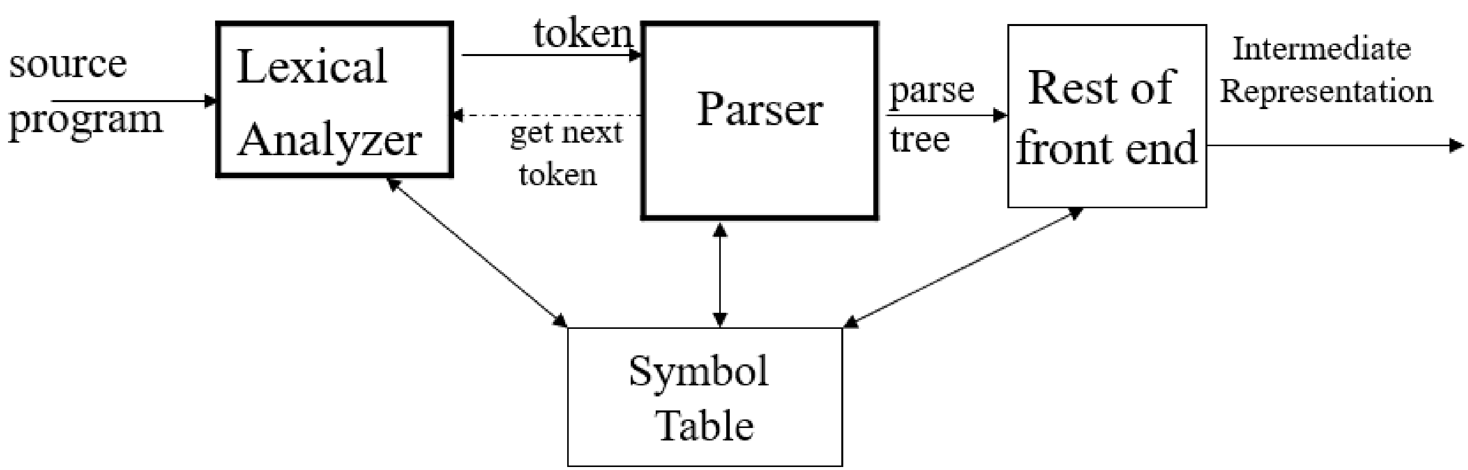
Module II
Syntax Analysis

Parsing

- **Syntax Analysis** or P**arsing** is the process of analyzing a text made of a sequence of tokens, to determine its grammatical structure with respect to a given formal grammar.
- A **syntax analyzer** or **parser** checks for correct syntax and builds a data structure (often parse tree).



Syntax Analyzer

- 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.
- The syntax of a programming language is described by a ***context-free grammar (CFG)*.**
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - a grammar can be directly converted into a parser by some tools.

Parsers

Parsers are categorized into two groups:

1. **Top Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom Up Parser**
   - the parse tree 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

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)
    - Terminals are basic symbols from which strings are formed.
  - A finite set of **non-terminals** (syntactic-variables)
    - Non terminals define sets of strings that help define the language generated by the grammar
    - Impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
  - 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)
    - Productions specify the manner in which terminals and non terminals can be combined to form strings.
  - A **start symbol** (one of the non-terminal symbol)
    - The set of strings denoted by the start symbol is the language defined the grammar.

Derivations

- **Example 1:**

  $E \rightarrow EAE \mid (E) \mid -E \mid id$

  $A \rightarrow + \mid - \mid * \mid /$

  E and A are non-terminals; E is the start symbol.
  All other symbols are terminals.

- **Example 2:**

  $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$

  $E \rightarrow (E)$

  $E \rightarrow id$

$E \Rightarrow E+E$

- 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.

Derivations

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 $\alpha A\beta \Rightarrow \alpha\gamma\beta$
  - if there is a production rule $A \rightarrow \gamma$ in our grammar;
  - where α and β 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)

$\overset{+}{\Rightarrow}$: derives in one or more steps

$\overset{*}{\Rightarrow}$: derives in zero 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

  w is a sentence of L(G) iff $S \stackrel{*}{\Rightarrow} w$ where w 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.
- $\mathbf{S \Rightarrow \alpha}$
  - 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

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

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

- 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**.

