~~Lexical Analysis~~
Parsing

(All def^ns are from Chomsky)

Language:

A lang. is a coll. of sentences of finite length all constructed from a finite alphabet of symbols

~~But~~

Grammar:

↳ regarded as a device that enumerates the sentences of a language

2 types of grammar

- **Generative:** construct sentences (how to write a lang)
- **Analytic** helps to check if we have correct ~~syntax~~ sentence or not (how to read a lang)

2 diff structures of a language

- **Deep structure** (semantic rep^n)
- **Surface structure** (for parsing)

How do we define a grammar?

Formal grammar is a quintuple

$$G = (N, \Sigma, P, S)$$

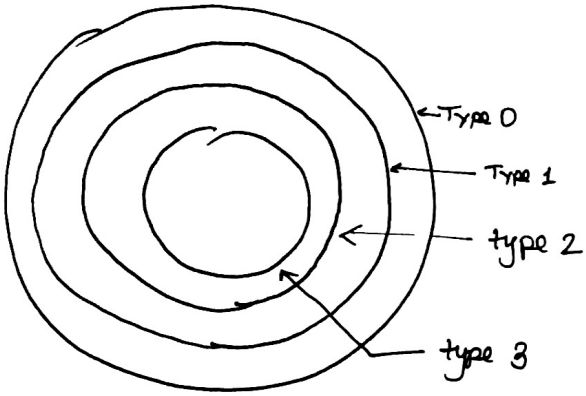
where, N = finite set of non-terminals → appear on the LHS

Σ = finite set of terminals → do not produce further symbols

P = finite set of production rules of the form $w \in (N \cup \Sigma)^* \rightarrow w \in (N \cup \Sigma)^*$ (w producing w)

$S \in N$ is the start ~~sys~~ symbol

Chomsky defined 4 types of grammar



- **Type 0 grammars** (unrestricted grammars)
  - include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine
  - infinite length string. Any symbol can follow any symbol
- **Type 1 grammars** (context sensitive grammars)
  - generates the context sensitive languages
  - These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where A is a non-terminal and α, β, γ are strings of terminals, and non-terminals
    The strings α and β may be empty, but γ must be non empty
    defn context of γ
  - ~~There~~ The rule $S \rightarrow \varepsilon$ is allowed, if S does not appear on the right side of any rule
    start symbol
  - The languages described by this grammar are exactly all languages that can be recognized by a Linear bounded automata (a turing m/c with finite length i/p)
- **Type 2 grammars** (context free grammars)
  - Generate the context free languages
  - Defn by the rules of the form $A \rightarrow \gamma$ where A is a non terminal and γ is a string of terminals and non-terminals

- These languages are exactly all languages that can be recognized by a non deterministic push down automata (essentially, a stack)

- Type 3 grammars (regular grammars)
  - Generates regular languages
  - Such a grammar restricts its rules to a single non-terminal on the LHS and a RHS consisting a single terminal, possibly followed (or preceded), but not both in the same grammar) by a single non-terminal

$A \longrightarrow aB$

$A \longrightarrow Ba$

[Convention is
- A, B, C... (caps) for non terminals
- a, b, c (or id) ... (small) for terminals
- α, β, γ (or exp) ... for strings of terminals and non-terminals]

  - The rule $S \rightarrow \varepsilon$ is also allowed here if S does not appear on the right side of any exp^n.
  - These languages are all languages that can be decided by a Finite State Automata

We study context free grammars

$$\left.\begin{array}{l} E \rightarrow EAE \\ E \rightarrow (E) \\ E \rightarrow -E \\ E \rightarrow id \end{array}\right\} E \rightarrow EAE \mid (E) \mid -E \mid id$$

(E is an expression)
A are arith. operators

in a grammar, LHS of first rule is the start symbol

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

Generating sentences

(i) checking if $\underbrace{-id+id}_{\text{sentence}}$ is valid

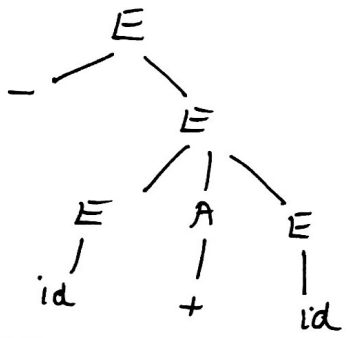
$$E \Rightarrow -E \Rightarrow -EAE \Rightarrow \underbrace{-id\,AE \Rightarrow -id+E \Rightarrow -id+id}_{\text{LTR (leftmost derivation)}}$$

