1. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add A to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to$ $M[A, \$]$ as well.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
    if ( X is a ) pop the stack and advance ip;
    else if ( X is a terminal ) error();
    else if ( M[X, a] is an error entry ) error();
    else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
        output the production X → Y1 Y2 ··· Yk;
        pop the stack;
        push Yk, Yk−1, ..., Y1 onto the stack, with Y1 on top;
    }
    set X to the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

For grammar

E->TE’

E’->+TE’|ε

T->FT’

T’->*FT’| ε

F->(E)|id

LL(1) parsing table is

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Parsing moves for id+id

Matched	Stack	Input	Action
	E$	id+id$	
	TE'E$	id+id$	o/p E→TE'
	FT'E'E$	id+id$	o/p T→FT'
id	idT'E'E$	id+id$	o/p F→id
id	T'E'E$	+id$	Match id
	E'$	+id$	o/p T'→ϵ
	+TE'$	+id$	o/p E'→+TE'
id+	TE'$	id$	Match +
	FT'E'$	id$	o/p T→FT'
	idT'E'$	id$	o/p F→id
id+id	T'E'$	$	Match id
	E'$	$	o/p T'→ϵ
	$	$	o/p E'→ϵ
			Accept
id+id$	$	$	Match $

Examples

1.

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \varepsilon$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$

So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$

- Eliminate the immediate left-recursion in S

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \varepsilon$

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \varepsilon$

Bottom up parsing

Introduction

Given a grammar and a sentence belonging to that grammar, if we have to show that the given sentence belongs to the given grammar, there are two methods.

1. Leftmost – Derivation
2. Rightmost – Derivation

In left most derivation we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence. The parsing methods based on this are known as top-down methods and we have already discussed this in previous chapter.

In rightmost derivation we start from the given sentence and using various productions, we try to reach the start symbol. The parsing method based on this concept are known as bottom-up method and are generally used in compilers generated using various tools like LEX and YACC. The bottom up parsing methods is more general and efficient. They can find syntactic error as soon as they occur.

We will be studying the following methods of parsing in this chapter

1. LR(0) Parsing
2. SLR (1) Parsing
3. LALR (1) Parsing Methods

Where LR stands for Left to Right scan and Rightmost derivation in reverse. SLR Stands for simple LR and LALR for Look Ahead LR and 1 in brackets indicate the number of look aheads.

LR (0) stands for no look ahead and LR (1) for one look ahead symbol. In general there can be LR (K) parsers with 'K' lookahead symbols.

Bottom up parsing

Bottom-up parsers in general use explicit stack. They are also known as shift reduce parsers.

Example: Consider a grammar S → a S b c and a sentence a a c b b. The parser put the sentence to be recognized in the input buffer appended with end of input symbol $ and bottom of stack has also $.

Step	Start	Input buffer	Action
1	$	a a c b b $	Shift
2	$ a	a a b b $	Shift
3	$ a a	c b b $	Shift
4	$ a a c	b b $	Reduce using S → c
5	$ a a S	b b $	Shift
6	$ a a S b	b $	Reduce using S → a S b
7	$ a S	b $	Shift
8	$ a S b	$	Reduce using S → a S b
9	$ S	$	Accept.

The parser consults a table indexed by two parameters to be discussed later. The parameters what is on the top of stack and input character pointed by input buffer pointer. Assume for the time being that the table tells the parser to do one of the following activities.

1.	Shift	–	Shift the symbol to stack
2.	Reduce	–	Pop some symbols in the stack and replace by a non-terminal
3.	Accept	–	Input is syntactically correct, therefore accept
4.	Error	–	Input is syntactically not correct.