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

- Rightmost derivations are also called ***canonical derivations***

Left-Most and Right-Most Derivations

Left-Most Derivation

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

Right-Most Derivation

$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$

- $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, α is a left sentential form of the grammar

- Top-down parsers try to find the left-most derivation of the given source program.
- Bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

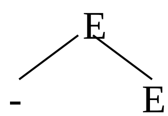
Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- **Yield or frontier** of the tree: leaves of the tree read from left to right
- A parse tree can be seen as a graphical representation of a derivation.

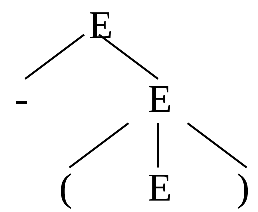
Parse Tree - Derivation

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

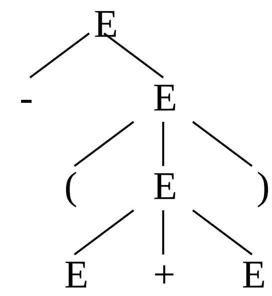
$E \Rightarrow -E$



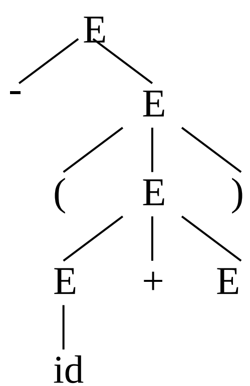
$\Rightarrow -(E)$



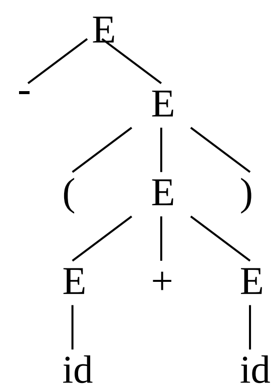
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$

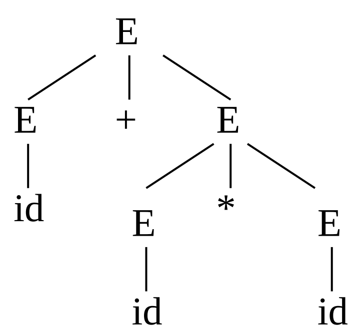


Ambiguity

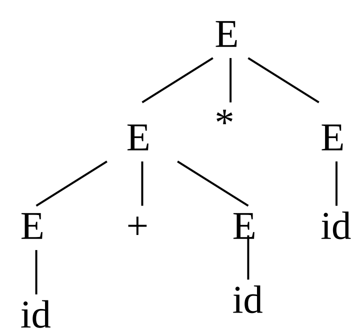
- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

- Produces **more than one leftmost or more than one right most derivation for the same sentence**

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



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



Ambiguity

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

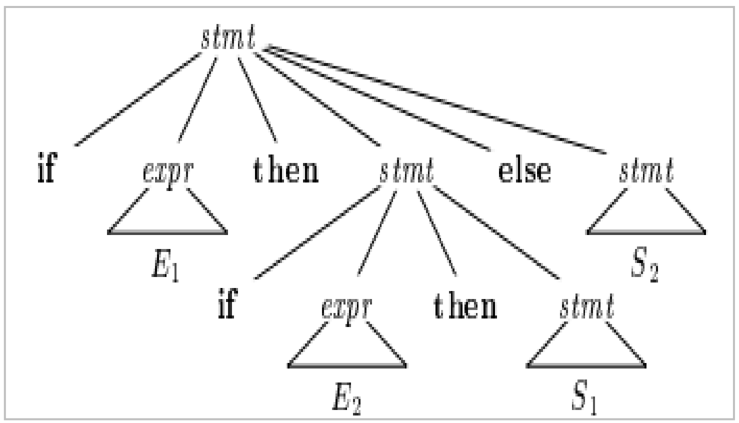
  ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
  - An unambiguous grammar should be written to eliminate the ambiguity.

