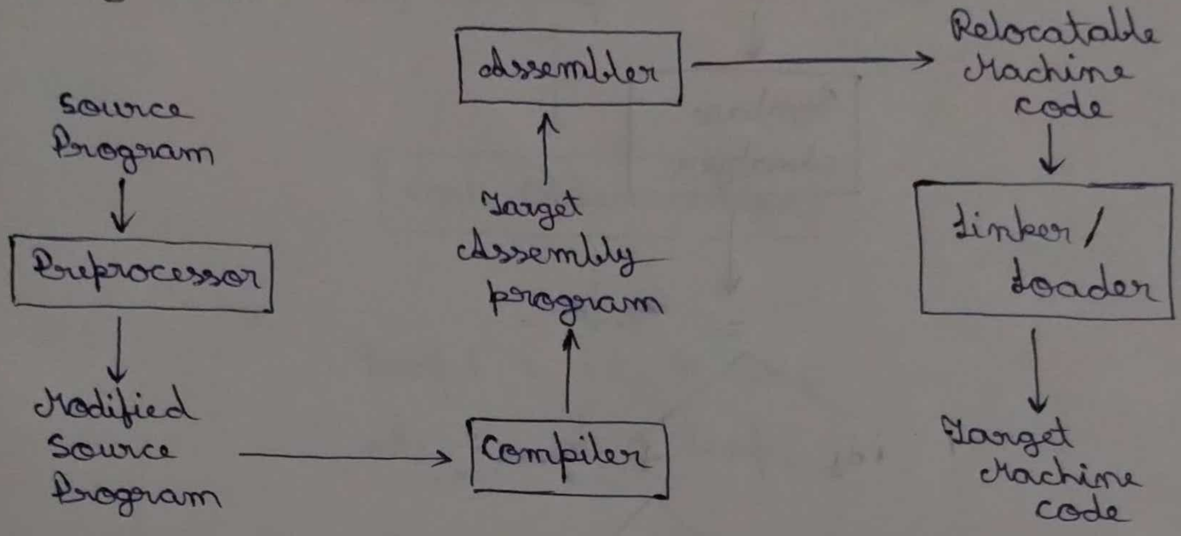
25/07/2023

- Compiler is basically a translator.
  It translates a source program to a target program of a certain processor.

- Interpretor is a language processor which executes line by line.



