MODULE3

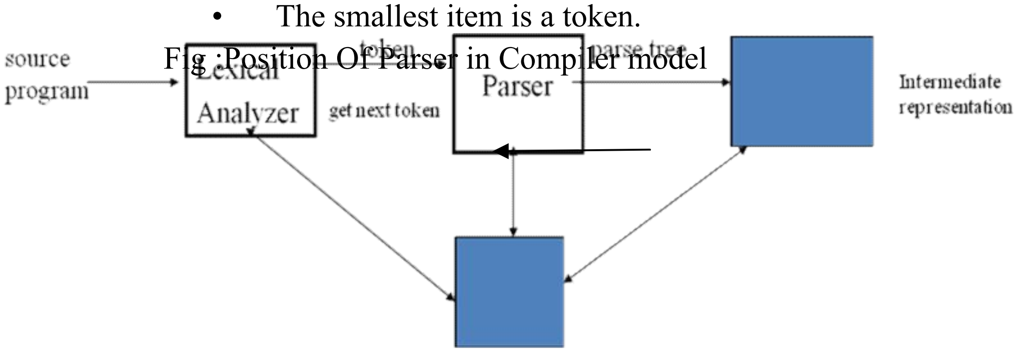
Introduction

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.

– a grammar can be directly converted into a parser by some tools.

- Parser works on a stream of tokens.
- The smallest item is a token.

Fig :Position Of Parser in Compiler model



SYMBOL TABLE

- We categorize the parsers into two groups:
  1. **Top-Down Parser**
  2. the parse tree is created top to bottom, starting from the root.

1. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-

classes of context-free grammars.

- LL for top-down parsing
- LR for bottom-up parsing

Syntax Error Handling

- Common Programming errors can occur at many different levels.

1. Lexical errors: include misspelling of identifiers, keywords, or operators.
2. Syntactic errors : include misplaced semicolons or extra or missing braces.
3. Semantic errors: include type mismatches between operators and operands.
4. Logical errors: can be anything from incorrect reasoning on the part of the programmer.

Goals of the Parser

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Error-Recovery Strategies

- Panic-Mode Recovery
- Phrase-Level Recovery
- Error Productions
- Global Correction

Panic-Mode Recovery

- On discovering an error, the parser discards input symbols one at a time until one of a designated set of Synchronizing tokens is found.
- Synchronizing tokens are usually delimiters.

Ex: semicolon or } whose role in the source program is clear and unambiguous.

- It often skips a considerable amount of input without checking it for additional errors.

Advantage:

Simplicity

Is guaranteed not to go into an infinite loop

Phrase-Level Recovery

- A parser may perform local correction on the remaining input. i.e

it may replace a prefix of the remaining input by some string that allows the parser to continue.

Ex: replace a comma by a semicolon, insert a missing semicolon

- Local correction is left to the compiler designer.
- It is used in several error-repairing compliers, as it can correct any input string.

- Difficulty in coping with the situations in which the actual error has occurred before the point of detection.

Error Productions

- We can augment the grammar for the language at hand with productions that generate the **erroneous constructs**.
- Then we can use the grammar augmented by these error productions to **Construct a parser.**
- If an error production is used by the parser, we can generate appropriate **error diagnostics** to indicate the erroneous construct that has been recognized in the input.

Global Correction

- We use algorithms that perform 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.
- It is too costly to implement in terms of time space, so these techniques only of theoretical interest.

Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  - A finite set of terminals (in our case, this will be the set of tokens)
  - A finite set of non-terminals (syntactic-variables)
  - A finite set of productions rules in the following form
    - $A \rightarrow \alpha$ where A is a non-terminal and

      α is a string of terminals and non-terminals (including the empty string)
  - A start symbol (one of the non-terminal symbol)

NOTATIONAL CONVENTIONS

1. **Symbols used for terminals are :**
   - Lower case letters early in the alphabet (such as a, b, c, . . .)
   - Operator symbols (such as +, *, . . .)
   - Punctuation symbols (such as parenthesis, comma and so on)
   - The digits(0…9)
   - Boldface strings and keywords (such as **id** or **if**) each of which represents

a single terminal symbol

2. Symbols used for non terminals are:

- ➢ Uppercase letters early in the alphabet (such as A, B, C, …)
- ➢ The letter S, which when it appears is usually the start symbol.
- ➢ Lowercase, italic names (such as *expr* or *stmt)*.