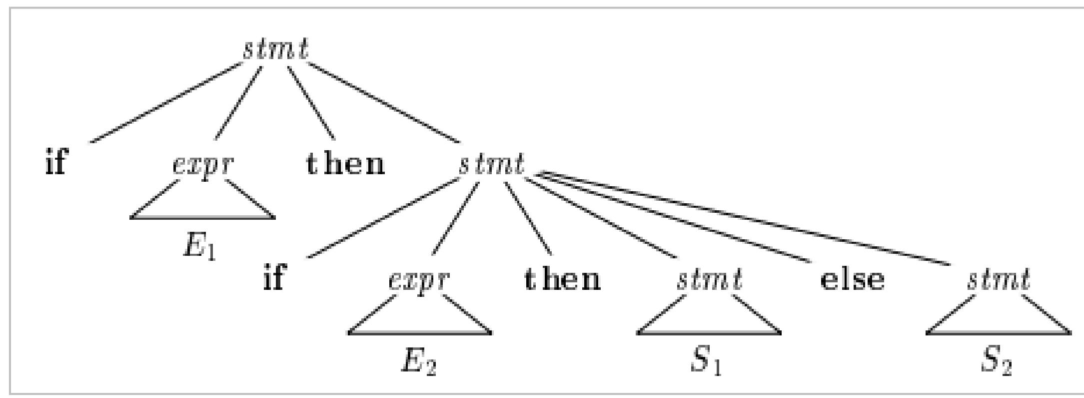
Ambiguity

stmt $\rightarrow$ if expr then stmt |
if expr then stmt else stmt | otherstmts

if E_1 then if E_2 then S_1 else S_2



(a)



(b)

Ambiguity

- 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 $\rightarrow$ matchedstmt | unmatchedstmt

matchedstmt $\rightarrow$ if expr then matchedstmt else matchedstmt
| otherstmts

unmatchedstmt $\rightarrow$ if expr then stmt |
if expr then matchedstmt else unmatchedstmt

Ambiguity

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

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

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

$\Downarrow$ disambiguate the grammar

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

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

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

Ambiguity

- Eliminate ambiguity from the above grammar. (Precedence order: id, ( ), ^ , * and /, + and -)

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

$E \rightarrow E*E \mid E/E$

$E \rightarrow E \wedge E$

$E \rightarrow (E) \mid id$

$\Downarrow$ disambiguate the grammar

E-> E+T | E-T |T

T-> T * P | T/P | P

P-> F ^ P | F

F-> (E) | id

Left Factoring

- Consider the productions,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ 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 to 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

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left Factoring-Algorithm

Algorithm 4.21: Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

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 $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

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

Left Factoring - Example1

$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$

$$\Downarrow$$

$$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$$

$$A' \rightarrow bB \mid B$$

$$\Downarrow$$

$$A \rightarrow aA' \mid cdA''$$

$$A' \rightarrow bB \mid B$$

$$A'' \rightarrow g \mid eB \mid fB$$

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 $\rightarrow$ `if` expr `then` stmt `else` stmt |
`if` expr `then` stmt

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

Left Recursion

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

  $A \overset{+}{\Rightarrow} A\alpha$ *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

$A \to A\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$$\left.\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{array}\right\}$$ *an equivalent grammar*

In general,

$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$$\left.\begin{array}{l} A \to \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' \to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon \end{array}\right\}$$ an equivalent grammar

Immediate Left Recursion - Example

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

$$A \rightarrow A\alpha \mid \beta$$
$$\Downarrow$$
$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

$$\Downarrow$$

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 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

      $A_i \rightarrow A_j \gamma$

        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

}

Eliminate Left Recursion - Example

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \to Sd$ with $A \to Aad \mid bd$
  So, we will have $A \to Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A

$$A \to bdA' \mid fA'$$
$$A' \to cA' \mid adA' \mid \varepsilon$$

- Equivalent grammar which is not left-recursive:

$$\boxed{\begin{aligned} &S \to Aa \mid b \\ &A \to bdA' \mid fA' \\ &A' \to cA' \mid adA' \mid \varepsilon \end{aligned}}$$

Top Down Parser

Top Down Parser

- Top-down parsing can be viewed
  - as an attempt to find a leftmost derivation for the input string
  - as the problem of constructing a parse tree for the input string, starting form the root and creating the nodes of the parse tree in preorder
- Recursive descent parsing is a general form of top down parsing which involves back tracking, i.e. making repeated scans of the input
- Predictive parser is special case of recursive descent parser where no backtracking is required

Recursive Descent Parser

- Consider the grammar:

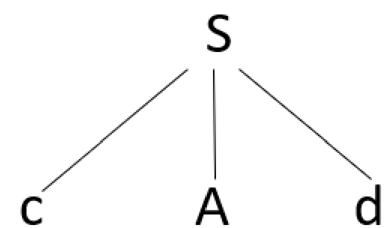
  $S \rightarrow cAd$

  $A \rightarrow ab \mid a$

  The input string, w=*cad*

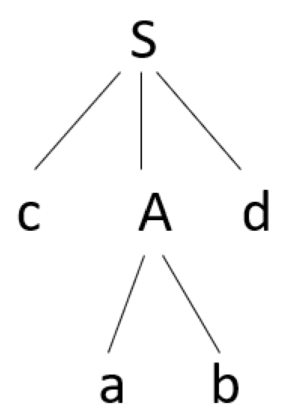
- Build parse tree:

  Step 1. From start symbol.