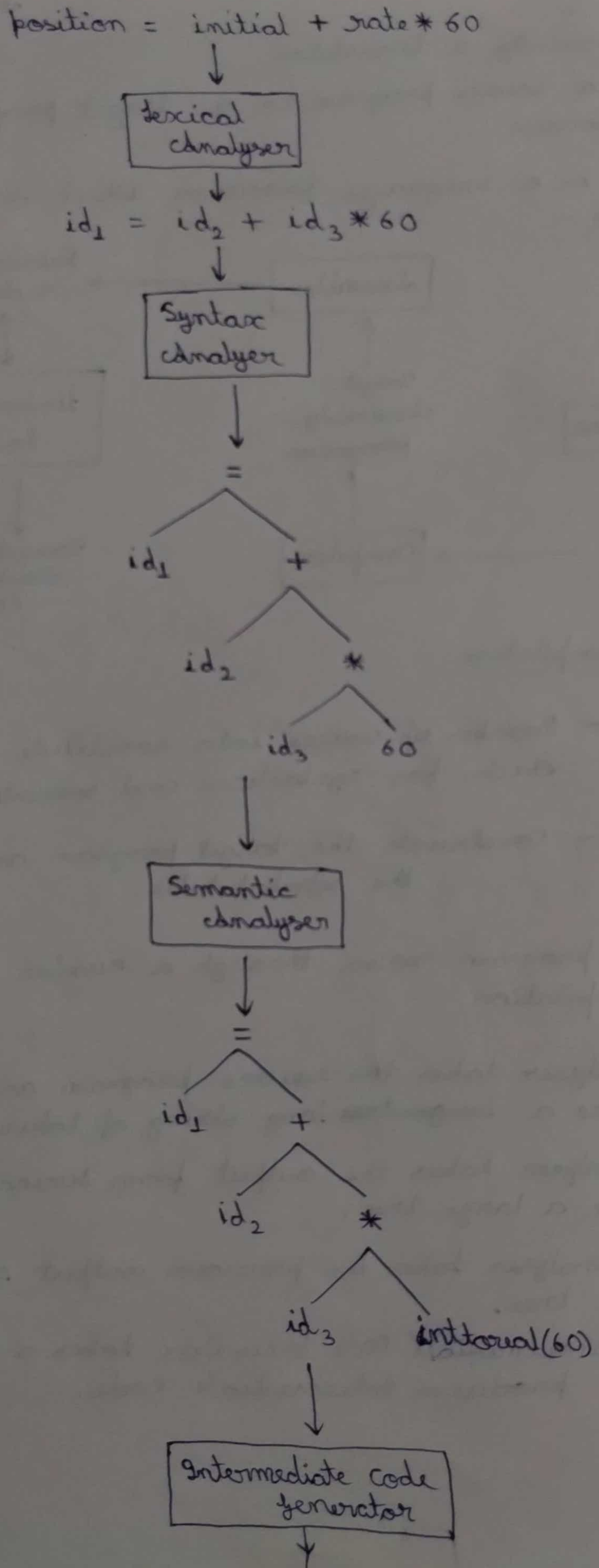

- <u>Models</u> of <u>Compilation</u>

  Analysis – Breakes up source into constitute pices checks for syntactical and semantic errors

  Synthesis – Constructs the target program and the symbol table

- The source program moves through a number of phases during compilation.

- Lexical Analyser takes the source program as input and produce a ~~large tree~~ long string of tokens.
- Syntax Analyser takes the output from lexical Analyser and produce a large tree.
- Semantic Analyser takes the previous output and produces another tree.
- Similarly, Intermediate code generator takes a tree as an input and produces intermediate code.


position = initial + rate * 60
Lexical Analyser
id1 = id2 + id3 * 60
Syntax Analyser
=
id1
+
id2
*
id3
60
Semantic Analyser
=
id1
+
id2
*
id3
inttoreal(60)
Intermediate code generator

$temp1 = inttoreal(60)$

$temp2 = id_3 * temp1$

$temp3 = id_2 + temp2$

$id_1 = temp3$

$\boxed{\text{code optimisation}}$

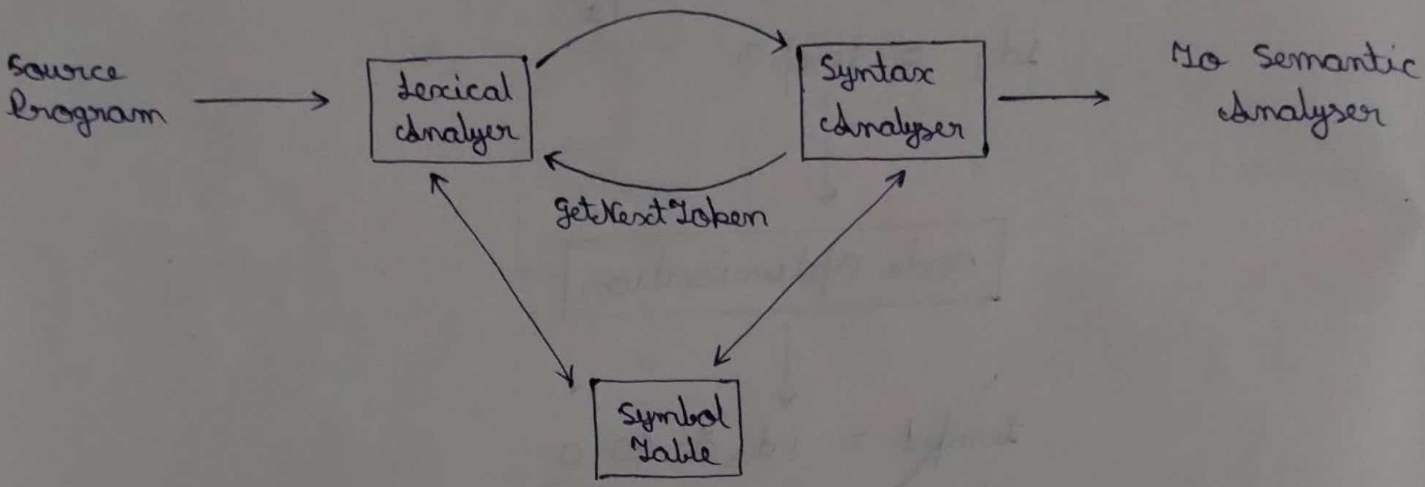
$temp1 = id_3 * 60.0$

$id_1 = id_2 + temp1$

Textbook:

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman "compilers Principles Techniques and Tools"

- Lexical Analyser




→ Reads input characters and produce a series of tokens as output

Token : It is a pair consisting of a token name and an optional attribute value.