3. **Lower case greek letters, α, β, ψ for example represent (possibly empty) strings of grammar symbols.**

Example: using above notations list out terminals, non terminals and start symbol in the following example

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid id$$

Here terminal are +, -, *, / , (,), id

Non terminals are E, T, F
Start symbol is E

Derivations

$E \rightarrow E+E$

- E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$E \rightarrow E+E \rightarrow id+E \rightarrow id+id$

- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is

$\alpha A J \rightarrow \alpha\psi$ if there is a production rule $A \rightarrow \psi$ in our grammar

where α and J are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\rightarrow$: derives in one step

$\rightarrow$: derives in zero or more steps

$\rightarrow$: derives in one or more steps

CFG – Terminology

- L(G) is *the language of G* (the language generated by G) which is a set of sentences.
- *A sentence of L(G)* is a string of terminal symbols of G.
- If S is the start symbol of G then

ξ is a sentence of L(G) iff S $\rightarrow$ ξ where ξ is a string of terminals of G.

- If G is a context-free grammar, L(G) is a *context-free language*.

- Two grammars are *equivalent* if they produce the same language.
- S → α - If α contains non-terminals, it is called as a *sentential* form of G.

  - If α does not contain non-terminals, it is called as a *sentence* of G.

Derivation Example

E → -E → -(E) → -(E+E) → -(id+E) → -(id+id)

OR

E → -E → -(E) → -(E+E) → -(E+id) → -(id+id)

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \xrightarrow{lm} -E \xrightarrow{lm} -(E) \xrightarrow{lm} -(E+E) \xrightarrow{lm} -(id+E) \xrightarrow{lm} -(id+id)$$

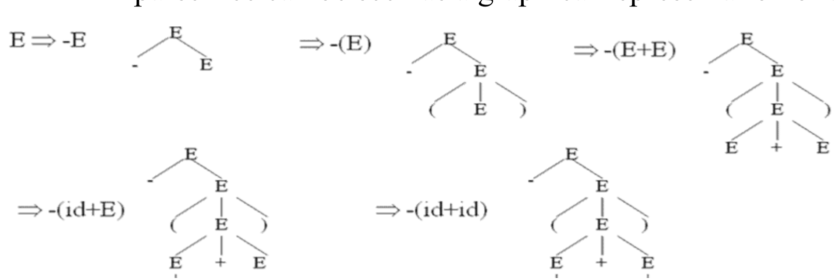
Right-Most Derivation

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

- We will see that the top-down parsers try to find the left-most derivation of the given source program.
- We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.



Problems on derivation of a string with parse tree:

1. Consider the grammar S→ (L) | a

   L→L,S | S

   i. What are the terminals, non terminal and the start symbol?
   ii. Find the parse tree for the following sentence
       a. (a,a)
       b. (a, (a, a))
       c. (a, ((a,a),(a,a)))
   iii. Construct LMD and RMD for each.

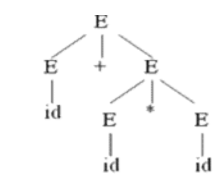
2. Do the above steps for the grammar S → aS | aSbS | χ for the string “aaabaab”

Ambiguity

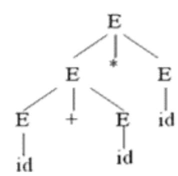
- A grammar produces more than one parse tree for a sentence