The table formation will be discussed later. From the figure above, after 3 shifts the table tells the parser to reduce in step-4 that is pop c and replace by (i.e., push) 'S'. In the 5th Step again shift is executed. In 6th step the parser is told to pop 3 symbols and replace it by S, in 7th step again parser is told to shift. In 8th step the parser pops 3 symbols & replaces it by S i.e., reduce action takes place. When top of stack is start symbol i.e., S in this case and input buffer is empty i.e., input buffer is pointing to $ (this indicates there is no more input available). The parser is told to carryout accept action. The parser is able to recognize that aacbb is indeed a valid sentence of the given grammar. We shall see later on how shift reduce and accept actions are indicated.

Example:

Consider another example of parsing using the grammar

S → a A c B e
A → A b b
B → d

Recognize abbcde

Start	Input buffer	Action
$	a b b c d e $	Shift
$ a	b b c d e $	Shift
$ a b	b c d e $	Reduce A → b
$ a A	b c d e $	Shift
$ a A b	c d e $	Reduce A → A b
$ a A	c d e $	Shift
$ a A c	d e $	Shift
$ a A c d	e $	Reduce B → d
$ a A c B	e $	Shift
$ a A c B e	$	Reduce S → A c B e
$ S	$	Accept

Rightmost Derivation

S ⇒a A c B e
⇒a A c d e replace B → d
⇒ a A b c d e replace A → A b
⇒ a b b c d e replace A → b

Reduction done in the shift reduce parses is exactly the reverse order of rightmost derivation replace statements.

LR (0) Items: An LR (0) Items, is any production with a dot on the right hand side of a production.

Example 1:

S → a S b has the following LR (0) Items

S → • a S b
S → a • S b
S → a S • b
S → a S b •

As we can see that there are 3 symbols on the right hand side of the production. Therefore there are four positions where in a dot can be put, in general if there are n characters on RHS of a production there will be n + 1 LR (0) Items from that production.

Example 2: Find the LR (0) Items

E → E + T

Solution: LR (0) Items are

E → • E + T
E → E • + T
E → E + • T
E → E + T •

Augmentation of grammar

A grammar must be augmented before an LR parser could be constructed. Augmentation is nothing but adding a new production, which is not already present in the grammar that derives the start symbol.

Example:

Given grammar S → a S b c

Augmented grammar

$S^1 \rightarrow S$
S → a S b c

Where S^1 is a new start symbol which derives S i.e., the start symbol of the grammar. Now S^1 becomes the new start symbol. This is required to uniquely identify the accept state.

There is a relation between LR (0) Items & Finite Automata. It is possible to construct a DFA to recognize all viable prefixes of a given grammar. Viable prefixes are right-sentential forms that do not contain any symbols to the right of the handle. Handle is any RHS of a production.

For Example:

S → a S b c Handles

are aSb and c

Viable prefixes are – a a c because c is handle
–a a S b because a c b is a handle
–a a a S b because a c b is a handle

To start the construction of DFA we need to augment the grammar as follows

$S^1 \rightarrow S$
S → a S b c

Then, list all the Items

$S^1 \rightarrow$ • S
S → • a S b
S → a • S b
S → a S • b
S → a S b •
S → • c
S → c •

There are seven LR (0) Items in the above grammar. These LR (0) Items form the state S of the DFA. We need to put $S^1 \rightarrow \bullet S$ always is the start state as follows. This state is called as state 0

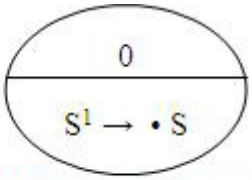


Fig 5.1 LR (0) of Augmented Production

Here we need to take closure of LR (0) Items. That is whenever a dot appears before a non-terminal we need to add to this set the first LR (0) Item derived by S i.e., $S \rightarrow \bullet a\ S\ b$ and $S \rightarrow \bullet c$. This is recursive definition. Now there are dots before only terminal a and c no closure is needed. Therefore state 0 of DFA has 3 Items.

