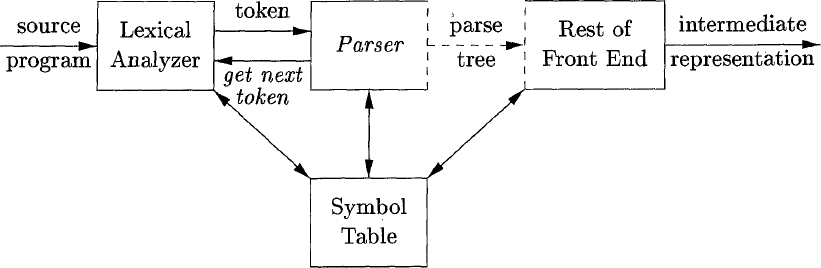
**Role of Parser:**

UNIT- II

Syntax Analysis

* Parser gets a string of tokens from lexical analyzer then construct parse tree and passes it to rest of the compiler for further processing.
* Checking and translation actions can be a part of parsing. So parse tree need not constructed explicitly.
* Parser can report any syntax error. It also recovers from commonly occurring errors to continue parsing.



* There are three types of parsers for grammar
  1. Universal
  2. Top down
  3. Bottom up
* Universal type of parser is inefficient to use in production compilers.
* Parsers commonly used in compiler are either top down or bottom up

1. Top down parser built parse tree from top (root) to bottom (leaves).
2. Bottom up method start from leaves work up to root

* In either case, input to parser is scanned from left to right, one symbol at a time.
* LL and LR grammar are enough to describe most syntactic structures in programming language.
* Parsers for class of LL grammar are constructed by hand and parsers for larger class of LR grammar are constructed by automated tools.

# Syntax error handling:

* If a compiler process only correct programs, its design and implementation would be simplified greatly.
* Compiler is expected some assistance from programmer in locating and tracking down errors and error handling is left to compiler designer.
* Parsing methods like LL and LR methods detect syntactic error efficiently.
* Common Programming errors occur at different levels of compiler are
  1. Lexical errors include misspellings as identifier, keyword and operators.
  2. Syntactic errors include misplaced semicolons and extra or missing braces.
  3. Semantic errors include type mismatches between operators and operands.
  4. Logical errors can be anything from incorrect reasoning
* Accurate detection of semantic and logical errors at compile time is a difficult task.
* Error handling in parser has some goals those are

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly to detect subsequent errors.
3. Add minimum overhead to processing of correct programs.

* Error handler report the line in which error is detected, because of this there is a good chance to detect actual error occurred within previous few tokens.

# Error Recovery Strategies:

* Simplest approach for parser is to quit from processing with informative error message when it detects first error.
* If number of errors is larger, it is better for compiler to quit after exceeding error limit.
* Some error recovery strategies are :

1. Panic mode
2. Phrase level
3. Error production
4. Global corrections

## Panic mode:

* + In this method, on discussing error, parser discards input symbols one at a time until designated set of tokens is found.
  + Panic mode correction skips considerable amount of input without checking it for

additional errors.

* + It is very simple but it is guaranteed not to go into infinite loop.

## Phrase level recovery :

* + On discovering error, Parser perform local corrections on remaining input like replace prefix of remain input with a string.
  + Local correction is to replace common by semicolon, delete semicolon or inserting missing semicolon.
  + We must be careful to choose replacements that do not lead to infinite loops.
  + Major drawback is very difficult to identify situation in which actual error has occurred before point of detection.

## Error Production :

* + By expecting common errors that might encounter, we construct grammar for language at hand with production that generates error point.
  + These error productions detect errors when parser using production. It also provides appropriate error diagnostics for errors those recognized in input.

## Global Correction:

* + Global Correction contains algorithms; those are used for choosing minimal subsequent changes to obtain globally least cost correction.
  + These provides small number of changes to convert incorrect string x to correct string y.
  + These modules are too costly to implement in terms of time and space. So these techniques are currently only theoretical.

# Context Free Grammar:

* Many programming language constructs have inherently recursive structure that can be defined by context free grammar.
* CFG consists of terminals, non terminals, stat symbol and productions.

1. Terminals are basic symbols from which strings are formed. When we are talking about grammars of Programming language if , then and else keywords are terminals.
2. Non terminals are syntactic variables that denote set of strings. These non terminals are helpful in define language generated by grammar. Statement and expression are non terminals.
3. In grammar one non terminal is indicated as start symbol and set of strings it denotes is language generated by grammar.
4. Productions of grammar specify the manner in which terminals and non terminals can be combined to form string. Each production of
   * Non terminal called head or left side of production.
   * Symbol → or ::=
   * Body or right side of production consists of zero or more terminals and non terminals.

expr → expr op expr expr → (expr)

expr → - expr expr → id

op → + op → - op → \*

op → / op → ↑

Start symbol: expr Terminals: id, +, -, \*, /,↑ Non terminals: expr, op

# National conventions:

1. Normally lower case letters, operators, digits, punctuation symbols (parenthesis, comma etc), boldface strings, if and id are terminals.
2. Normally uppercase letters, lowercase italic names such as expr or stmt are non terminals. letter s is starting symbol.
3. Uppercase letters x, y, z represent grammar symbol i.e either terminal or non terminals.
4. Lowercase letters u, v, w, z represent (empty) strings of terminals.
5. Lowercase Greek letters 𝘢, β, γ represent (empty) strings of grammar symbols.
6. If A → 𝘢1, A → 𝘢2, ---- A → 𝘢k are products as A on left then we write A →𝘢1|𝘢2| 𝘢k.
7. Unless stated otherwise, left side of the first production is start symbol. Example:

expression → expression + term expression → expression – term expression → term

term → term \* factor term → term / factor term → factor

factor → (expression) factor → id

Using the above conventions given grammar is rewritten as E → E + T | E - T | T

T → T \* F | T / F | F F → (E) | id

# Derivations:

* Construction of parse tree can be made exactly by taking a derivational view, in which productions are treated as rewriting rules.
* In derivation, we start with starting symbol; each rewriting step replaces a non-terminal by body of one of its productions.
* This derivational view corresponds to top down construction of parse tree, but the correctness afforded by derivations will helpful when bottom up parsing is discussed.
* At each step in derivation, there are two choices to be made. We need to choose which non terminal to replace .Based on this derivations are two types
  1. leftmost derivation
  2. rightmost derivation
* In leftmost derivation, the left most non terminal in each sentential is always chosen .If

𝘢 ⇒ β is step in which leftmost non terminal in 𝘢 is replaced, we write as 𝘢 ⇒ β.