Recursive Descent Parser (Cont.)

Step 2. We expand A using the first alternative A→ab to obtain the following tree:

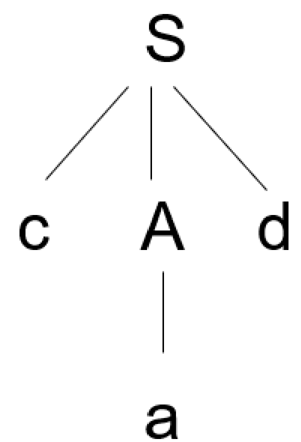


- Now, we have a match for the second input symbol “a”, so we advance the input pointer to “d”, the third input symbol, and compare d against the next leaf “b”.

Recursive Descent Parser (Cont.)

- Backtracking
  - Since “b” does not match “d”, we report failure and go back to A to see whether there is another alternative for A that has not been tried - that might produce a match!
  - In going back to A, we must reset the input pointer to “a”.

- Step 3



Recursive Descent Parser

Recursive descent parser for the Grammar

E → TE′

E′ → +TE′

T → FT′

T′ →∗ FT′ | ε

F → (E) | id

- The major approach of recursive-descent parsing is to relate each non-terminal with a procedure.
- There is a procedure for each non-terminal in the grammar.
- The objective of each procedure is to read a sequence of input characters that can be produced by the corresponding non-terminal, and return a pointer to the root of the parse tree for the non-terminal.

Recursive Descent Parser

Procedure E()
Begin
    T()
    E'()
End

$E \to TE'$

Procedure E'()
Begin
    If input symbol = '+' then
    Begin
        Advance()
        T()
        E'()
    End
End

$E' \to +TE'$

Recursive Descent Parser

```
Procedure T()
Begin
      F()
      T'()
End
```

T→FT'

```
Procedure T'()
Begin
        If input symbol = '*' then
        Begin
                Advance()
                F()
                T'()
        End
End
```

T'→*FT'

Recursive Descent Parser

```
Procedure F()
Begin
      If input symbol = 'id ' then
            Advance()
      Else if input symbol = '('  then
      Begin
            Advance()
            E()
            if input symbol = ')' then
                  Advance()
             else error()
      End
      Else Error()
```

$F \rightarrow id \mid (E)$

Predictive Parser

- By writing a grammar and eliminating left recursion and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive descent parser that needs no backtracking, i.e. a predictive parser
- Predictive parsers can be constructed for a class of grammars called LL(1).
  - L->Left to right scanning
  - L->Leftmost derivation
  - 1->One input symbol (lookahead symbol) at each step
- No left recursive or ambiguous grammar can be LL(1)