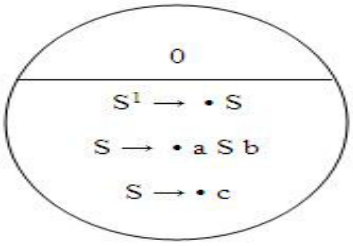


Fig 5.2 LR(0) of S^1 & S

Now we have to advance each of LR (0) Items in state 0. This is possible by assuming input is S, a and c (Note S cannot appear as input, only terminals can appear as input). The resulting states are

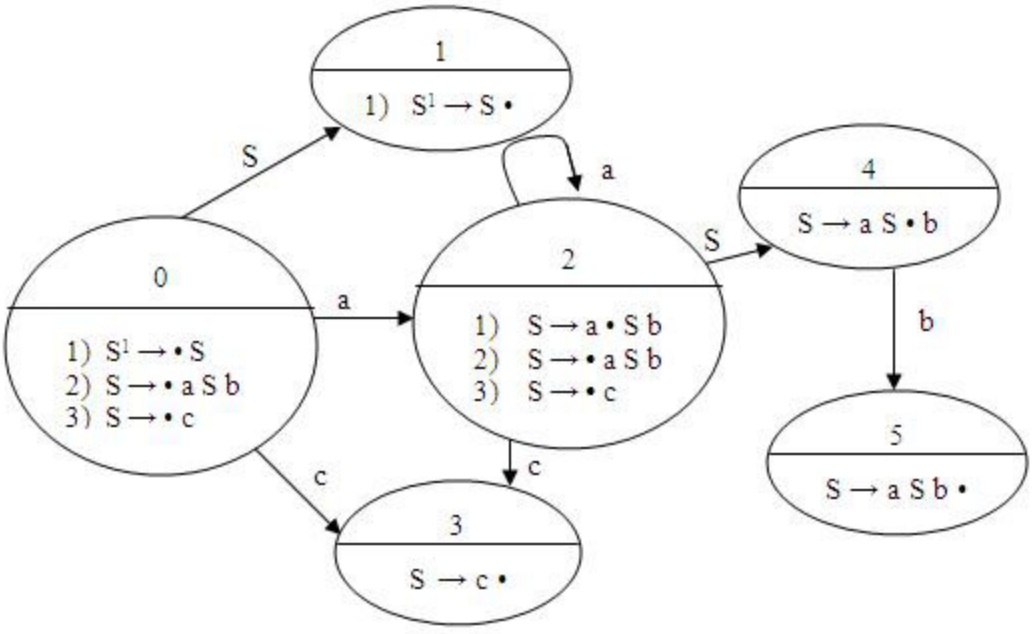


Fig 5.3 DFA of LR(0) Items

The resulting states are 1, 2 & 3. When dot moves to the end of RHS of a production and there are no other LR (0) Items, such states are called reduce states. Now we have 1, 3 & 5 are reduced states, because dot has moved to the rightmost position and no other LR (0) Items in those states. Whereas 0 and 2 state has three LR (0) Items, we say that, if 'a' occurs in state 0 then action to is shift and resulting state is 2. Similarly if 'c' occurs in state 0 then action is shift and goto state 3.

Consider state 2. Here there are three LR (0) Items, because of the closure of production – 1 now to progress from state of three are 3. Possibilities i.e., S, a and c. If 'c' occurs it goes to state 3 and 'a' occurs it go to itself as shown. When S occurs in state 2 it goes to state 4. By taking closure in state 4 no new Items are added. In state 4 when b comes it goes to state 5. Since the dot has reached the last position in state 5, no further states are generated.

We conclude that with 6 states (i.e., state 0 to 5) we can recognize all viable prefixes i.e.,

Viable Prefixes are

1) S
2) a a* c
3) a a* S
4) a a* S b
5) c

Since we need to recognize five viable prefixes we need to have 6 states.

Classification of states of DFA

States of the DFA are classified as following:

Accept State: The state which recognizes $S^1 \rightarrow S$. Note this is also a special reduce state

Reduce State: Where LR (0) Items have their dot placed on the extreme right position example 3 and 5

Shift State: Where all LR (0) Items have their do not in the extreme right position.