𝑙𝑚

* In rightmost derivation the right most non terminal is always chosen, we write as 𝘢 ⇒ β.

𝑙𝑚

Example: construct leftmost and rightmost derivations for given grammar for string id + id.

E → E + E | E \* E | (E) | id Leftmost derivation is

E ⇒ E + E ⇒ id + E ⇒ id + id

𝑙𝑚

Rightmost derivation is

𝑙𝑚

𝑙𝑚

E ⇒⇒ E + E ⇒⇒ E + id ⇒⇒ id + id

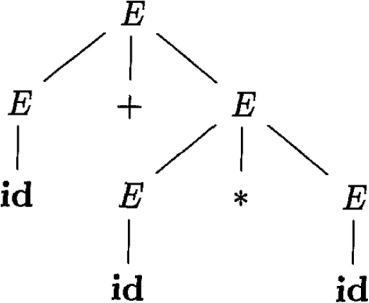
𝑟𝑚

𝑟𝑚

𝑟𝑚

# Parse Tree:

* Parse tree is graphical representation of derivation that filters out the order which productions are applied to replace non terminals.
* Interior node is labelled with non terminal in the head of production.
* Leaves of parse tree are labelled by non terminal or terminals.
  + Parse tree of the string id + id \* id for given grammar E → E + E | E \* E | (E) | id is



* There is a many two one relationship between parse trees and derivations.

# Ambiguity:

* A grammar that produces more than one parse tree for some input string Is said to be ambiguous.
* Ambiguous grammar is one that produces more than one left most derivation or more than one right most derivation for some input string.
* Below grammar permits two distinct left most derivations for input string “id + id \*id “. E → E + E | E \* E | (E) | id

E ⇒ E + E E

𝑙𝑚

⇒ id + E

𝑙𝑚

⇒ id + E \* E

𝑙𝑚

⇒ id + id \* E

𝑙𝑚

⇒ id + id \* id

𝑙𝑚

⇒⇒ E \* E

𝑟𝑚

⇒⇒ E + E \* E

𝑟𝑚

⇒⇒ id + E \* E

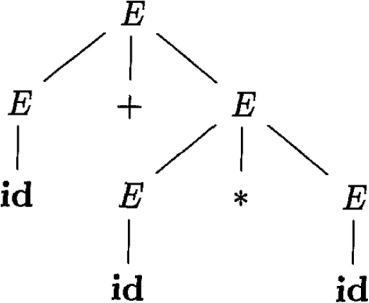
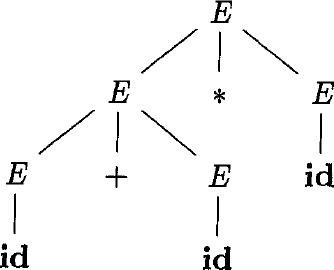
𝑟𝑚

⇒⇒ id + id \* E

𝑟𝑚

⇒⇒ id + id \* id

𝑟𝑚

# Context Free Grammar Vs Regular Expression:

* Context free grammars are strictly more powerful than regular expressions.
* Any language that can be generated using regular expressions can be generated by context free grammar but not vice versa.
* Every regular language is context free language but not vice versa.
* Regular expression (a|b)\* abb and grammar. A0 → aA0|bA0|aA1

A1 → bA2 A2 → bA3 A3 → ~~ε~~

Describe the same language, the set of strings as a ’s and b ’s ending in abb.

* The usage of regular expression is in context of lexical analysis phase, where as context free grammar is in context of syntax analysis phase.
* Regular expressions are very easy to understand when compare to context free grammar.

# Lexical Vs Syntax Analysis:

* Separating syntactic structure of language into lexical and non lexical parts provides compiler front end into two manageable sized components.
* Lexical rules of language are frequently quite simple; we do not need notation as grammar to describe them.
* Regular expressions are easier to understand notation for tokens than grammars.
* More efficient lexical analyzers can be constructed automatically from regular expressions than from grammars.
* Regular expressions are useful for describing construct like identifier, constants, keywords and whitespaces. Grammars, on other hand useful for describing if-then-else, balanced parenthesis, matching begin-ends.

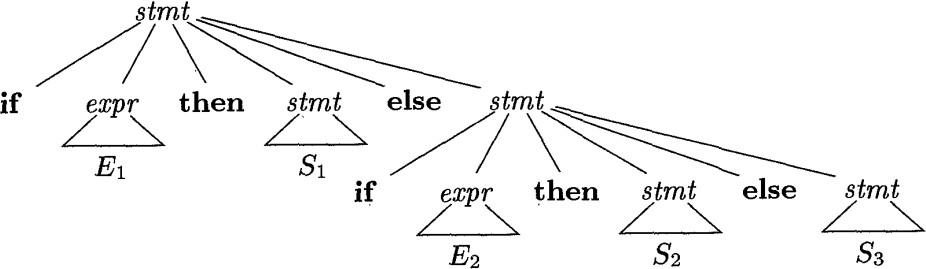
# Eliminating Ambiguity:

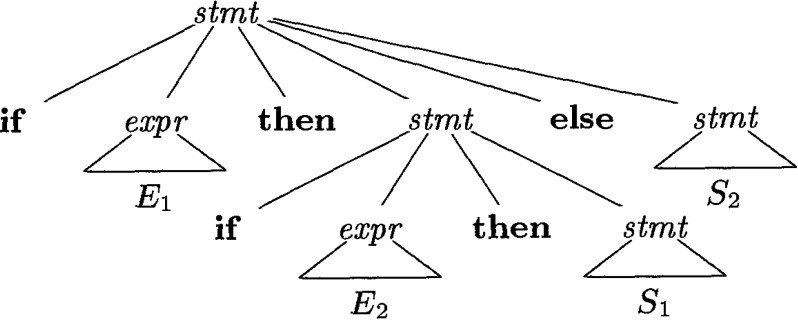
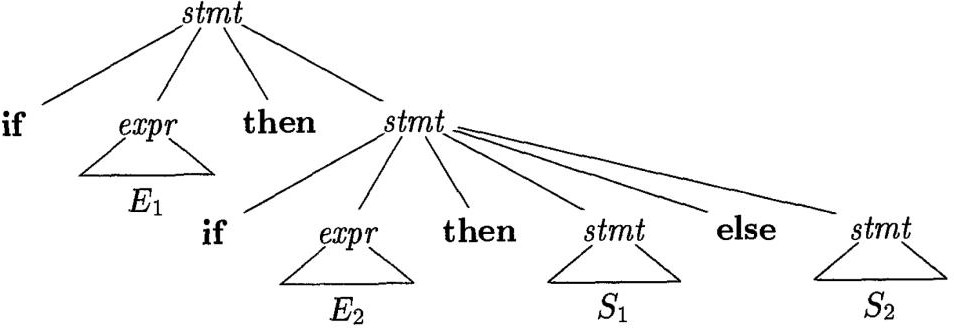
* Sometimes an ambiguous grammar can be rewritten to eliminate ambiguity.
* Eliminate the ambiguity from following dangling else grammar: Stmt if expression then statement

| if expr then stmt else stmt

| other

* According to this grammar, the compound conditional statement , If E1 then S1 else if E2 then S2 else S3.



* Grammar is ambiguous since the string, If E1 then if E2 then S1 else S2.
* To eliminate the ambiguity for above grammar, we can reconstruct the above grammar as shown below.

stmt → matched\_stmt

|open\_stmt

matched\_stmt → if expr then matched\_stmt else matched\_stmt

| other open\_stmt → if expr then stmt

| If expr then matched\_stmt else open\_stmt

# Elimination of Left Recursion:

+

* Grammar is left recursive if it has non terminal A such that there is a derivation A ⇒ Aᾳ for some string ᾳ
* Top down parsing methods can’t handle left recursive grammars .So, eliminate left recursion.
* To eliminate left recursion ,each left recursive pair of productions A → Aᾳ | β could be replaced by non left recursive productions:

A → βAI

A → ᾳAI | ε

Example: Eliminate the left recursion for the below grammar E → E + T | T

T → T \* F | F F → (E) | id

Left recursion elimination process

|  |  |  |
| --- | --- | --- |
| E → E + T | T E → T EI  EI → T EI | ε | T → T \* F | F T → F TI  TI → \* F TI | ε | F → (E) | id  No left recursion |

After elimination grammar is

E → T EI

EI → + T EI | ε T → F TI

TI → \* F TI | ε F → (E) | id

## Algorithm for Left Recursion:

Input: Grammar G with no cycles or ε-production. Output: Equivalent grammar with no left recursion.

Method: Resulting non left recursive grammar may have €-productions.

1. Arrange the non terminals in some order A1, A2, ……, An.
2. For(each i from 1 to n){
3. For(each j from 1 to i-1){
4. Replace each production of form Ai → Aj γ by productions Ai → δ1 γ| δ2 γ | | δk γ

Where Aj → δ1 | δ2 |..... | δk are all current Aj productions 5. }

1. Eliminate left recursion among Ai productions. 7. }

# Left Factoring:

* When the choice between two alternative A-productions is not clear by reading initial elements in input .This situation is called non-deterministic.
* Top down parsing methods can’t be non-deterministic situation .So, eliminate non- deterministic by left factoring.
* To implement left factoring, each non-deterministic productions A→ᾳß1/ᾳß2 can be replaced by

A → ᾳ AI

AI → β1 | β2

Example: Eliminate non-deterministic (left factoring) on below grammar S → i E + S | i E + S e S | a

E → b

Left recursion elimination process

|  |  |
| --- | --- |
| S → i E + S | i E + S e S | a S → i E + S SI | a  S → e S | ε | E → b  No non deterministic |

After elimination grammar is

S → i E + S SI | a S → e S | ε

E → b

## Left Factoring Grammar:

Input: Grammar G

Output: Equivalent left factored grammar

Method: For each non terminal A, find longest prefix ᾳ common to two or more its alternatives. Replace all A-productions A → ᾳ β1 | ᾳ β2 |….. | ᾳ βn | γ Where r represents all alternative not begin with ᾳ by

A → ᾳ AI | γ

A → β1 | β2 |….. | βn

# Non Context Free Language Constructs:

* Few syntactic constructs found in typical programming languages cant by specified using grammars alone. Two of these constructs are shown below
  1. Language consists of form WCW, where W represents declaration of identifier W, C represents program fragment, and second W represents the user of identifier.
  2. Problem by checking number of formal parameters in declaration of function agrees with number of actual parameters in the use of function. Language consists of string of