is called as an ***ambiguous*** grammar

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



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



- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar

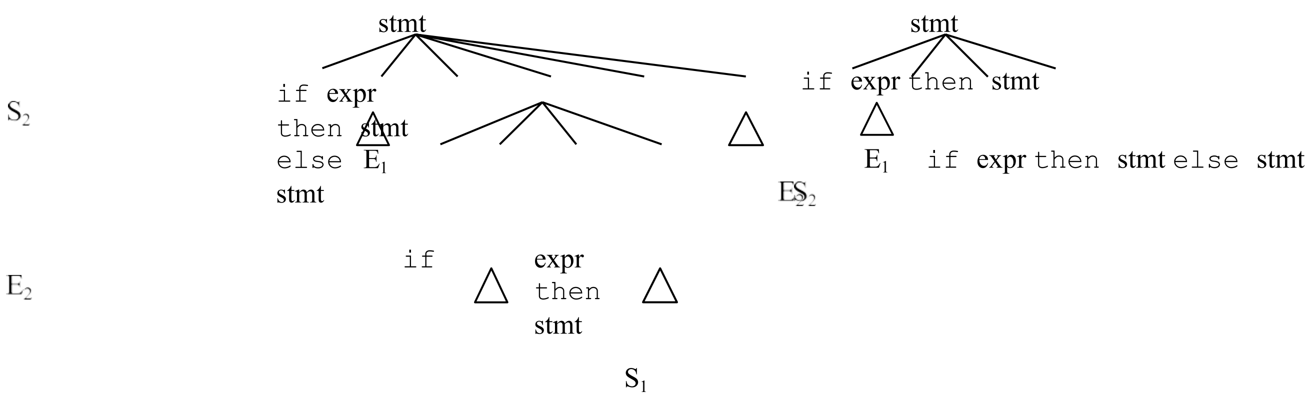
  → unique selection of the parse tree for a sentence
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An ambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

- EG:

Ambiguity (cont.)

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



1 **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

Problems on ambiguous grammar:

Show that the following grammars are ambiguous grammar by constructing either 2 lmd or 2 rmd for the given string.

1. S → S(S)S | χ with the string (( ) ( ))

2. S → S+S | |SS | (S) |S* a with the string

(a+a)*a S → aS | aSbS | χ with the string abab

.

Ambiguity – Operator Precedence

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

$E \to E+E \mid E*E \mid E\text{^}E \mid id \mid (E)$

disambiguate the grammar

precedence: ^ (right to left)
* (left to right)
\+ (left to right)

| T
E → | F
| G (E)
E+T

T →
T*F F

→ G^F

G → id

|

Left Recursion

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

$A \overset{+}{\to} A\alpha$ for some string α

- 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 left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \to A\ \alpha \mid \beta$ Y

$A \rightarrow J\ A'$

$A' \rightarrow \alpha\ A' \mid \sigma$

where J does not start with A

eliminate immediate left recursion

an equivalent grammar

In general,

$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_m \mid J_1 \mid \ldots \mid J_n$ where $J_1 \ldots J_n$ do not start with A

Y eliminate immediate left recursion

$A \rightarrow J_1\ A' \mid \ldots \mid J_n\ A'$

$A' \rightarrow \alpha_1\ A' \mid \ldots \mid \alpha_m\ A' \mid \sigma$

an equivalent grammar

Example

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

$\Downarrow$ eliminate immediate left recursion

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

Left-Recursion – Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \rightarrow Aa \rightarrow \underline{S}ca$ or $\underline{A} \rightarrow Sc \rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

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

$$A_i \rightarrow A_j \psi$$

by

$$A_i \rightarrow \alpha_1 \psi \mid \ldots \mid \alpha_k \psi$$

where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

  }
  - eliminate immediate left-recursions among A_i productions
}

Example2:

$$S \to Aa \mid b$$

$A \to Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

for S:

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

$A \to SdA' \mid fA'$

$A' \to cA' \mid \sigma$

- Replace $S \to Aa$ with

S →

SdA'a | fA'a

| fA'a | b

So, we will have S → SdA'a

- Eliminate the immediate left-recursion in S

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

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

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

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

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

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \sigma$

Problems of left recursion

1. S → S(S)S **|** χ

2. S → S+S | |SS | (S) |S* a

3. S → SS+ | SS* | a

4. bexpr → bexpr or bterm | bterm

bterm →bterm and bfactor | bfactor

bfactor → not bfactor | (bexpr) | true | false

5. S → (L) | a, L → L,S | S

Left-Factoring

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

  grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt → if expr then stmt else stmt |
if expr then stmt

- when we see if, we cannot now which production rule to choose to re-write *stmt* in the derivation.
- 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

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha J_1 \mid \ldots \mid \alpha J_n \mid \psi_1 \mid \ldots \mid \psi_m$$

convert it into

$$A \rightarrow \alpha A' \mid \psi_1 \mid \ldots \mid \psi_m$$

$$A' \rightarrow J_1 \mid \ldots \mid J_n$$

Left-Factoring – Example1

A → abB | aB | cdg | cdeB | cdfBY

A → aA' | cdg | cdeB | cdfB

A' → bB | B

Y

A → aA' | cdA''