Introduction

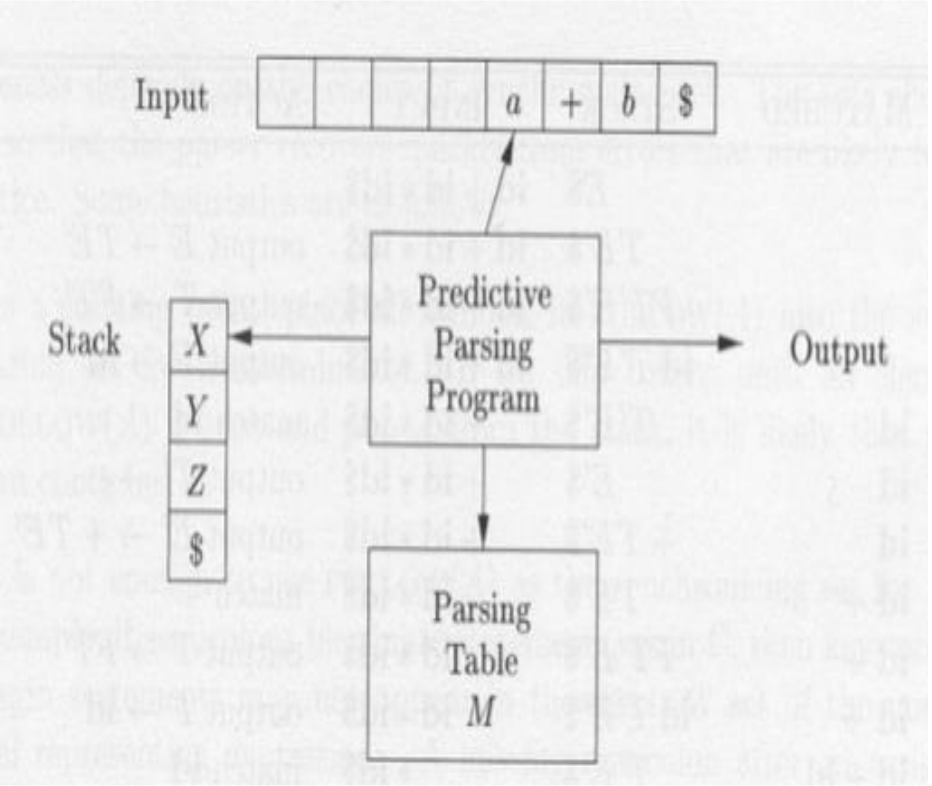
- To construct a predictive parser we must know
  - Input symbol a
  - Non terminal A to be expanded i.e. which one of the alternatives of production $A \rightarrow \alpha1 | \alpha2 | …. | \alpha n$ is the unique alternative that derives a string beginning with a
  - Proper alternative must be detectable by looking at only first symbol it derives.
- Flow of control constructs with their distinguishing keywords are usually detected.
- Keywords tells us which alternative is the one that could find the statement.
- Eg:- stmt → if expr then stmt else stmt

  |while expr do stmt

  |begin stmt_list end

Non recursive Predictive Parsing

- The predictive parser has :-
- **input** - contains string to be parsed followed by $
- **stack** - sequence of grammar symbols preceded with $. Initially the stack contains the start symbol S on top of $
- **parsing table** - two dimensional array M[A,a] where A is a nonterminal and *a* is the terminal or the symbol $
- **output**



Non recursive Predictive Parsing

- The parser is controlled by a program that behaves as follows:
- If X=a=\$, parser halts and announces successful completion of parsing.
- If X=a≠\$, parser pops X off the stack and advances the input pointer to next input symbol.
- If X is non terminal, program consults entry M[X,a]of the parsing table, M
  - If M[X,a]={X→UVW}, the parser replaces X on top of the stack with WVU (U on top)
  - If M[X,a]=error, the parser calls an error recovery routine

Algorithm for nonrecursive predictive parsing:

Input : A string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has $\$S$ on the stack with *S*, the start symbol of *G* on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

```
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 X is a terminal or $ then
                if X = a then
                        pop X from the stack and advance ip
                else error()
        else                            /* X is a non-terminal */
            if M[X, a] = X →Y1Y2 … Yk then begin
            pop X from the stack;
            push Yk, Yk-1, … ,Y1 onto the stack, with Y1 on top;
            output the production X → Y1 Y2 . . . Yk
        end
        else error()
until X = $
```

Predictive Parser

Input: id + id * id $

Grammar

E -> TE’
E’ -> +TE’|ε
T -> FT’
T’ -> *FT’|ε
F -> (E)|id

Stack:

E
$

Parsing Table

Non Terminal	Terminals					
	id	**+**	*****	**(**	**)**	**$**
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> id			F -> (E)		

MATCHED	STACK	INPUT	ACTION
	\$ E	id+id * id\$	
	\$ E’ T	id+id * id\$	E->TE’
	\$ E’ T’ F	id+id * id\$	T->FT’
	\$ E’ T’ id	id+id * id\$	F->id
id	\$ E’ T’	+id * id\$	Match id
id	\$ E’	+id * id\$	T’->Є
id	\$ E’ T +	+id * id\$	E’-> +TE’
id+	\$ E’ T	id * id\$	Match +
id+	\$ E’ T’ F	id * id\$	T-> FT’
id+	\$ E’ T’ id	id * id\$	F-> id
id+id	\$ E’ T’	* id\$	Match id
id+id	\$ E’ T’ F*	* id\$	T’-> *FT’
id+id*	\$ E’ T’ F	id\$	Match *
id+id*	\$ E’ T’ id	id\$	F-> id
id+id*id	\$ E’ T’	\$	Match id
id+id*id	\$ E’	\$	T’-> Є
id+id*id	\$	\$	E’-> Є