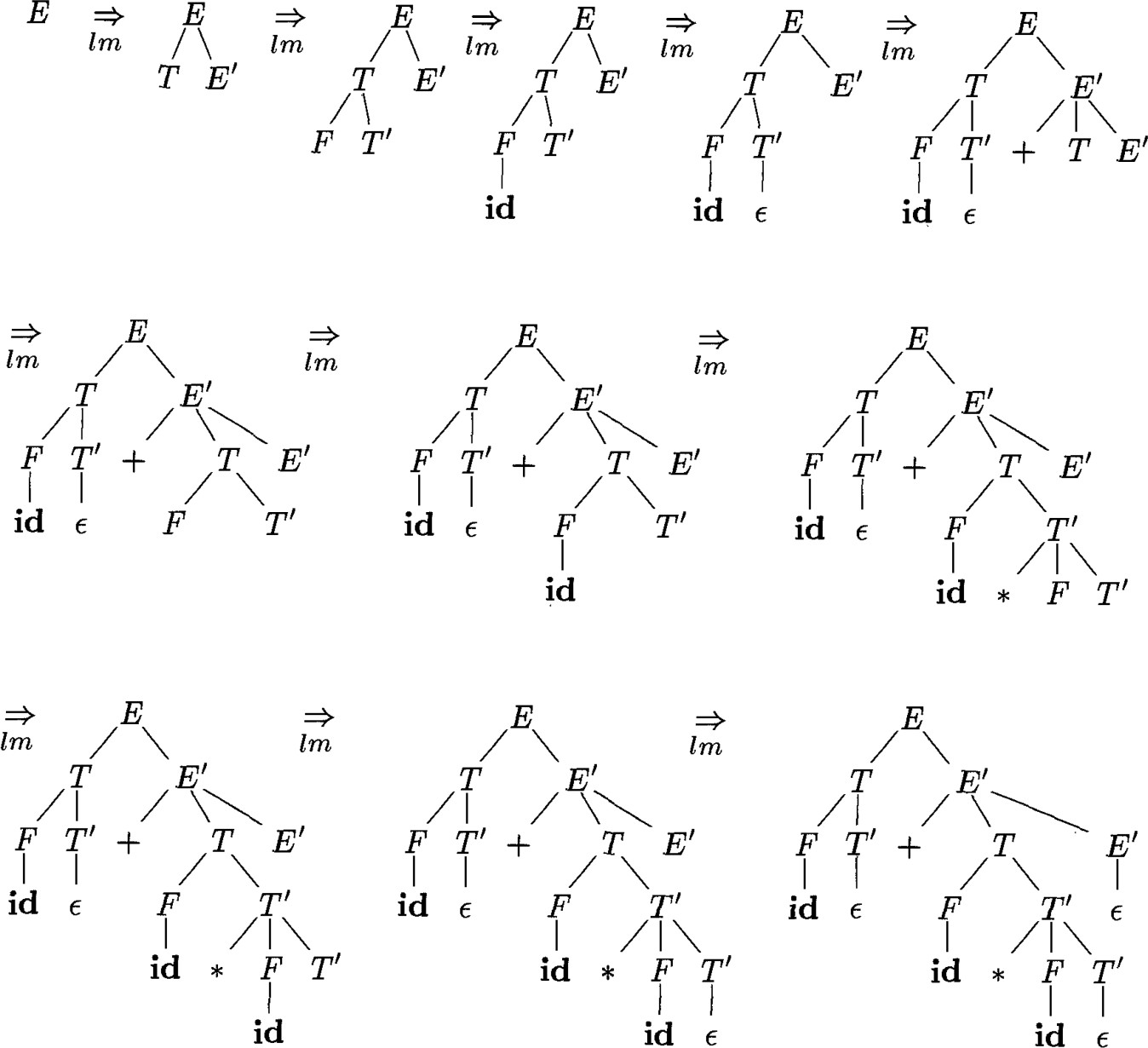
form an bm cn dm. anbm represents formal parameter list and cndm represents actual parameter list

# Top down parsing:

* Top down parsing can be viewed as problem of constructing parse tree for input, starting from root and creating nodes for parse tree in pre order.
* Top down parsing can be viewed as finding left most derivation for input string.
* At each step, determining production to be applied for non terminal say A is key problem. Once A production is chosen, rest of process consists matching terminals in production body with input.

Example: sequence of parse trees of Top down approach for input id + id \* id with E → T EI

EI → + T EI | ε T → F TI

TI → \* F TI | ε F → (E) | id

## Procedure for non terminal in top down parser:

Void A () {

1. Choose A productions A → X1X2X3 Xk;
2. For(i=1 to K) {
3. if(Xi is a nonterminal)
4. Call procedure Xi();
5. else if(Xi equal current input symbol a)
6. Advance the input to next symbol;
7. else /\*an error has occurred\*/;

}

}

Parsers

Top down parsers Bottom up parsers

Brute force approach predictive parser (or)

Recursive desent parser with backtracking

Recursive predictive parser Non Recursive predictive parser (or) (or)

Recursive desent parser Non Recursive desent parser (or) LL(1) without backtracking

* Problem with Top Down parsing are
  1. back tracking
  2. left recursion
  3. left factoring
  4. ambiguity

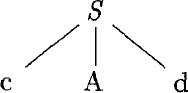
# Brute force approach:

* It requires back tracking: that is it may require repeated scans over input.
* Back tracking parsers are not seen frequently because reading input number of times may be complex and time consuming task.
* Brute force approach do not keep restriction on grammar that is grammar can have left recursion and left factoring

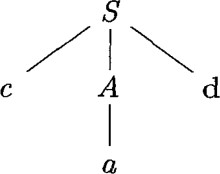
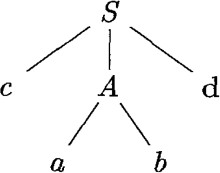
Example consider grammar

S → cAd

A → ab | a Construct parse tree for input string w=cad.

* Being with root element labelled S, and input pointer pointing to C first symbol of w. S has only one production expands it.
* Left most leaf labelled c, matches first symbol of input w. so, we advance pointer to a, second symbol of w and next leaf element labelled A.
* We expand A using first alternative, a match for second symbol a, so we advances pointer

to d, third input symbol and compare d against next has labelled b. b doesn’t match d. We report failure and go back to A to check another alternative.