A' → bB | B

A'' → g | eB | fB

Example2

A → ad | a | ab | abc | b

Y

A → aA’ | b

A’ → d |σ | b | bc

Y

A → aA’ | b

A’ → d |σ | bA’’ A’’ → σ | c

Problems on left factor

1. S → iEtS | iEtSeS | a,

2. S → S(S)S | χ

3. S → aS | aSbS |χ

4. S → SS+ | SS* | a

E → b

6. S→ 0S1 | 01

7. S → S+S | |SS | (S) |S* a

8. S → (L) | a, L → L,S | S

5. bexpr → bexpr or bterm | bterm

bterm →bterm and bfactor | bfactor rfactor

bfactor → not bfactor | (bexpr) | true | false

9. rexpr → rexpr + rterm | rterm

rterm →rterm rfactor |

rfactor → rfactor* | rprimary

rprimary →a |b

do both

leftfactor and left recursion

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.

- L1 = { ξcξ | ξ is in (a | b)*} is not context-free

➔ Declaring an identifier and checking whether it is declared or not later.

We cannot do this with a context-free language. We need not context-free). semantic analyzer (which is

- L2 = $\{a^n b^m c^n d^m \mid n\Sigma 1 \text{ and } m\Sigma 1\}$ is not context-free

Declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

Top – down parser

Recursive Descent parser

Predictive

parser LL(1) parser LL(k)

parser

First L stands for left to right scan Second L stands for LMD

(1) stands for only one i/p symbol to predict the parser

(2) stands for k no. of i/p symbol to predict the parser

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - Needs a special form of grammars (LL (1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.

Non-Recursive (Table Driven) Predictive Parser is also known as LL (1) parser.

Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.

- It tries to find the left-most

derivation. S → aBc

B → bc | b

S S

input: abc

a c a B B

c

b c fails, backtrack b

Predictive Parser

- When re-writing a non-terminal in a step, a predictive parser can derivatio

  uniquely choose a production rule by just looking the current symbol in the input string.

  $A \to \alpha_1 \mid \ldots \mid \alpha_n$ input: ... a ....

  current token

example

stmt → if ...... |

while ...... |
begin ...... |
for .....

- When we are trying to write the non-terminal *stmt*, if the current token is if we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

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.

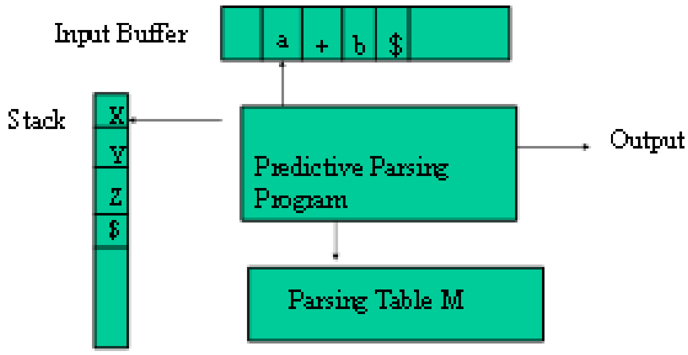


Fig: Model Of Non-Recursive predictive parsing

LL(1) 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-dimension

al

ar
ra
y
M
[A
,
a]
ea
ch
ro
w
is

a

no
n-
ter
mi
na
l
sy
m
bo
l
each column is a terminal symbol or the special symbol $

each entry holds a production rule.

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to σ, then σ is also in FIRST(α) .
- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.
  - a terminal a is in FOLLOW(A) if $S \rightarrow \alpha A a J$
  - \$ is in FOLLOW(A) if $S \rightarrow \alpha A$

Compute FIRST for Any String X

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \rightarrow \sigma$ is a production rule ➔ σ is in FIRST(X).
- If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule

  ➔ if a terminal **a** in FIRST(Y_i) and σ is in all FIRST(Y_j) for j=1,...,i-1then **a** is in FIRST(X).

  ➔ if σ is in all FIRST(Y_j) for j=1,...,n then σ is in FIRST(X).

- If X is σ ➔ FIRST(X)={σ}
- If X is $Y_1Y_2..Y_n$ ➔ if a terminal **a** in FIRST(Y_i) and σ is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).

  ➔ if σ is in all FIRST(Y_j) for j=1,...,n then σ is in FIRST(X).