Pattern : The set of strings is described by a rule called pattern associated with that token.

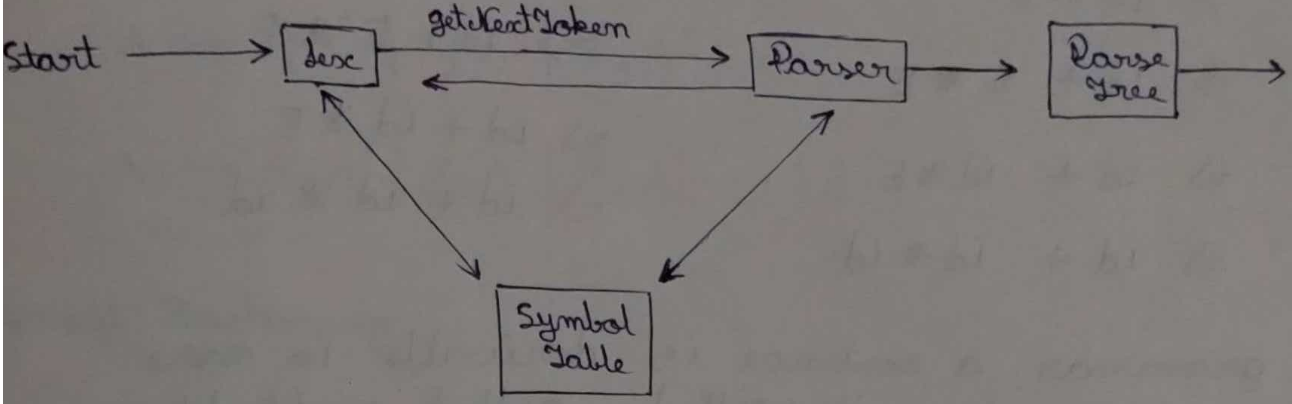
Lexeme : It is a sequence of characters in the source program that matches the pattern for a token and is identified by a lexical analyser as an instance of a pattern.

- Tokens can be specified using :
  (i) Regular Expression
  (ii) DFA
  (iii) NDFA
  (iv) NDFA with Empty Transitions

- We have to draw transition diagrams and then write programs for them.

01/08/2023

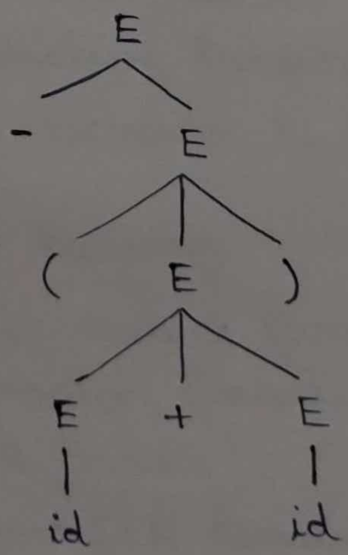
## · Syntax Analyzer




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

$$
\begin{aligned}
E &\Rightarrow -E \\
&\Rightarrow -(E) \\
&\Rightarrow -(E+E) \\
&\Rightarrow -(id+E) \\
&\Rightarrow -(id+id)
\end{aligned}
$$

So, $-(id+id)$ can be derived from E.



Q. Derive id + id * id

Ans. :

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

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

If by a grammar, a sentence is derivable in more than one way, then it will be called an 'Ambigous grammer'.

- Elimination of Left Recursion

$$A \rightarrow A\alpha \mid \beta .$$

This will generate left recursion.

This rule can be rewritten as:

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

Q. $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Remove left recursions.

$\Rightarrow$

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

If a production rule is like,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid \beta_1 \mid \beta_2 \cdots$$

then, after removing left recursion, it will look like,

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \cdots$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \cdots \mid \varepsilon$$

- Left Factoring

If $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ is the production rule, Left Factoring says that,

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

This is done when we cannot choose what production rule to use, for derivation.