* In going back to A, we must reset input pointer to position 2, then proceed with alternative production. To store input pointer position we use a local variable.
* In alternative production leaf a match 2nd symbol as W and leaf d match 3rd symbol of w then halt and announces completion of parsing.
* Biggest drawback of brute force in recursive descent with backtracking parser is, if one of a phase enters into infinite loop due to backtracking in left recursion lead compiler or machine to crash.

# Predictive Parsing:

* Predictive parse doesn’t allow grammar, that has left recursion, non deterministic and backtracking.
* Predictive parser is special type of recursive descent parser.
* It can predicted with production is suitable for completion of parsing based on input symbol.

## Recursive Predictive Parsing:

* + Recursive descent parsing program consists of set of procedures. One for each non terminal.
  + It consists input buffer contains string to be parsed, followed by end marker $.
  + It can be built by maintaining stack implicitly via recursive calls by using one input symbol of look ahead at each step to make parsing decision.

Example: Let us consider grammar E → id T

T → + id T | ɛ to parse input string id + id with recursive descent approach.

Procedure: E()

{

if (lookahead==’id’)

{

match (‘id’); T();

}

else

}

return;

T()

{

if (lookahead==’+’)

{

match(‘+’);

if (lookahead==’id’)

{

’);

}

match(‘id T();

else

}

return ;

else

}

return;

match(chart t)

{

if (lookahead==’t’) lookahead==next-token; else

printf(“error”)

}

main()

{

E();

if(lookahead==’$’) printf(“Success”) ;

}

## First and Follow:

* Construction of both top down and bottom up parser is by two functions, First and Follow, associated with grammar G.
* During top down 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 token.
* Design first(A), consider two A-productions A → α | β where first(α) and first(β) are disjoint sets .we can choose between these A-productions by looking at next input symbol

a. since a can be in at most one of first(α) and first(β) not both.

* To compose First(X) for all grammar symbols X, apply the following rules until no more terminals or ɛ can be added to any first set.

1. If X is a terminal then first(x) = {x}
2. If X is non terminal and X → Y1Y2.........Yk is a production for k >= 1 then add all non ɛ symbols of first(Y1) to first(X). If first(Y1) contains ɛ then add non ɛ symbols of first(Y2) to first(X) and so on.
3. If X → ɛ is production ,then add ɛ to first(X)

* Design follow(A), for non terminal A, to be set of terminals a that can appear immediately to right of A in some sentential form S⇒αAaβ, for some α and β. If A can be rightmost symbol in some sentential form then $ is in follow(A).
* To compute follow(A) for all non terminals A, apply the following rules until nothing can be added to any follow set.

1. Place $ in follow(S), where S is the starting symbol and $ is input right end marker.
2. If there is a production A → α B β, where first(β) doesn’t contains ɛ, then everything in first(β) is in follow(B)
3. If there is a production A → α B or A→ α B β, where first(β) contains ɛ, then everything in follow(A) is in follow(B)

Example: Consider grammar

E → T EI

EI → + T EI | ε T → F TI

TI → \* F TI | ε

F → (E) | id construct first and follow for the given grammar

Procedure:

Terminals = {+, \*, id, (, )}

Non-Terminals= {E, EI, T, TI, F} first of terminals:

first(+)={ + }

first(\*)={ \* } first(id)={ id } first(()={ ( }

first())={ ) } first(**ε**)={ **ε** }

first of non terminals: first(E):

first(E)=first(TEI) (∴ rule 2)

=first(T) =first(FTI) (∴ rule 2)

=first(F)

|  |  |
| --- | --- |
| F → (E)  (rule 2) first(F)=first((E))  ={ ( } | F → id (rule 1)  first(F)=first(id)  ={ id } |

∴ first(F)={ (, id }

∴ first(E) = first(T) = first(F) = {(,id}

first(EI):

|  |  |
| --- | --- |
| EI → + T EI  (rule 2) first(EI)=first(+TEI)  =first(+)  ={ + } | EI → **ε**  (rule 1) first(EI)=first(**ε**)  ={ **ε** } |

∴ first(EI)={ +, **ε** }

first(TI):

|  |  |
| --- | --- |
| TI → \* F TI  ( rule2) first(TI)=first(\*FTI)  =first(\*)  ={ \* } | TI → **ε**  (rule 1) first(TI)=first(**ε**)  ={ **ε** } |

∴ first (TI)={ \*, **ε** }

follow of non terminals:

: Non terminals={E, EI, T, TI, F} Follow(E):

|  |  |
| --- | --- |
| (rule 1)  If E is starting symbol follow(E)={ $ } | F→ (E)  (rule 2) follow(E)= first( ))  ={ ) } |

Follow(EI):

Follow(T):

∴ follow(E)={ $ } ⋃ { ) }

= { $, ) }

|  |  |
| --- | --- |
| E→ TEI  (rule 3)  follow (EI)=follow(E)  ={ $, ) } | EI→+TEI  (rule 3) follow(EI)=follow(EI)  ={ $, ) } |

∴ follow(EI)={ $, ) } ⋃ { $, ) }

={ $, ) }

∴ follow(E)=follow(EI)={ $, ) }

|  |  |
| --- | --- |
| E →TEI  (rule 2) follow(T)=first(EI) if ε is there  ={ first(EI) - { ε }} ⋃ follow(E)  ={{+, ε} - { ε }} ⋃ { $, ) }  ={ +, $ , ) } | EI→+TEI  (rule 2) follow(T)=first(EI) if ε is there  ={ first(EI) - { ε }} ⋃ follow(EI)  ={{+, ε} - { ε }} ⋃ { $, ) }  ={ +, $ , ) } |

∴ follow(T)={ +, $, ) } ⋃ { +, $ , ) }

= { +, $, ) }

∴ follow(T)={ +, $, ) }

Follow(TI):

|  |  |
| --- | --- |
| T→FTI  (rule 3) follow(T')=follow(T)  ={ +, $, ) } | TI→\*FTI  (rule 3) follow(T')=follow(T')  ={ +, $, ) } |