Compute FOLLOW (for non-terminals)

- If S is the start symbol ➔ \$ is in FOLLOW(S)
- if $A \rightarrow \alpha B J$ is a production rule ➔ everything in FIRST(J)

is FOLLOW(B) except σ

- If (A → αB is a production rule) or (A → αBβ is a production rule and σ is in FIRST(β))

  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are $ ➔ parser halts (successful completion)
2. If X and a are the same terminal symbol (different from $)

➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If X is a non-terminal

➔ parser looks at the parsing table entry M[X, a]. If M[X, a] holds a production rule $X \rightarrow Y_1Y_2...Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1Y_2...Y_k$ to represent a step of the derivation.

1. none of the above ➔ error
   - all empty entries in the parsing table are errors.

If X is a terminal symbol different from a, this is also an error case.

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 → Y1Y2···Yk ) {
        output the production X → Y1Y2···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

LL(1) Parser – Example1

LL (1) Parsing Table

$S \rightarrow aBa$

$B \rightarrow bB \mid \sigma$

FIRST FUNCTION

FIRST(S) = {a} FIRST (aBa) = {a}

FIRST (B) = {b} FIRST (bB) = {b} FIRST (σ) = {σ}

FOLLOW
FUNCTION

FOLLOW(S) = {$}   FOLLOW (B) = {a}

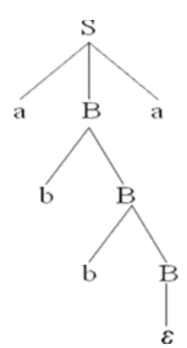
	a	**b**	**$**
S	S → aBa		
B	B → σ	B → bB	

stack **input** **output**

)

$S	abba$	S → aBa
$aBa	abba$	
$aB	bba$	B → bB
$aBb	bba$	
$aB	ba$	B → bB
$aBb	ba$	
$aB	a$	B → σ
$a	a$	
$	$	accept, successful completion

Outputs: S → aBa   B → bB   B → bB   B → σ



parse tree

Derivation(left-most): S→aBa→abBa→abbBa→abba

Example2

E → TE'

E' → +TE' | σ

T → FT'

T' → *FT' | σ

F → (E) | id

Soln:

FIRST Example

)

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \sigma$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \sigma$

$F \rightarrow (E) \mid id$

FIRST(F) = {(,id}

FIRST(T') = {*, σ}

FIRST(T) = {(,id}

FIRST(E') = {+, σ}

FIRST(E) = {(,id}

FIRST(σ) = {σ}

FIRST(TE') = {(,id}

FIRST(+TE') = {+}

FIRST(σ) = {σ}

FIRST(FT') = {(,id}

FIRST(*FT') = {*}

)

FIRST((E)) = {(}
FIRST(id) = {id}

FOLLOW Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \sigma$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \sigma$

$F \rightarrow (E) \mid id$ FOLLOW (E) = {$,)}

FOLLOW (E') = {$,)}

FOLLOW (T) = {+,), $}

FOLLOW (T') = {+,), $}

FOLLOW (F) = {+, *,), $}

Constructing LL (1) Parsing Table – Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
    - for each terminal a in FIRST(α) ➔ add $A \rightarrow \alpha$ to M[A, a]
    - If σ in FIRST(α) ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A, a]
    - If σ in FIRST(α) and $ in FOLLOW(A) ➔ add $A \rightarrow \alpha$ to M[A, $]

– All other undefined entries of the parsing table are error entries.

Constructing LL (1) Parsing Table – Example

$E \rightarrow TE'$ FIRST (TE') = {(, id} ➔ $E \rightarrow TE'$ into M [E, (] and M[E, id]

$E' \rightarrow +TE'$ FIRST (+TE') = {+} ➔ $E' \rightarrow +TE'$ into M [E', +]

$E' \rightarrow \sigma$ FIRST (σ) = {σ} ➔ none

)

but since σ in FIRST(σ) and

FOLLOW(E')={$,)}

➔ E' → σ into M[E',$] and M[E',)]

T → FT' T' → *FT'	FIRST (FT') = {(, id} FIRST (*FT') = {*}	➔ T → FT' into M[T,(] and M[T, id] ➔ T' → *FT' into M [T',*]
T' → σ	FIRST (σ) = {σ}	➔ none

but since σ in

FIRST(σ)

and FOLLOW(T')={$,) ,+}

➔ T' → σ into M [T', $], M [T' ,)] and M [T',+]

F → (E)	FIRST ((E)) = {(} FIRST (id) = {id}	➔ F → (E) into M [F, (]
F → id		➔ F → id into M [F, id]