⊙ Approaches in parsing –

(a) Top-down Parsing
- → Recursive Descent Parsing
- → Predictive Parsing
- → Non-recursive Predictive Parsing

(b) Bottom-up Parsing
- → Shift-reduce Parsing
- → Operator Precedence Parsing
- → LR Parser
  - → SLR Parsing Table
  - → Look-ahead LR Parsing Table
  - → Canonical LR Parsing Table

03/08/2023

# Top-Down Parsing

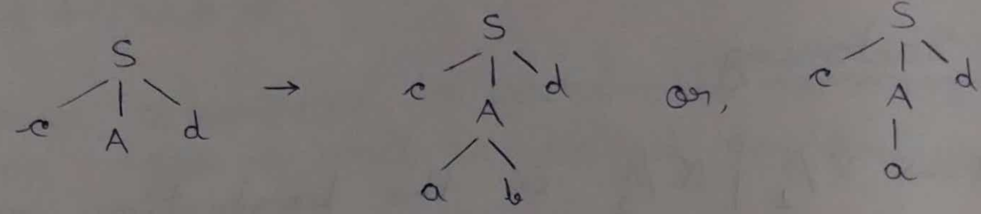
## (i) Recursive Descent Parsing

Given $S \rightarrow cAd$

$A \rightarrow ab \mid a$

Input string, $w = cad$

Derivation :




## (ii) Predictive Parsing

Given $S \rightarrow cAd$

$A \rightarrow ab \mid a$

Input string, $w = cad$

- In predictive parsing, there is no back-tracking.
- Modifying the production rules,

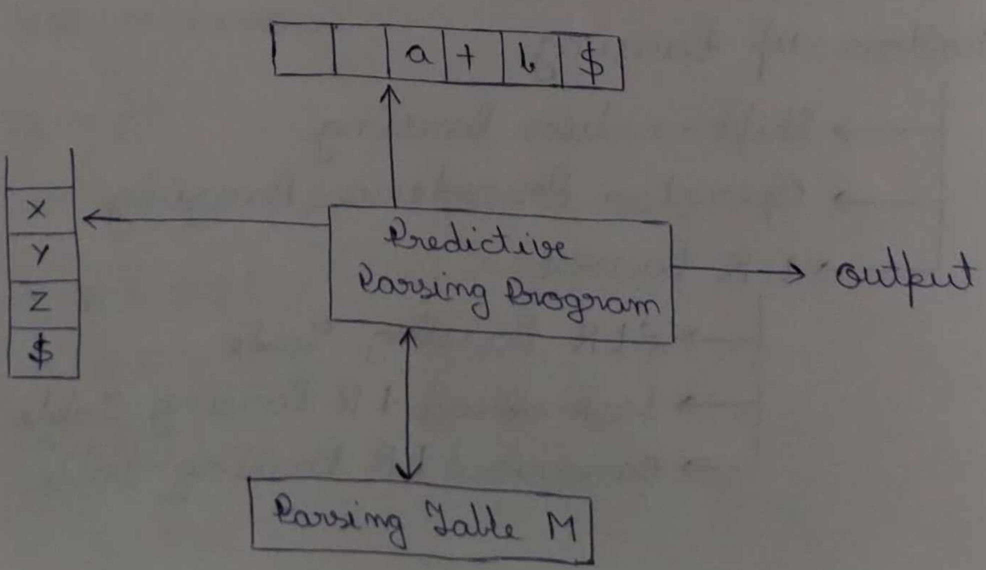
$S \rightarrow cAd$

$A \rightarrow aA'$

$A' \rightarrow b \mid \varepsilon$

This way there is only one good way to generate string.

## (iii) Non-recursive Predictive Parsing



The Parsing Table will help in choosing the production rule whereas the output will say if the string is accepted or not.