∴ follow (TI)={ +, $, ) }

Follow(F):

|  |  |
| --- | --- |
| T→FTI  (rule 2)  follow(F)= first(TI) if ε is there  ={ first(TI) - { ε }} ⋃ follow(T)  ={{\*, ε} - { ε }} ⋃ { +, $, ) }  ={ \*, +, $ , ) } | TI→ \*FTI  (rule 2)  follow(F)= first(TI) if ε is there  ={ first(TI) - { ε }} ⋃ follow(TI)  ={{\*, ε} - { ε }} ⋃ { +, $, ) }  ={ \*, +, $ , ) } |

∴ follow (F)= { \*, +, $, ) } ⋃ { \*, +, $ , ) }

= { \*, +, $, ) }

∴ follow(T)={ \*, +, $, ) }

|  |  |  |
| --- | --- | --- |
|  | First | Follow |
| E | (, id | $, ) |
| EI | +, **ε** | $, ) |
| T | (, id | +, $, ) |
| TI | \*, **ε** | +, $, ) |
| F | (, id | \*, +, $, ) |

# LL(1) grammar:

* Non recursive predictive parsing can be constructed for class as grammar called LL(1).
* First ’L’ in LL(1) stands for scanning input from left to right, second ’L’ for producing left most derivation and ’1’ for using one input symbol as look ahead at each step to make parsing decision.
* Grammar G is LL(1) if and only if whenever A→ α | β are two distinct productions as G ,by holding following conditions.

1. For terminal a, both α and β cannot drive strings beginning with a.
2. At most one as α and β can drive empty string.

* Next algorithm effects information from FIRST and FOLLOW sets into predictive passing table M [A, a], it is two-dimensional array, where A is no terminal, a is terminal and $ is end marker.

## Algorithm for construction of parsing table

INPUT: Grammar G. OUTPUT: Parsing table M .

Method: Each production A → α of grammar, do the following:

1. For each terminal a in first (α), add A → α to M [A, a].
2. If ε in first(α) ,then for each terminal b in follow(A) add A → α to M[A ,b].
3. There is no production at all in M [A, a] then set M [A, a] to error.

* In LL(1) grammar ,each table entry uniquely identifies a production or signal an error. Some grammar are not LL(1),because table entry multiply defined.

Example: Construct predictive passing table for the below grammar E → T EI

EI → + T EI | ε T → F TI

TI → \* F TI | ε F → (E) | id

Procedure:

“Construct FIRST and FOLLOW”

|  |  |  |
| --- | --- | --- |
|  | First | Follow |
| E | (, id | $, ) |
| EI | +, **ε** | $, ) |
| T | (, id | +, $, ) |
| TI | \*, **ε** | +, $, ) |
| F | (, id | \*, +, $, ) |

Production E → TEI is in the form A → α first(TE’)=first (T)

={ (, id }

Therefore E → TEI is place in M[ E, ( ] and M[ E, id ].

Production EI → +TEI

first(+TEI)={ + }

Therefore EI → +TEI is place in M[ E’, + ].

Production EI → ε

first(ε)={ ε }

Therefore E’→ ε is placed in M[ EI, follow(EI) ]

E’→ ε is placed in M[ EI, ( ] and M[ EI, id ].

Production T → FTI

first(FTI)=first(F)

={ (, id }

Therefore T → FTI is placed in M[ T, ( ] and M[ T, id ].

Production TI → \*FTI

first(\*FTI)=first(\*)

={ \* }

Therefore TI → \*FTI is placed in M[ TI, \* ].

Production TI → ε

first(ε)={ ε }

Therefore TI → ε is placed in M[ TI, FOLLOW(TI) ]

TI → ε is placed in M[ TI, + ], M[ TI, $ ] and M[ TI, ) ].

Production F → (E)

first((E))= first (()={ ( } Therefore F → (E) is placed in M[ F, ( ].

Production F → id

first(id)={id}

Therefore F → id is placed in M[ F, id ].

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Non Terminal | Input Symbols | | | | | |
| id | + | \* | ( | ) | $ |
| E | E → TEI |  |  | E → TEI |  |  |
| EI |  | EI → +TEI |  |  | EI → ε | EI → ε |
| T | T → FTI |  |  | T → FTI |  |  |
| TI |  | TI → ε | TI → \*FTI |  | TI → ε | TI→ ε |
| F | F → id |  |  | F → (E) |  |  |

Therefore The given grammar is LL(1) because every entry in table is unique production.

Example: consider a grammar

S → i E t S SI | a SI → e S | ε

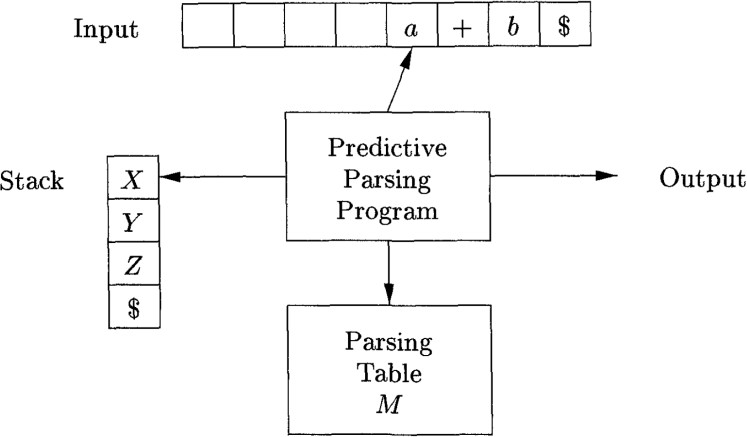
E → b

Check whether the given grammar is LL(1) or not

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Non Terminal | Input Symbols | | | | | |
| a | b | e | i | t | $ |
| S | S → a |  |  | S → i E t S SI |  |  |
| SI |  |  | SI → ε SI→ e S |  |  | SI → ε |
| E |  | E → b |  |  |  |  |

Therefore Given grammar is not LL(1) because multiple production entries in M[SI,e].