FIRST and FOLLOW

- To construct the parsing table, we need two functions associated with grammar G
  - FIRST
  - FOLLOW
- If α is any string of grammar symbols, FIRST(α) is any set of terminal symbols that begin the strings derived from α.
- If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α)
- FOLLOW(A) is the set of all terminals that can appear immediately to the right of A (i.e. *follow A*) in some sentential form,
  - i.e. the set of all terminals a in $S \overset{*}{\Rightarrow} \alpha A a \beta$ for some α and β.
- If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A)

FIRST (X)

- **Rules**

1. If X is a terminal, then FIRST(X) is {X}.

2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X)

3. If $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production, then place 'a' in FIRST(X), if for some i, a is in Y_i and ϵ is in all of FIRST(Y_1), …..,FIRST(Y_{i-1}), i.e. $Y_1Y_2 \ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$

   – If ϵ is in FIRST(Y_j) for all j=1,2,…,k, then add ϵ to FIRST(X)

   – Everything in FIRST (Y_1) is in FIRST(X).

   – IF Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) to FIRST(X) and so on

FIRST (X)

- We compute FIRST ($X_1X_2…X_n$) as follows:
  - Add to FIRST ($X_1X_2…X_n$) all non ϵ symbols of FIRST(X_1)
  - If ϵ is in FIRST(X1), add all non ϵ symbols of FIRST(X_2)
  - Also add non ϵ symbols of FIRST(X_3) if ϵ is in both FIRST(X_1) and FIRST(X_2)
  - Finally add ϵ in FIRST ($X_1X_2…X_n$) if for all i, FIRST(X_i) contains ϵ

Example: Rule 3

$X \rightarrow Y_1Y_2Y_3$

$Y1 \rightarrow a|\varepsilon$

$Y2 \rightarrow b|\varepsilon$

$Y3 \rightarrow c|\varepsilon$

Case 1:

$X \Rightarrow Y_1Y_2Y_3$

$\Rightarrow aY_2Y_3$

Case 2:

$X \Rightarrow Y_1Y_2\,Y_3$

$\Rightarrow Y_2Y_3$

$\Rightarrow bY_3$

Case 3:

$X \Rightarrow Y_1Y_2Y_3$

$\Rightarrow Y_2Y_3$

$\Rightarrow Y_3$

$\Rightarrow c$

Case 4:

$X \Rightarrow Y_1Y_2Y_3$

$\Rightarrow Y_2Y_3$

$\Rightarrow Y_3$

$\Rightarrow \varepsilon$

- FIRST(X)={a,b,c,ε}

FIRST()

- **Production Rules:**

E -> TE’
E’ -> +TE’ | Є
T -> FT’
T’ -> *FT’ | Є
F -> (E) | id

- FIRST(F)=FIRST((E)) U FIRST(id) ={(, id}

FIRST()

- **Production Rules:**

E -> TE’
E’ -> +TE’|Є
T -> FT’
T’ -> *FT’ | Є
F -> (E) | id

- FIRST(F)=FIRST((E)) U FIRST(id) ={(, id}
- FIRST(T)= FIRST(FT’) =FIRST(F) ={(, id}
- FIRST(E)= FIRST(TE’) =FIRST(T) ={(, id}

FIRST()

- **Production Rules:**

E -> TE’
E’ -> +TE’|Є
T -> FT’
T’ -> *FT’ | Є
F -> (E) | id

- FIRST(F)=FIRST(T) = FIRST(E)= {(, id}

- FIRST(E’)=FIRST(+TE’) U FIRST(Є) ={+, Є}

FIRST()

- FIRST(F)=FIRST(T) = FIRST(E)= {(, id}
- FIRST(E’)=={+, Є}

E -> TE’
E’ -> +TE’|Є
T -> FT’
T’ -> *FT’ | Є
F -> (E) | id

- FIRST(T’)=FIRST(*FT’) U FIRST(Є) ={*, Є}

FIRST(E)= FIRST(T)= FIRST(F)={ (, id}
FIRST(E’)={+, Є}
FIRST(T’)={*, Є}

FOLLOW(A)