The program should implement,

(a) if $X = a = \$$, the parser halts and announces successful completion of parsing

(b) if $X = a \neq \$$, the parser pops-off the stack and advances the input pointer to the next input symbol

(c) If X is a non-terminal symbol, then program consults the Parsing Table

Example :

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Forming the parsing table :

The row-headers are non-terminal symbols whereas the column-headers are terminal symbols and right-end marker.

| | id | + | * | ( | ) | \$ |
|---|---|---|---|---|---|---|
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| E' | | $E' \rightarrow +TE'$ | | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ | | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | | | $F \rightarrow (E)$ | | |

| Stack | Input Buffer | Outputs |
|---|---|---|
| $\$E$ | $id + id * id \$$ | |
| $\$E'T$ | $id + id * id \$$ | $E \to TE'$ |
| $\$E'T'F$ | $id + id * id \$$ | $T \to FT'$ |
| $\$E'T'id$ | $id + id * id \$$ | $F \to id$ |
| $\$E'T'$ | $+ id * id \$$ | |
| $\$E'$ | $+ id * id \$$ | $T' \to \epsilon$ |
| $\$E'T+$ | $+ id * id \$$ | $E' \to +TE'$ |
| $\$E'T$ | $id * id \$$ | |
| $\$E'T'F$ | $id * id \$$ | $T \to FT'$ |
| $\$E'T'id$ | $id * id \$$ | $F \to id$ |
| $\$E'T'$ | $* id \$$ | |
| $\$E'T'F*$ | $* id \$$ | $T' \to *FT'$ |
| $\$E'T'F$ | $id \$$ | |
| $\$E'T'id$ | $id \$$ | $F \to id$ |
| $\$E'T'$ | $\$$ | |
| $\$E'$ | $\$$ | $T' \to \epsilon$ |
| $\$$ | $\$$ | $E' \to \epsilon$ |

10/08/2023

- construction of 'Parsing Table' for non-recursive predictive parser :

Two functions are needed,

FIRST(X) → Set of Terminals

FOLLOW(A) → Set of Terminals

Rules for FIRST( ) :

1. If X is a terminal symbol, then FIRST(X) is {X}
2. If $X \rightarrow \epsilon$ is a production rule, then add $\epsilon$ to FIRST(X)
3. If X is a non-terminal symbol and
   $X \rightarrow Y_1 Y_2 Y_3 \ldots$ is a production rule, then
   place FIRST($Y_1$) to FIRST(X).

Example :

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

$$\begin{aligned} FIRST(E) &= FIRST(TE') \\ &= FIRST(T) \\ &= FIRST(FT') \\ &= FIRST(F) \\ &= \{(, id\} \end{aligned}$$

$$\therefore FIRST(E) \equiv FIRST(T) \equiv FIRST(F) = \{(, id\}$$

$$FIRST(E') = \{+, \epsilon\}$$
$$FIRST(T') = \{*, \epsilon\}$$

Rules for FOLLOW( ):

1. Place \$ in FOLLOW(S) where S is the start symbol and \$ is the input right-end marker
2. If there is a production rule $A \to \alpha B \beta$, then everything in FIRST($\beta$) except for $\epsilon$, is placed in FOLLOW(B)
3. If there is a production $A \to \alpha B \ldots$ or a production rule $A \to \alpha B \beta$ where FIRST($\beta$) contains $\epsilon$ (i.e. $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B)

In the previous example,

FOLLOW(E) = { \$ }

considering $F \to (E)$

FIRST( ) ) = { ) }

∴ FOLLOW(E) = { \$, ) }

FOLLOW(E') has to be found out.

$$E \to \underbrace{T}_{\alpha}\underbrace{E'}_{B}\underbrace{}_{\beta}$$

By rule 3, FOLLOW(E') = { \$, ) }

FOLLOW(T) has to be found out.

$$E \to \underbrace{}_{\alpha}\underbrace{T}_{B}\underbrace{E'}_{\beta}$$

FIRST($\beta$) has empty symbol.

So, FOLLOW(E) will be in FOLLOW(T)

Now, $E' \to \underbrace{+}_{\alpha}\underbrace{T}_{B}\underbrace{E'}_{\beta}$

By rule 2, '+' will be in FOLLOW(T)

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

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

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

- LL(1) grammer

A grammar is said to be LL(1) if the parsing table does not contain multiple entries in one cell.

Example 1: $S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

Example 2: $E \to E+T \mid T$

$T \to TF \mid F$

$F \to F* \mid a \mid b$

- extension for input is '.l'.

definitions
%%
rules
%%
user code

%{ <something> %} for verbatim copying

17/08/2023

## Bottom-up Parsing

We start from the last symbol, and if we can reach the start one, the string is accepted.