Shift/Reduce State: Mix of shift & reduce Items as defined above.

Constructing parsing table SLR (1)

The parsing table has rows indexed by state of DFA. Whereas columns are indexed by terminals (including $) and non-terminals (except S^1). The portion of table having terminal has index is known as ACTION Part and that indexed by non-terminals is known as GOTO portion as shown below.

States	ACTION				GOTO
	a	b	c	$	S
0	S2		S3		1
1				Acc	
2	S2		S3		4
3					
4					
5					

Table 5.4 Table for SLR (1) Parser

The filling of table is simple. In state 0 on 'a' it goes to state 2. Therefore it is a shift action, hence an entry S2. Similarly on 'c' it goes to state 3, again shift action, hence S3. On 'S' it goes to state '1' since 'S' is non-terminal it is filled only with state without any action.

Similarly all the other states are filed. Since $ is always follows any end of sentence, since we are putting ourselves to recognize the end of input and also when $S^1 \rightarrow S.$ comes. It is treated as accept state, as this state is unique.

Example: Consider the following grammar

$S^1 \rightarrow S$ 1) $S \rightarrow (S)\ S$
$S \rightarrow (S)\ S$ 2) $S \rightarrow$
$S \rightarrow$

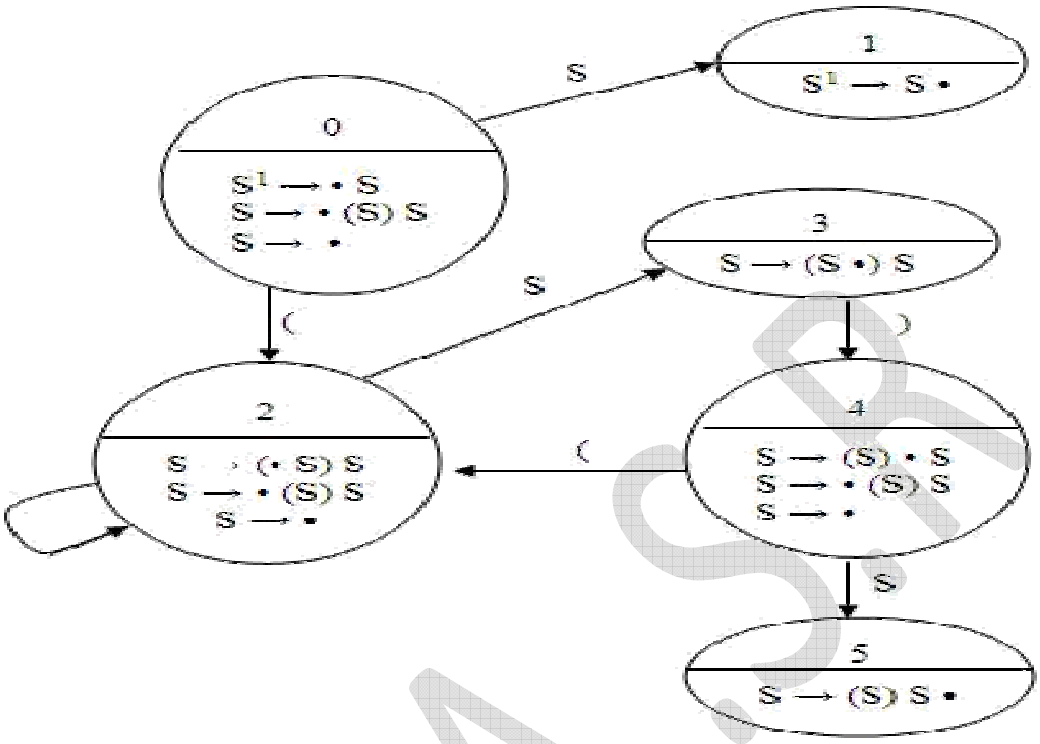


Fig 5.4 DFA for LR (0) Items

FOLLOW(S)={), $ }

State	Action			Goto
	(	)	S	S
0	S2	r2	r2	1
1			Acc	
2	S2	r2	r2	3
3		S4		
4	S2	r2	r2	5
5		r1	r1	