1. Place \$ in FOLLOW(S) where S is the start symbol and \$ is the right end marker

2. If $A \rightarrow \alpha B\beta$ is a production, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B\beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example

- **<u>Rule 2</u>**
- $S \rightarrow aAB$
- $A \rightarrow c$
- $B \rightarrow b|d$ FIRST(B)= {b,d}

- $S \Rightarrow aAB \Rightarrow aAb$
- $S \Rightarrow aAB \Rightarrow aAd$

Example

- **Rule 3**
- $S \rightarrow aAd \mid Aa$ FOLLOW(A)= {d,a}
- $A \rightarrow cB$ *[of the form $A \rightarrow \alpha B$]*
- $B \rightarrow b$

- $S \Rightarrow aAd \Rightarrow acBd$
- $S \Rightarrow Aa \Rightarrow cBa$

FOLLOW(E)

1. Put $ in FOLLOW(E) by rule 1

2. Production F → (E)

- Apply rule 2
- A=F, α=(, B=E, β=)
  - Add ')' to FOLLOW(E)
- FOLLOW(E)={$,)}

1. Place $ in FOLLOW(S) where S is the start symbol and $ is the right end marker

2. If A -> αBβ is a production, then everything in FIRST(β) except Є is in FOLLOW(B).

FOLLOW(E')

<u>Production $E \rightarrow TE'$</u>

If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

- Apply rule 3

- $E \rightarrow TE'$ of the form $A \rightarrow \alpha B$, where $A=E$, $\alpha=T$ and $B=E'$,
- Everything in FOLLOW(E) is in FOLLOW(E')

- FOLLOW(E')=FOLLOW(E)={\$,)}

FOLLOW(E')

<u>Production E' $\rightarrow$ +TE'</u>

- Apply rule 3
- E' $\rightarrow$ +TE' of the form A$\rightarrow$αB, where A=E', α=+T and B=E',
- Everything in FOLLOW(E') is in FOLLOW(E')

- FOLLOW(E')= {$,)}

FOLLOW(T)

If $A \rightarrow \alpha B\beta$ is a production, then everything in FIRST(β) except Є is in FOLLOW(B).

<u>Production $E \rightarrow TE'$</u>

- **Apply rule 2**
- $E \rightarrow TE'$ of the form $A\text{->}\alpha B\beta$, where $A=E$, $\alpha=\epsilon$ and $B=T$ and $\beta=E'$,
- Everything in FIRST(E') except ϵ is in FOLLOW(T)

- ***Add {+} to FOLLOW(T)***

FIRST(E')={+, Є}

FOLLOW(T)

If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B\beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Production $E \rightarrow TE'$

- **Apply rule 3**
- $E \rightarrow TE'$ of the form $A \rightarrow \alpha B\beta$, where $A=E$, $\alpha=\epsilon$ and $B=T$ and $\beta=E'$,
- FIRST(E') contains ϵ, hence everything in FOLLOW(E) is in FOLLOW(T)
- ***Add {$,)} to FOLLOW(T)***

FOLLOW(T)

Production E’ $\rightarrow$ +TE’

- **Apply rule 2**
- Everything in FIRST(E’)- ϵ is in FOLLOW(T)
- **Apply rule 3**
- Everything in FOLLOW(E’) is in FOLLOW(T)

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

2. If A$\rightarrow$ αBβ is a production, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production A$\rightarrow$αB or a production A $\rightarrow$αBβ, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

FOLLOW(T’)

Production T $\rightarrow$ FT’

- Apply rule 3
- T $\rightarrow$ FT’ of the form A$\rightarrow\alpha$B, where A=T, α=F and B=T’, everything in FOLLOW(T) is in FOLLOW(T’)

- FOLLOW(T’)=FOLLOW(T) ={$,), +}

FOLLOW(T')

Production T' $\rightarrow$*FT'

- Apply rule 3
- T' $\rightarrow$ *FT' of the form A$\rightarrow$αB, where A=T, α=*F and B=T',
- Everything in FOLLOW(T') is in FOLLOW(T')

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

FOLLOW(F)