(i) Shift Reduce Parsing (It's a model)

Q. Given, $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

check if 'abbcde' is acceptable.

Method : (i) Choose a sub-string which matches the right-hand side of a production rule. This will be called 'Handle'.

(ii) Keep doing it

a $\underline{b}$ b c d e

a $\underline{Abc}$ d e ..... $A \rightarrow b$ is the handle

a A $\underline{d}$ e ..... $A \rightarrow Abc$ is the handle

$\underline{aABe}$ ..... $B \rightarrow d$ is the handle

S ..... $S \rightarrow aABe$ is the handle

Handle : Sub-string that matches the right side of a production rule and reduction is done through the non-terminal.

Handle Pruning : A right most derivation in reverse is known as handle pruning.

Q. Given, $E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

The input sentence is $id_1 + id_2 * id_3$.

| Right Sentential form | Handle | Reducing Production |
|---|---|---|
| $id_1 + id_2 * id_3$ | $id_1$ | $E \rightarrow id$ |
| $E + id_2 * id_3$ | $id_2$ | $E \rightarrow id$ |
| $E + E * id_3$ | $id_3$ | $E \rightarrow id$ |
| $E + E * E$ | $E * E$ | $E \rightarrow E * E$ |
| $E + E$ | $E$ | $E \rightarrow E + E$ |
| $E$ | | |

-> Stack Implementation of Shift-reduce Parser

| Stack | Input Buffer | Action |
|---|---|---|
| $\$$ | $id_1 + id_2 * id_3 \$$ | Shift |
| $\$ id_1$ | $+ id_2 * id_3 \$$ | Reduce by $E \rightarrow id$ |
| $\$ E$ | $+ id_2 * id_3 \$$ | Shift |
| $\$ E +$ | $id_2 * id_3 \$$ | Shift |
| $\$ E + id_2$ | $* id_3 \$$ | Reduce by $E \rightarrow id$ |
| $\$ E + E$ | $* id_3 \$$ | Reduce by $E \rightarrow E + E$ |
| $\$ E$ | $* id_3 \$$ | Shift |
| $\$ E *$ | $id_3 \$$ | Shift |
| $\$ E * id_3$ | $\$$ | Reduce by $E \rightarrow id$ |
| $\$ E * E$ | $\$$ | Reduce by $E \rightarrow E * E$ |
| $\$ E$ | $\$$ | Accepted |

This is done manually, so reduction can be done like this or also after getting $\$ E + E * E$ in the stack. Although the later is preferable.

There are 4 actions here —

(a) Shift

(b) Reduce

(c) Accept

(d) error

(ii) Operator Precedence Parsing

→ operator grammar

(a) There will be no empty producting at right

(b) Has no production right side with two adjacent non-terminals

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

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

This is not operator grammar.

But,

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

is an operator grammar.

We will use some Precedence rules here.

| Relation | Meaning |
|---|---|
| $a <\cdot b$ | a yields precedence to b |
| $a \doteq b$ | a has same precedence as b |
| $a \cdot> b$ | a takes precedence over b |

These three disjoint relations are required to form the parsing table.

In the table, row and column headers are there, which will be same, only ~~non~~ terminals.

| | id | + | − | * | \$ |
|---|---|---|---|---|---|
| id | | $\gtrdot$ | $\gtrdot$ | $\gtrdot$ | $\gtrdot$ |
| + | $\lessdot$ | $\gtrdot$ | | $\lessdot$ | $\gtrdot$ |
| − | | | | | |
| * | $\lessdot$ | $\gtrdot$ | | $\gtrdot$ | $\gtrdot$ |
| \$ | $\lessdot$ | $\lessdot$ | | $\lessdot$ | |

Formation is problem specific, that is, it's heuristically chosen.

$$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$$

Take two pointers and mark the predence.