Table 5.5 SLR Parsing table

Parsing using the above table. Input ( )

	Input	Action
$ 0	( ) $	S2
$ 0 ~~C 2~~	) $	R2 : S → €
$ 0 C 2 5 3	) $	S4
$ 0 (2 5 3) 4	$	r2 : S → €
$ 0 ~~(2 5 3) 4 5 5~~	$	r1 : S → (S) S
$ 0 S 1	$	Accept
$ 0		

Example: Consider the following grammar.

A → (A) a ,

Augment the grammar

0) $A^1 \rightarrow A$

1) A → (A)

2) A → a

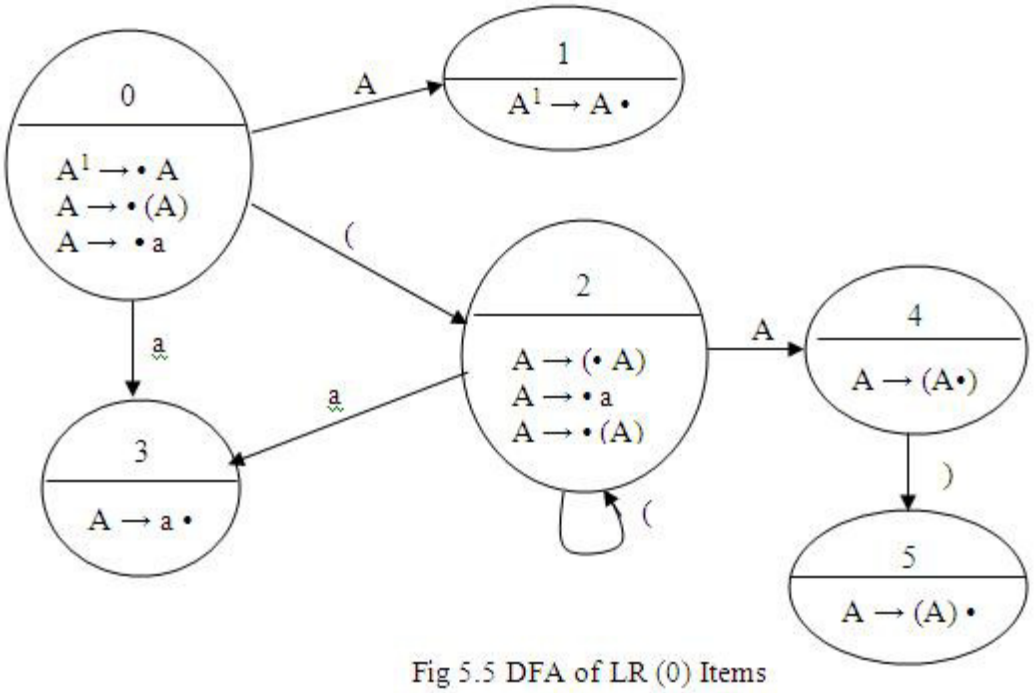


Fig 5.5 DFA of LR (0) Items

Follow (A) = {), $ }

State	Action				Goto
	a	(	)	$	A
0	S3	S2			1
1				acc	
2	S3	S2			4
3			r2	r2	
4			S5		
5			r1	r1	