)

	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → σ	E' → σ
T	T → FT'			T → FT'		
T'		T' → σ	T' → *FT'		T' → σ	T' → σ
F	F → id			F → (E)		

<u>**stack**</u>	<u>**input**</u>	<u>**output**</u>
$E	id+id$	E → TE'
$E'T	id+id$	T → FT'
$E' T'F	id+id$	F → id
$ E' T'id	id+id$	
$ E' T'	+id$	T' → σ
$ E'	+id$	E' → +TE'
$ E' T+	+id$	
$ E' T		
$ E' T'F		

i

d

$

i

d

$

)

bnh

)

Construct the predictive parser LL (1) for the following grammar and parse the given string

· S S(S)S **|** with the string (( ) ( )) · S + S S | |*SS | a with the string “+*aa a**”** · S aSbS | bSaS | with the string “aabbbab” · bexpr bexpr or bterm | bterm bterm bterm and bfactor | bfactor bfactor not bfactor | (bexpr) | true | false string “ not(true or false)” 5. S 0S1 | 01 string “00011” 6. S aB | aC | Sd |Se B bBc | f C g	7. P Ra | Qba . R aba | caba | Rbc Q bbc |bc cababca” strin g “ 8. S PQR . P → a | Rb | γ Q c | dP | R e | f string “ adeb” 9. E E+ T |T T id | id[ ] | id[X] X E, E | E string “id[id]” 10. S (A) | 0 A SB B ,SB | string “ (0, (0,0))” 11. S a | | (T) T T,S | S String (a,(a,a)) String ((a,a), , (a),a)

LL (1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL (1) grammar. one input symbol used as a look-head symbol do determine parser action LL (1) left most derivation input scanned from left to right
- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL (1) grammar.

A Grammar which is not LL (1)

S → i C t S E | a

E → e S | σ

C → b

FIRST(iCtSE) = {i} FIRST(a) = {a}
FIRST(eS) = {e}

FIRST(σ) = {σ}

FIRST(b) = {b}

FOLLOW(S) = {$, e}
FOLLOW (E) = {$, e}
FOLLOW(C) = {t }

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e\,S$ $E \rightarrow \sigma$			$E \rightarrow \sigma$
C		$C \rightarrow b$				

two production rules for M[E, e]

Problem ➔ ambiguity

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If it's (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL (1) grammar.
  - $A \rightarrow A\alpha \mid \beta$

    ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \rightarrow \beta\alpha$.

    ➔ If β is σ, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

    ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL (1) grammar.

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.

  - f the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A, a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recovered from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery

– Skipping the input symbols until a synchronizing token is found.

- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we would like a compiler to make as few changes as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL (1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.

- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.

- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery – Example

S → AbS | e | σ

A → a | cAd

Soln:
FIRST (S) = FIRST (A) = {a, c}
FIRST (A) = {a, c}
FOLLOW (S) = {$}
FOLLOW (A) = {b, d}

–

	a	**b**	**c**	**d**	**e**	**$**
S	**S → AbS**	***sync***	**S → AbS**	***sync***	**S → e**	**S → σ**
A	**A → a**	***sync***	**A → cAd**	***sync***	***sync***	***sync***

Eg: input string "aab"

<u>stack</u>	**<u>input</u>**	**<u>output</u>**
$S	aab$	S → AbS
$SbA	aab$	A → a
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	S → AbS
$SbA	ab$	A → a
$Sba	ab$	

$Sb	b$	
$S	$	$S \rightarrow \sigma$
$	$	accept

Eg: Another input string "ceadb"

stack **input** **output**

$SS ceadb$ S → AbS

$SbA

$SbdAc

ceadb$ A → cAd

ceadb$

$SbdA eadb$ Error:unexpected e (illegal A)

(Remove all input tokens until first b or d, pop A)

$Sbd d

$Sb b

$S $

b

$

$ S → σ

$ $ accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - Change, insert, or delete input symbols.
  - issue appropriate error messages
  - Pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

  S → ... → ξ (the right-most derivation of ξ)

  → (the bottom-up parser finds the right-most derivation in the reverse order)

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol

sequence is the right side of a production) will replaced by the non- terminal at the left side of that production.