Then remove all non-terminals.

$$\$ \ E \ + \ E \ * \ E \ \$$$

$$\Downarrow$$

$$\$ \lessdot + \lessdot * \gtrdot \$$$

So, we choose E*E as handle and reduce it.

$$\$ E + E \$$$

Remove non-terminals again.

$$\$ \lessdot + \gtrdot \$$$

So, we choose E+E as handle and reduce it.

$$\$ E \$$$

(accepted)

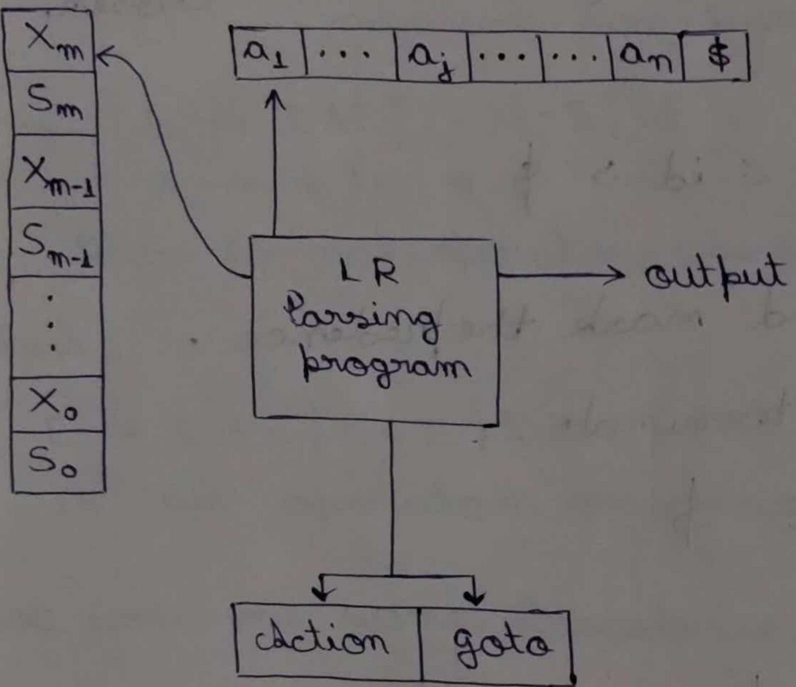
## (iii) LR Parsing

LR(K) Parsing is a general model. It will be followed by all types of LR parsers.

L → left to right scanning of the input

R → for constructing a right most derivation in reverse

K → the number of input symbols of lookahead that are used in making parsing decision




whenever we have a CFG, we can apply LR(K) parsing.

There are some actions –

1. Shift [K number of symbols are pushed to stack]
2. Reduce by a grammar production rule
3. Accept
4. Error

<u>Example</u> : $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

| state | action | | | | | | goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | $S_5$ | | | $S_4$ | | | 1 | 2 | 3 |
| 1 | | $S_6$ | | | | acc | | | |
| 2 | | $r_2$ | $S_7$ | | $r_2$ | $r_2$ | | | |
| 3 | | $r_4$ | $r_4$ | | $r_4$ | $r_4$ | | | |
| 4 | $S_5$ | | | $S_4$ | | | 8 | 2 | 3 |
| 5 | | $r_6$ | $r_6$ | | | $r_6$ | | | |
| 6 | $S_5$ | | | $S_4$ | | | | 9 | 3 |
| 7 | $S_5$ | | | $S_4$ | | | | | 10 |
| 8 | | $S_6$ | | | $S_{11}$ | | | | |
| 9 | | $r_1$ | $S_7$ | | $r_1$ | $r_1$ | | | |
| 10 | | $r_3$ | $r_3$ | | $r_3$ | $r_3$ | | | |
| 11 | | $r_5$ | $r_5$ | | $r_5$ | $r_5$ | | | |

Fig: Parsing Table

| Stack | Input | Action |
|---|---|---|
| 0 | id + id * id $ | Shift |
| 0 id 5 | + id * id $ | Reduce by F→id |
| 0 F 3 | + id * id $ | Reduce by T→F |
| 0 T 2 | + id * id $ | Reduce by E→T |
| 0 E 1 | + id * id $ | Shift |
| 0 E 1 + 6 | id * id $ | Shift |
| 0 E 1 + 6 id 5 | * id $ | Reduce by F→id |
| 0 E 1 + 6 F 3 | * id $ | Reduce by T→F |
| 0 E 1 + 6 T 9 | * id $ | Shift |
| 0 E 1 + 6 T 9 * 7 | id $ | Shift |
| 0 E 1 + 6 T 9 * 7 id 5 | $ | Reduce by F→id |
| 0 E 1 + 6 T 9 * 7 F 10 | $ | Reduce by T→T*F |
| 0 E 1 + 6 T 9 | $ | Reduce by E→E+T |
| 0 E 1 | $ | Accept |

When we reduce by a rule, twice the no. of symbols on the right side of the rule has to be popped from stack.