Table 5.6 SLR Parsing table

Example: Consider the grammar S → (L) a , L → L, S S, Augment the grammar

S1 → • S
S → • (L) a
L → • L, S S

1) S → (L)
2) S → a
3) L → L, S
4) L → S

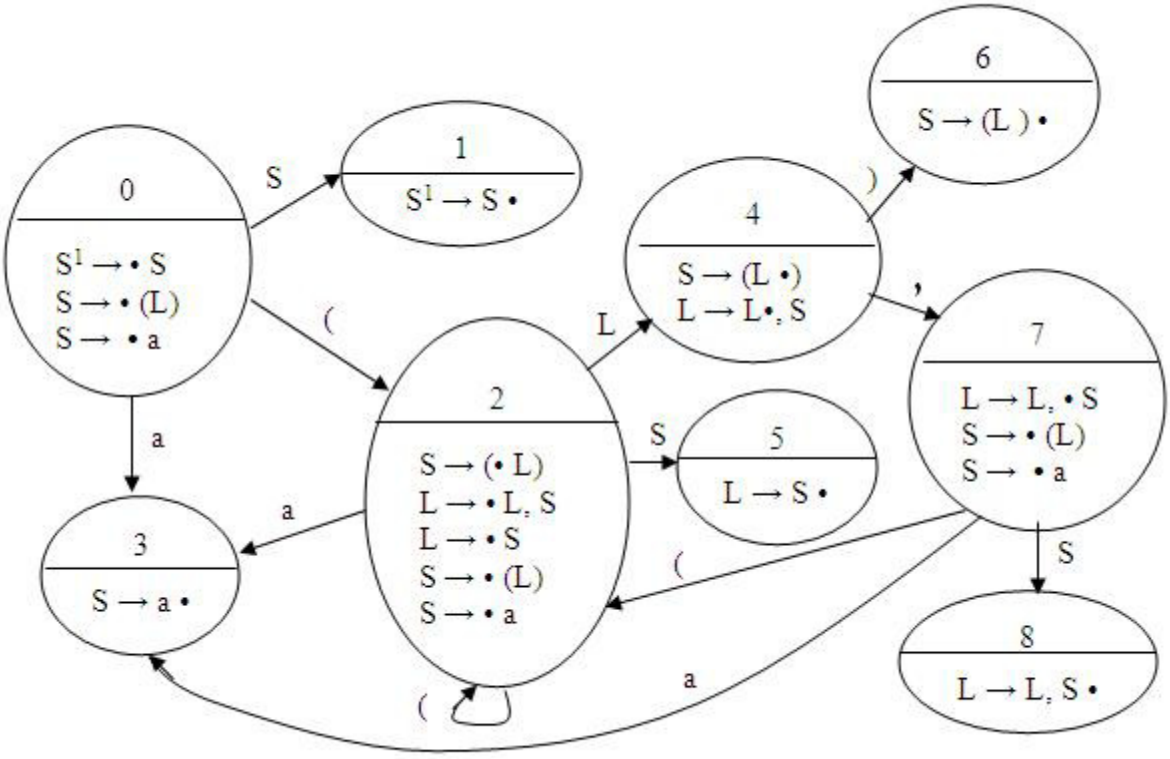


Fig 5.6 DFA of LR (0) Items

Follow (L) = { , ,) }

Follow (S) = {), $}

State	Action					Goto	
	a	(	)	'	S	S	L
0	S3	S2				1	
1					acc		
2	S3	S2				5	4
3			r2	r2	r3		
4			S6	S7			
5			r4	r4			
6			r1	r1	r1		
7	S3	S2				8	
8			r3	r3	r3		

Table 5.7 SLR Parsing table

We have seen the construction SLR (0) parser. This parser is able to parse most of the programming constructs.

Construction of LR(0) Parser:

LR (0) Parser is not a popular parser and it can be constructed following the procedure used for SLR (1) parser. The difference between LR(0) and SLR (1) parser is LR(0) parser actions, do not depend on the look ahead.

Consider the SLR (0) parser constructed for the simple grammar

S → a S b c

Procedure followed earlier is used to construct the DFA shown in fig 5.3. There are 6 states Action of each state depends whether it is a reduce state or shift state. In fig. 5.3 we see that there are

1) Reduce States 1, 3 and 5
2) Shift states 0, 2 & 4

Here the shift or reduce actions are done without looking at the input symbol. The table will have the following entries.