- There are also two more actions: accept and error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

a string ➔

the starting symbol reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

		$ E' T'	$	T' →σ
$ E' T'id		$ E'	$	E' →σ
	id$	$	$	accept

Rightmost Derivation: $S \rightarrow \xi$

Shift-Reduce Parser finds:

$\xi \rightarrow \ldots \overset{rm}{\rightarrow} S$

Example

S → aABb input string: aaabb

aaAbb

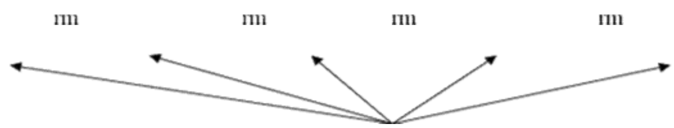
A → aA | a

B → bB | b

aAbb aABb S

Y reduction → aAbb → aaAbb → aaabb

S → aABb



Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is

    h a d

    n l

    e

- A **handle** of a right sentential form

ψ (÷ α⌡ ξ) is

a production rule A → ⌡ and a position of ψ

where the string ⌡ may be found and replaced by A to produce

the previous right-sentential form in S → αAξ → α⌡ ξ a rightmost derivation of ψ.

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ξ is a string of terminals.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$

input string

non

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

- Handle pruning help in finding handle which will be reduced to a terminal, that is the process of shift reduce parsing.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Right-Most Derivation of id+id*id

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential Form	Reducing Production
<u>id</u>+id*id	F → id
<u>F</u>+id*id	T → F
<u>T</u>+id*id	E → T
E+<u>id</u>*id	F → id
E+<u>F</u>*id	T → F
E+T*<u>id</u>	F → id
E+<u>T*F</u>	T → T*F
<u>E+T</u>	E → E+T
E	

Handles are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-parser action:
  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.
- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

Consider the following grams and parse the respective strings using shift-reduce parser.

(1) E → E+T | T T → T*F | F

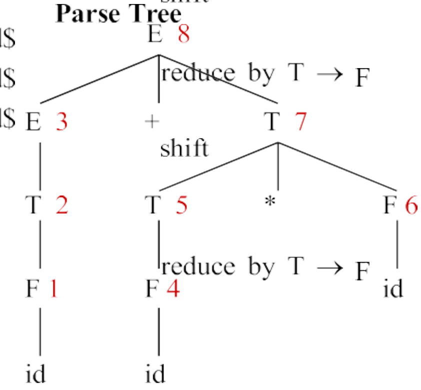
F → (E) | id string is “id + id * id” Here we follow 2 rules

1. If the incoming operator has more priority than in stack operator then perform shift.
2. If in stack operator has same or less priority than the priority of incoming operator then perform reduce.

Implementation of A A Stack Shift-Reduce Parser

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	
$F	+id*id$	reduce by T → F
$T	+id*id$	
$E	+id*id$	shift
$E+	id*id$	
$E+id	*id$	
$E+F	*id$	reduce by T → F
$E+T	*id$	
$E+T*	id$	
$E+T*id	$	
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T

Parse Tree



$E	$	accept

(2) S →TL;

T →int | float

L → L, id | id

String is “int id, id;” do shift-reduce parser.

(3) S →(L) |a L

→ L,S | S

String “(a,(a,a))” do shift-reduce parser.

Shift reduce parser problem

- Take the grammar:
- Sentence --> NounPhrase VerbPhrase VerbPhrase --> Verb | Adverb Verb Verb --> jumps | sings | ...

NounPhrase --> Art
Noun Art --> the | a
| ...
Noun --> dog | cat | ...

And the input: “the dog jumps”. Then the bottom up parsing is:

Stack	**Input Sequence** the dog jumps$ dog	**ACTION**
$	jumps$	SHI
$the	dog jumps$ jumps$ jumps$ jumps$	FT
$Art	$	word
$Art dog	$	onto
$Art Noun	$	stack
$NounPhrase	$	RED
$NounPhrase jumps		UCE
$NounPhrase Verb		usin
$NounPhrase VerbPhrase		g
		gram
$Sentence		mar
		rule

SHI
FT..
R
E
D
U
C
E
.
.
R
E
D
U
C
E
S
H
I
F
T
R
E
D
U
C
E
R
E
D
U
C
E
R
E
D
U
C
E
S
U
C
C

E
S
S

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

  left to right scanning | right-most derivation | k lookhead

- An ambiguous grammar can never be a LR grammar.