<u>Production T $\rightarrow$ FT’</u>

- **Apply rule 2**
- T $\rightarrow$ FT’ of the form A$\rightarrow$αBβ, where A=T, α=ϵ and B=F and β=T’
- Everything in FIRST(T’) except ϵ is in FOLLOW(F)

- ***Add {*} to FOLLOW(F)***

FIRST(T’)={*, Є}

FOLLOW(F)

Production $T \rightarrow FT'$

- **Apply rule 3**
- $T \rightarrow FT'$ of the form A->αBβ, where A=T, α=ϵ and B=F and β=T',
- FIRST(T') contains ϵ, hence everything in FOLLOW(T) is in FOLLOW(F)
- ***Add {$,), +} to FOLLOW(F)***

FOLLOW(F)

<u>Production T’ $\rightarrow$ *FT’</u>

- **Apply rule 2**
- Everything in FIRST(T’)- Є is in FOLLOW(F)
- **Apply rule 3**
- Everything in FOLLOW(T’) is in FOLLOW(F)
- FOLLOW(F)= {$,), +, *}

FIRST and FOLLOW

FIRST(E)= FIRST(T)= FIRST(F)={ (, id}

FIRST(E’)={+, Є}

FIRST(T’)={*, Є}

FOLLOW(E)=FOLLOW(E’)= {$,)}

FOLLOW(T)=FOLLOW(T’)= {$,), + }

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

Construction of Predictive Parsing Table

- INPUT: Grammar G.
- OUTPUT: Parsing table M.
- METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:
- 1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A,a]
- 2. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $] as well.
- 3. Make each undefined entry of M to be error

Construction of Predictive Parsing Table

E -> TE'
E' -> +TE'|Є
T -> FT'
T' -> *FT' | Є
F -> (E) | id

1. E → TE'

FIRST(TE')=FIRST(T)={(,id}

Add the production E→TE' to M[E,(] and M[E,id]

2. E' → +TE'

FIRST(+TE')={+}

Add the production E' → +TE' to M[E',+]

3. E' → Є

Add E' → Є to M[E',)] and M[E',$] since FOLLOW(E')={),$}

Construction of Predictive Parsing Table

4. T → FT’

FIRST(FT’)=FIRST(F)={(,id}

Add the production T → **FT’** to M[T,(] and M[T,id]

5. T’ → *FT’

FIRST(***FT’**)={*}

Add the production **T’** → ***FT’** to M[T’,*]

6. T’ →Є

Add T’ → Є to M[T’,)], M[T’,+] and M[T’,$] since FOLLOW(T’)={),$, +}

Construction of Predictive Parsing Table

7. F $\rightarrow$ (E)

Add F $\rightarrow$ (E) to M[F, (]

8. F $\rightarrow$ id

Add F $\rightarrow$ id to M[F, id]

Predictive Parsing Table

Non Terminal	Terminals					
	id	**+**	*****	**(**	**)**	**$**
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> id			F -> (E)		

Predictive Parser

- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.

- For some grammars, however, M may have some entries that are multiply defined.

- For example, if G is left-recursive or ambiguous, then M will have at least one multiply defined entry.

Predictive Parser

- $S \rightarrow iCtSS'|a$
- $S' \rightarrow eS\ |\varepsilon$
- $C \rightarrow b$
- FIRST(S)={i,a}
- FIRST(S')={e, ε}
- FIRST(C)={b}
- FOLLOW(S)={e,$}
- FOLLOW(S')={e,$}
- FOLLOW(C)={t}

Predictive Parser

S → iCtSS' | a
S' → eS | ε
C → b

- FIRST(S)={i,a}
- FIRST(S')={e, ε}
- FIRST(C)={b}

- FOLLOW(S)={e,$}
- FOLLOW(S')={e,$}
- FOLLOW(C)={t}

	a	**b**	**e**	**i**	**t**	**$**
S	S → a			S → iCtSS'		
S'			S' → eS S' → ε			S' → ε
C		C → b				

LL(1) Grammar

- A grammar whose parsing table has no multiply defined entries is said to be LL(1).
- No left recursive or ambiguous grammar can be LL(1)
- A grammar G is LL(1) if whenever $A \rightarrow \alpha|\beta$ are two distinct productions of G, the following conditions hold
  - For no *a*, do α and β derive strings beginning with *a*
  - At most one of α and β can derive the empty string
  - If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)