State	Action	Rule	Input			GOTO
			a	b	c	
0	Shift	- - - - - - -	2	-	3	1
1	Reduce	S1 → S	-	-	-	-
2	Shift	- - - - - - -	2	-	3	4
3	Reduce	S → c	-	-	-	-
4	Shift	- - - - - - -		5		
5	Reduce	S → a S b	-	-	-	-

Construction of the table is same as followed in SLR (1) parser and there is no need to calculate follow symbol for all the reduce states.

Now let use the above parser table to check the input a c b

Step	Parsing Stack	Input	Action
1	$ 0	a c b $	Shift
2	$ 0 a 2	c b $	Shift
3	$ 0 a 2 ~~c 3~~	b $	Reduce S → c
4	$ 0 a 2 S 5	b $	Shift
5	$ 0 ~~a 2 S 4 b 5~~	$	Reduce S → a S b
6	$ 0 S 1	$	Accept

To start with put starting state i.e., state 0 in parsing stack and input 'acb' appended by $ in input. Stack has stage '0', which is a shift state therefore shift a in to the stack. 0 state on a goes to state 2 as seen in the table. Therefore push state 2 on to the stack. State 2 is again shift state, therefore shift character c on to the stack. State 2 on character 'c' goes to state 3, therefore push state 3 on to the stack. This procedure continues i.e., state on the top of the stack decides the action to be gone without looking at the input symbol. State 1 is a reduce state but it is special because occurs only when we are ready to accept the input. As we see there is not much change from SLR (1) parsing.

Limitations of SLR (1)

SLR(1) as we have seen in simple, yet powerful to parse almost all languages constructs. But this fail in two cases.

1. Shift/Reduce conflict: When a state has both shift and reduce Items are present and parser indicates that both the operations to be done on a particular symbol. At this point parser will not be able to resolve and if fails.

2. Reduce/Reduce conflict: When a particular state has LR(1) Items which are both reduce Items which are both reduce Items i.e., the dot in LR (0) Items has reached rightmost position. And both are to be reduced on the same symbol.

Let us consider the parser to understand shift/reduce conflict.

$S^1 \rightarrow S$
$S \rightarrow L = R$
$S \rightarrow R$
$L \rightarrow A R$
$L \rightarrow i d$
$R \rightarrow L$

Consider the SLR(1) Parser

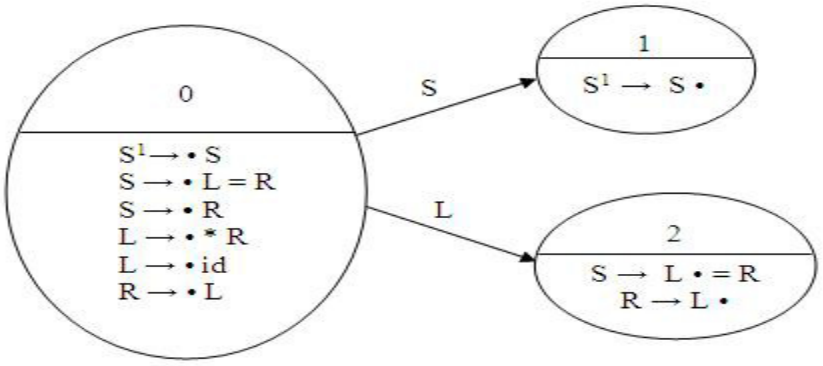


Fig 5.7 DFA for LR (0) Items

Let us analyze the state 2. There are two Items

1. S → L • = R
2. R → L •
3.

Item–1 is shift them and Item–2 is reduce Item of we calculate the follow symbols of R = { = }

There is a conflict on symbol =. We are unable to decide whether to use production R → L to reduce or shift S → L • = R to next state. This is known shift/reduce content. There is no solution and parser cannot proceed and SLR (1) fails.