# Non recursive predictive parsing:

* Non recursive predictive parser can be built by maintaining stack explicitly rather than implicitly via recursive calls.
* w is input that has been matched so far, stack holds sequence of grammar symbols.
* Double driven parser has input buffer, stack a parsing table constructed based on LL(1)grammar.
* Input buffer contains string to parse followed by end marker $. we use symbol $ to mark the bottom of the stack and initially top of stack is starting symbol.
* Program considers X, top of stack, a is current input symbol, then parser chooses x productions by consulting M[X, a] in parsing table m. otherwise check for matching to input if X is terminal.

## Algorithm for table driven predictive parsing:

Input: string W and parsing table M for grammar G. Output: if w is in L [G] leftmost derivation of W or error.

Method: program takes input and parsing table for parsing the input. let a be first symbol of W

let X be top of stack

while(X!= $) {/\*stack is not empty\*/

if(X=a) pop the stack and a be the next symbol of W. else if(X is terminal) error();

else if(M [X, a] is error entry) error(); else if(M[X, a] = X→Y1Y2 Yk) {

pop the stack;

push Yk,Yk-1 Y1, on to stack with Y1 on top.

}

let X be top stack symbol;

}

Example: consider grammar

E → T EI

EI → + T EI | ε T → F TI

TI → \* F TI | ε F → (E) | id

Construct parsing table and check id+id\*id string is accepted by grammar or not.

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| $ E | id + id \* id $ | output E → T EI |
| $ EI T | id + id \* id $ | output T → F TI |
| $ EI TI F | id + id \* id $ | output F → id |
| $ EI TI id | id + id \* id $ | match id |
| $ EI TI | + id \* id $ | output TI → ε |
| $ EI | + id \* id $ | Output EI → + T EI |
| $ EI T + | + id \* id $ | match + |
| $ EI T | id \* id $ | output T → F TI |
| $ EI TI F | id \* id $ | output F → id |
| $ EI TI id | id \* id $ | match id |
| $ EI TI | \* id $ | output TI → \* F TI |
| $ EI TI F \* | \* id $ | match \* |
| $ EI TI F | id $ | output F → id |
| $ EI TI id | id $ | match id |
| $ EI TI | $ | output TI → ε |
| $ EI | $ | output EI → ε |
| $ | $ | accepted |

# Error recovery in predictive parsing:

* Error is detected during predictive parsing when top of stack does not match next input symbol or M [A, a] is error.
* Error recovery in predictive parsing is done in 2 ways.
  1. Panic mode
  2. phase level recovery

## Panic mode:

* It is based on idea of skipping over symbols on input until a set of synchronizing tokens appear.
* Some ways are as follows.

1. place all symbols in follow(A) in to the synchronizing set for non terminal. If we skip tokens until an element of follow(A) is seen and pop A from stack.
2. if we add symbols in follow(A) to synchronizing set for non terminal A, then it may possible to resume parsing to A if symbol in first(A) appears in input.
3. if non terminal can generate empty string, then deriving e can be used as default. It postpones some error but not missed.
4. if terminal on top of stack cannot be matched, simple idea is to pop terminal, issue message saying terminal was inserted.

Example: consider grammar

E → T EI

EI → + T EI | ε T → F TI

TI → \* F TI | ε F → (E) | id

Construct parsing table and apply follow and first symbols into synchronizing sets and check the acceptance of )id\*+id.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Non Termi | Input Symbols | | | | | |
| id | + | \* | ( | ) | $ |
| E | E → TEI |  |  | E → TEI | synch | synch |
| EI |  | EI → +TEI |  |  | EI → ε | EI → ε |
| T | T → FTI | synch |  | T → FTI | synch | synch |
| TI |  | TI → ε | TI → \*FTI |  | TI → ε | TI→ ε |
| F | F → id | synch | synch | F → (E) | synch | synch |

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| $ E | ) id \* + id $ | error , skip ) id is in first(E)  error , M[F, + ] = synch, F has been popped |
| $ E | id \* + id $ |
| $ EI T | id \* + id $ |
| $ EI TI F | id \* + id $ |
| $ EI TI id | id \* + id $ |
| $ EI TI | \* + id $ |
| $ EI TI F \* | \* + id $ |
| $ EI TI F | + id $ |
| $ EI TI | + id $ |
| $ EI | + id $ |
| $ EI T+ | + id $ |
| $ EI T | id $ |
| $ EI TI F | id $ |
| $ EI TI id | id $ |
| $ EI TI | $ |
| $ EI | $ |
| $ | $ |

## Phrase Level recovery:

* It is implemented by filling in the blank entries by error routines. These routines may change, insert or delete symbols on input and issue error messages.
* They pop from stack alteration of stack symbols or inserting new symbols on stack is not supported for several reasons.

1. Steps carried out by parser might then not correspond to derivation of any word in language.
2. We must ensure that there is no possibility of an infinite loop.

**Types of Grammar:**

**Type 0(Unrestricted Grammar):**

* If there is no restriction on any grammar then that grammar is categorized as type 0 or Un- restricted grammar.
* In this grammar, non terminal and terminals in production has no limit.

α → β

α ∊ (N+T)+

β ∊ (N+T)\* Example: aAb → bB

aA → **ε**

**Type 1 (Context Sensitive):**

* Apply some restrictions to type 0 grammar is called as type 1 or Context sensitive grammar.
* Context sensitive means before and after of non terminal should be a terminal or non terminal.
* Right hand side of grammar is always greater than in length of length hand side.

α → β (∴|α|<=|β|) α, β ∊ (N+T)+

Example: aAb → bbb

aA → bB

**Type 2(Context Free):**

* Apply context free restriction on type 1 grammar is called Context free grammar.
* Context free means before or after of any non terminal on left hand should be empty.
* In this grammar, left hand side of a production should be only one non terminal.