-> LR Grammar

Constructing the SLR Parsing Table :-

First, we have to find the LR(0) items.

LR(0) items → a dot at some position of the RHS of production rules

For $A \to XYZ$, LR(0) items are

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

We have to perform 'closure' and 'goto' actions on the Canonical LR(0) items.

We have to introduce a new start symbol $S'$. This is called 'Augmenting' the grammar.

If there are many S productions like,

$$S \to X_1$$
$$S \to X_2 \quad \text{and so on,}$$

it will be difficult to know when to stop. So, we augment the grammar and parser will stop when $S' \to S$ is encountered.

Eg. : $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Now, we augment the grammar,

$$S' \to E$$

$I = \{[S' \to \cdot E]\}$

Closure$(I) = \{[S' \to \cdot E], [E \to \cdot E+T], [E \to \cdot T], [T \to \cdot T*F], [T \to \cdot F], [F \to \cdot (E)], [F \to \cdot id]\}$

Take production rules whenever new non-terminals appear in right-hand side.

$I_0$ = closure (I)

Now, we have to find goto($I_0$, E),

consider all productions where '·E' is on RHS.

$I_1$ = goto ($I_0$, E)

= { [S' → E·], [E → E·+T] }

Then, $I_2$ = goto ($I_0$, T)

= { [E → T·], [T → T·*F] }

Then, $I_3$ = goto ($I_0$, F)

= { [T → F·] }

Then $I_4$ = goto ($I_0$, ( )

= { [F → (·E)] } U closure(E)

since E is non terminal.

∴ $I_4$ = { [F → (·E)], [E → ·E+T], [E → ·T],

[T → ·T*F], [T → ·F], [F → ·(E)], [F → ·id] }

Then $I_5$ = goto ($I_0$, id)

= { [F → id·] }

Then $I_6$ = goto ($I_1$, +) = { [E → E+·T] } U closure(T)

= { [E → E+·T], [T → ·T*F], [T → ·F],

[F → ·(E)], [F → ·id] }

Then $I_7$ = goto ($I_2$, *)

= { [T → T*·F] } U closure(F)

= { [T → T*·F], [F → ·(E)], [F → ·id] }

Then goto $(I_4, E) = I_8$

$= \{[F \to (E\cdot)], [E \to E\cdot + T]\}$

Then $I_9 = goto(I_4, T)$

$= \{[E \to T\cdot], [T \to T\cdot * F]\} = I_2$

Then $I_{10} = goto(I_4, F)$

$= \{[T \to F\cdot]\} = I_3$

Then $I_{11} = goto(I_4, ()$

$= \{[F \to (\cdot E)]\} \cup closure(E)$

$= I_4$

Then $I_{12} = goto(I_4, id)$

$= \{[F \to id\cdot]\} = I_5$

Then $I_{13} = goto(I_6, T)$

$= \{[E \to E+T\cdot], [T \to T\cdot * F]\}$

Then $I_{14} = goto(I_6, F)$

$= \{[T \to F\cdot]\} = I_3 = I_{10}$

Then $I_{15} = goto(I_6, ()$

$= \{[F \to (\cdot E)]\} \cup closure(E)$

$= I_4 = I_{11}$

Then $I_{16} = goto(I_6, id) = \{[F \to id\cdot]\} = I_5 = I_{12}$

Then $I_{17} = goto(I_7, F)$

$= \{[T \to T * F\cdot]\}$

Then, $I_{18} = goto(I_7, ()$

$= \{[F \rightarrow (\cdot E)]\} \cup closure(E)$

$= I_4 = I_{11} = I_{15}$

Then $I_{19} = goto(I_7, id)$

$= \{[F \rightarrow id\cdot]\} = I_5 = I_{12}$

Then, $I_{20} = goto(I_8, ))$

$= \{[F \rightarrow (E)\cdot]\}$

Then, $I_{21} = goto(I_8, +)$

$= \{[E \rightarrow E+\cdot T]\} \cup closure(T)$