($\because E \to -E$)

$$\Rightarrow \underbrace{-EAid \Rightarrow -E+id \Rightarrow -id+id}_{\text{RTL (rightmost derivation)}}$$

intermediate forms are called ~~sentential~~ sentential forms

Parse tree form (for leftmost derivation)



(ambiguity obtained here) ⇒ since no precedence was defined

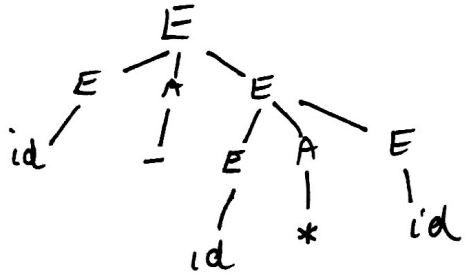
(ii) id - id * id

(rightmost)

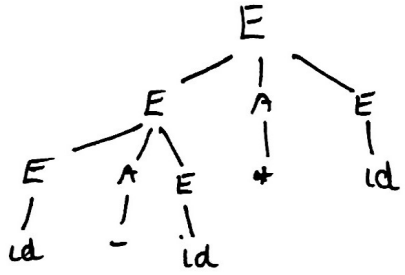
$$EAE \Rightarrow EAEAE \Rightarrow EAEA\,id \Rightarrow EAE * id$$
$$\Rightarrow EA\,id * id$$
$$\Rightarrow E - id * id$$
$$\Rightarrow id - id * id$$

$$E \Rightarrow EAE \Rightarrow EAEAE$$
$$\Rightarrow id\,AEAE \Rightarrow id - EAE \Rightarrow id - id\,AE \Rightarrow id - id * E$$
$$\Rightarrow id - id * id$$

(leftmost derivation used)



(parse tree)
(for ~~leftmost~~ rightmost derivation)



(parse tree)
for leftmost derivation

To include precedence

$E \to EAE \mid (E) \mid id$
$A \to + \mid - \mid *$

} initial rules

(adding precedence)

$E \to E \text{ addop } E \mid T$ ⇒ addop gets lower precedence ~~higher priority~~ than mul

$\text{addop} \to + \mid -$

~~mulo~~

$T \to T \text{ mulop } T \mid F$

$\text{mulop} \to *$

$F \to (E) \mid id$

Associativity rules

- Left associative

$E \to \cancel{E \text{ addop}}\ E \text{ addop } T \mid T$

$\text{addop} \to + \mid -$

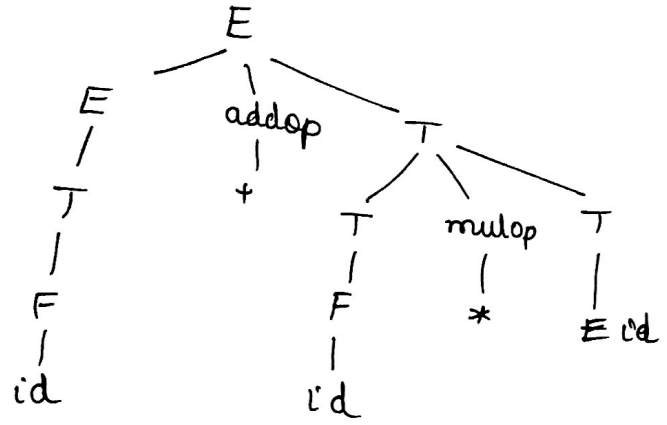
$T \to T \text{ mulop } F \mid F$

$\text{mulop} \to *$

$F \to (E) \mid id$

Again, consider id + id * id
(leftmost derivation)

$E \Rightarrow E \text{ addop } T$
$\Rightarrow T \text{ addop } T$
$\Rightarrow F \text{ addop } T$
$\Rightarrow id \text{ addop } T$
$\Rightarrow id + T$
$\Rightarrow id + T \text{ mulop } F$
$\Rightarrow id + F \text{ mulop } F$
$\Rightarrow id + id \text{ mulop } F$
$\Rightarrow id + id * F$
$\Rightarrow id + id * id$