Operator Precedence Parsing

1. Precedence Relations

Bottom-up parsers for a large class of context-free grammars can be easily developed using *operator grammars*.

Operator grammars have the property that no production right side is empty or has two adjacent nonterminals. This property enables the implementation of efficient *operator-precedence parsers*. These parser rely on the following three precedence relations:

Relation	Meaning
$a \lessdot b$	a yields precedence to b
$a \doteq b$	a has the same precedence as b
$a \gtrdot b$	a takes precedence over b

These operator precedence relations allow to delimit the handles in the right sentential forms: $<\cdot$ marks the left end, $=\cdot$ appears in the interior of the handle, and $\cdot>$ marks the right end.

Let assume that between the symbols a_i and a_{i+1} there is exactly one precedence relation. Suppose that \$ is the end of the string. Then for all terminals we can write: \$ $<\cdot$ b and b $\cdot>$ \$. If we remove all nonterminals and place the correct precedence relation: $<\cdot$, $=\cdot$, $\cdot>$ between the remaining terminals, there remain strings that can be analyzed by easily developed parser.

For example, the following operator precedence relations can be introduced for simple expressions:

	id	+	*	\$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
\$	$<\cdot$	$<\cdot$	$<\cdot$	

Example: The input string:

$$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$$

after inserting precedence relations becomes

$$\$ <\cdot \ \mathbf{id}_1 \ \cdot> + <\cdot \ \mathbf{id}_2 \ \cdot> * <\cdot \ \mathbf{id}_3 \ \cdot> \$$$

Having precedence relations allows to identify handles as follows:

- scan the string from left until seeing $\cdot>$
- scan backwards the string from right to left until seeing $<\cdot$
- everything between the two relations $<\cdot$ and $\cdot>$ forms the handle

Note that not the entire sentential form is scanned to find the handle.

2. Operator Precedence Parsing Algorithm

Initialize: Set *ip* to point to the first symbol of *w*$

Repeat: Let *X* be the top stack symbol, and *a* the symbol pointed to by *ip*
if $ is on the top of the stack and ip points to $ **then return**
else
Let *a* be the top terminal on the stack, and *b* the symbol pointed to by *ip*
if $a <\cdot b$ **or** $a =\cdot b$ **then**
push *b* onto the stack
advance *ip* to the next input symbol
else if $a \cdot> b$ **then**
repeat
pop the stack
until the top stack terminal is related by $<\cdot$
to the terminal most recently popped
else *error()*
end

3. Making Operator Precedence Relations

The operator precedence parsers usually do not store the precedence table with the relations, rather they are implemented in a special way.

Operator precedence parsers use precedence functions that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison.

Not every table of precedence relations has precedence functions but in practice for most grammars such functions can be designed.

Algorithm for Constructing Precedence Functions

1. Create functions f_a for each grammar terminal a and for the end of string symbol;

2. Partition the symbols in groups so that f_a and g_b are in the same group if $a =\cdot b$ (there can be symbols in the same group even if they are not connected by this relation);

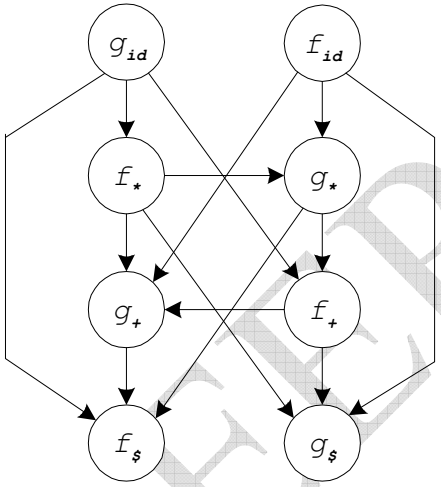
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b;

4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

Example: Consider the above table

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Using the algorithm leads to the following graph:



from which we extract the following precedence functions:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0