α → β (∴|α|=1) β ∊ (N+T)\*

Example: A → BCD

B → a

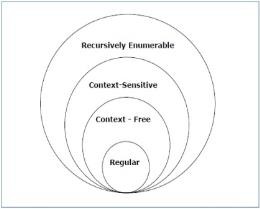
**Type 3(Regular):**

* If any grammar is left linear, right linear and middle linear then it is called linear grammar.
* If any grammar is left linear and right linear, but not middle linear then it is called regular grammar.

Examples: A → xB/y (∴ RL)

A → Bx/y (∴ LL)

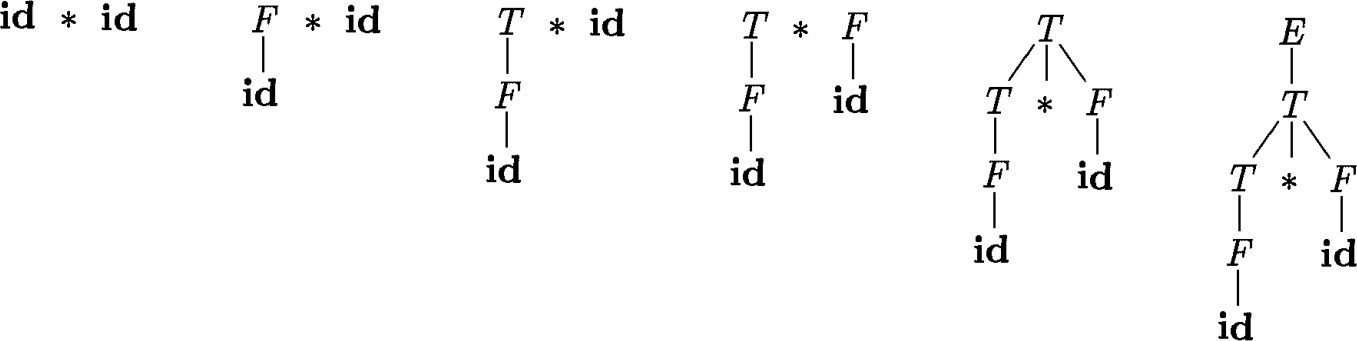
A, B ∊ N

x, y ∊ T\*

# Bottom up parsing:

* + Bottom up parse corresponds to the construction of parse tree for input string beginning at leaves and working up towards root.
  + Largest class of grammars for which shift reduce parser can be built is LR grammars
  + It is too much work to built LR parser by hand, tools like automated parser generators make it is to construct LR parser from suitable grammars.

Example: Sequence of parse tree of bottom up approach for input id \* id with E → E + T | T

T → T \* F | F F → (E) | id

Parsers

Top down parsers Bottom up parsers

Shift reduce parser

Operator procedure

parsing

SLR CLR LALR

## Reduction:

* + bottom up parsing is process of reducing string w to start symbol of grammer.at each reduction step ,substring matching body is replace by head of production.
  + Key decisions during bottom up parsing are about when to reduce and about what production to apply.

Example: consider grammar

E → E + T | T

T → T \* F | F F → (E) | id

and sequence of reduction of above grammar for string id\*id is

Id \* id F \* id T \* id T \* F T E

## Handle pruning:

* + Handle is substring that matches the body of production and whose reduction represents one step along the reverse of right most derivation.
  + Process of replacing handle with head of production is called handle parsing.
  + Leftmost substring that matches the body of some production need not be handle.

|  |  |  |
| --- | --- | --- |
| Right Sentential Form | Handle | Reducing Production |
| id1 \* id2 | id1 | F → id |
| F \* id2 | F | T → F |
| T \* id2 | Id2 | F → id |
| T \* F | T \* F | T → T \* F |
| T | T | E → T |
| E |  |  |

## Shift Reduce parsing:

* + It is the form of bottom up parsing in which stack holds grammar symbols and input buffer holds rest of string to be parsed.
  + Handle always appear at top of stack. We use $ to make bottom of stack and right end of input.
  + Initially, stack is empty and string w on input.

Stack input

$ w$

* + During parsing, parser shifts zero or more input symbols onto stack, until it is ready to reduce, then reduce it with head of production.
  + The parser repeats cycle until it has detected an error or until stack contain start symbol and input is empty.

Stack input

$S $

* + there are actually four possible actions shift reduce parser can make is as follows

1. Shift: shift next input symbol on to top of stack
2. Reduce: right end of string to be reduced with head of production, which matches the body of that production.
3. Accept: Announce successful completion of string.
4. Error: Discover syntax error and call error recovery method. Example: consider grammar

E → E + T | T

T → T \* F | F F → (E) | id

then check the acceptance of id\*id with shift-reduce parser.

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| $ | id1 \* id2 $ | shift |
| $ id1 | \* id2 $ | reduce F → id |
| $ F | \* id2 $ | reduce T → F |
| $ T | \* id2 $ | shift |
| $ T \* | id2 $ | shift |
| $ T \* id2 | $ | reduce F → id |
| $ T \* F | $ | reduce T → T \* F |
| $ T | $ | reduce E → T |
| $ E | $ | accept |

## Conflicts during shift reduce parsing:

* + In shift reduce parsing; parser cannot decide which action to be taken. this situation is called conflict.
  + Two types of conflicts should be possible in shift reduce parser.
  1. shift/reduce conflict
  2. reduce/reduce conflict

## Shift or reduce conflict:

* + Passive cannot decide whether to use shift or reduce conflict will occur. stack input

$ E+T \*id$

* + To solve the above problem, we take actions based on operator precedence and associativity.

## Reduce/Reduce conflict:

* + Parser cannot decide which one of several reductions will use, then reduce/reduce conflict will occur.

stack input

$ E+T\*F $

* + To solve the above problem, we will take action based on rightmost elements of stack should reduce first.
  + These conflicts will concentrate for those grammars which are not LR or those grammars are ambiguous.