(parse tree for LMD)

for rightmost derivation,

$E \Rightarrow E \text{ addop } T \Rightarrow E \text{ addop } T \text{ mulop } F$

$\Rightarrow E \text{ addop } T \text{ mulop id}$

$\Rightarrow E \text{ addop } T * \text{id}$

$\Rightarrow E \text{ addop } F * \text{id}$

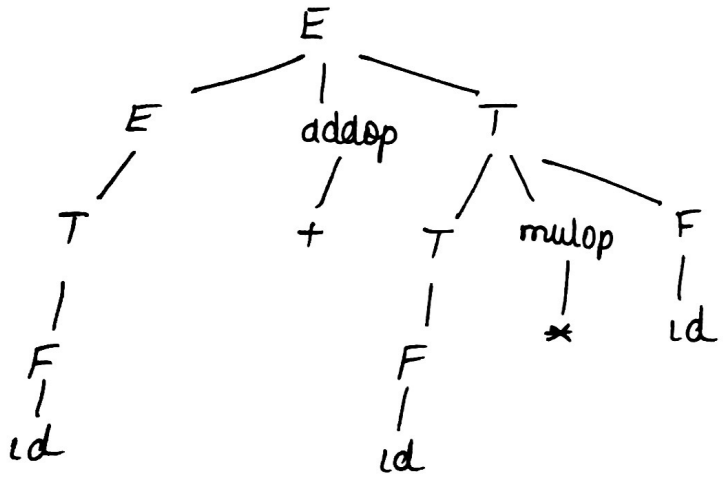
$\Rightarrow E \text{ addop id} * \text{id}$

$\vdots$

$\Rightarrow \cancel{id}\ E\ \cancel{addo} + \text{id} * \text{id}$

$\Rightarrow T + \text{id} * \text{id}$

$\Rightarrow F + \text{id} * \text{id} \Rightarrow \text{id} + \text{id} * \text{id}$



(parse tree for RMD)

<u>It is the same</u>

Another example of ambiguity (Dangling else problem)

statement → if-stmt | other

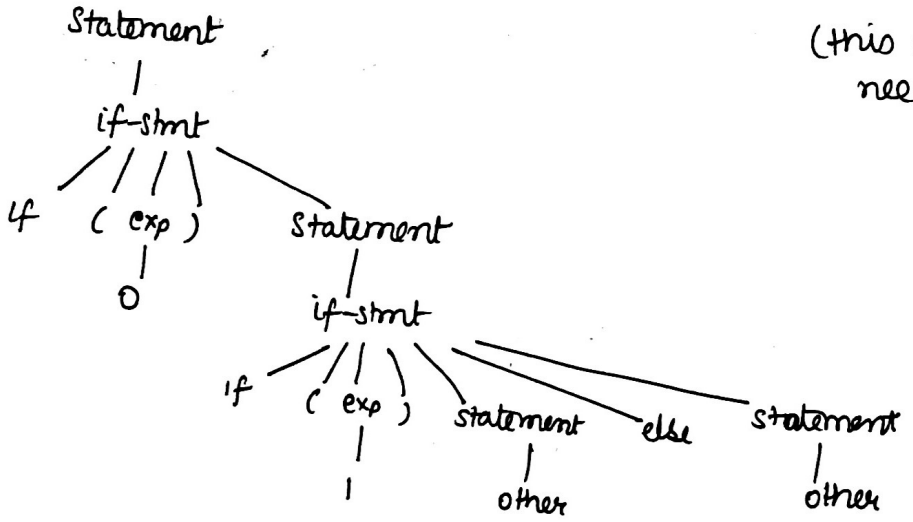
if-stmt → if (exp) statement | if (exp) statement else statement

exp → 0 | 1

if (0) if (1) other $\underbrace{\text{else other}}$

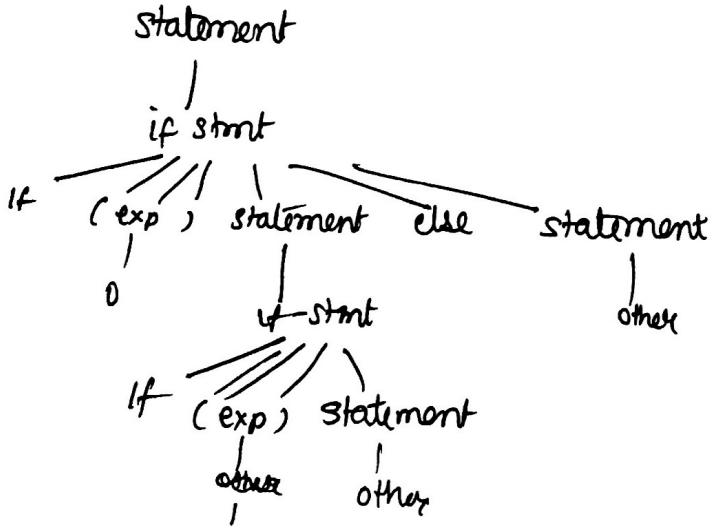
which if to go with? diff if obtained for LMD & RMD

LMD



(this is what we need in C)

RMD



Improved grammar (to avoid dangling if)

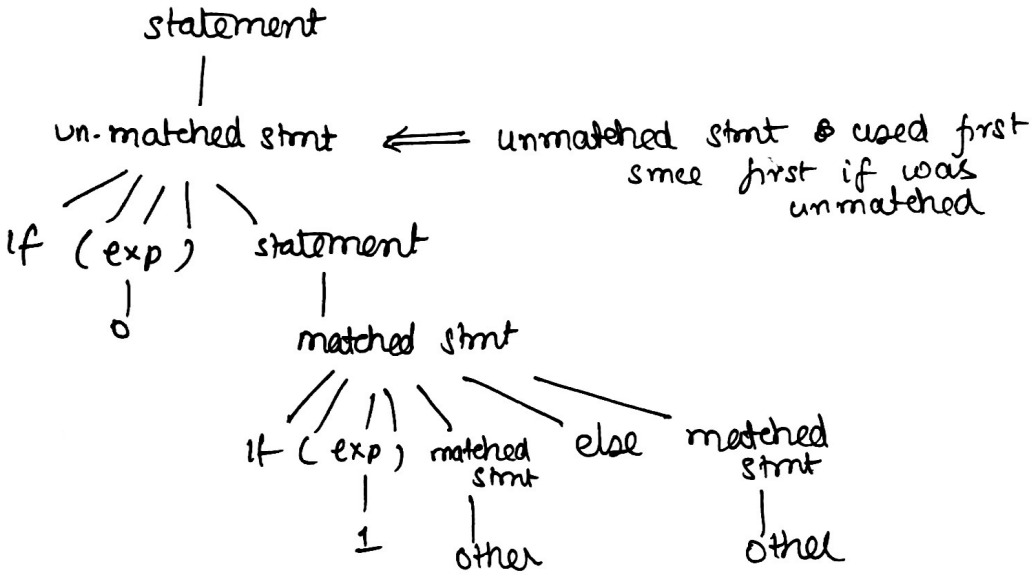
statement → matched-stmt | unmatched-stmt

matched-stmt → if (exp) matched-stmt else matched-stmt | other

unmatched-stmt → if (exp) statement | if (exp) matched-stmt
else unmatched stmt

exp → 0/1

if (0) if (1) other else other



4/2/15

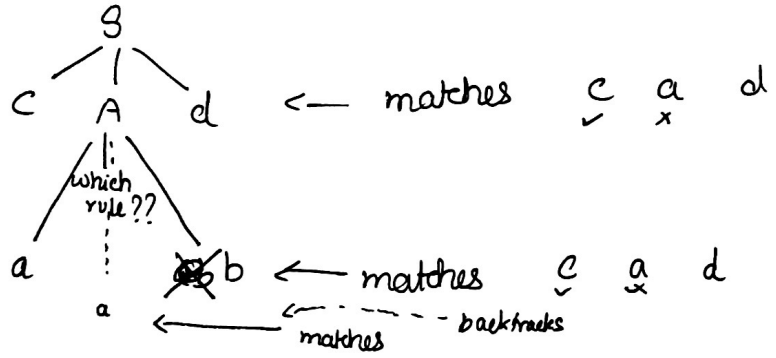
Parsing

- Parsing
  - Top-down
    - Backtracking
    - ~~Recursive Descent~~ Predictive parsing (relies on look ahead tokens) → Recursive descent parsing
      - → LL(1) (mostly used) (also called non recursive predictive parsing)
  - Bottom-UP (covers a large no of grammars)

$S \rightarrow \cancel{cAD}\ cAd$

$A \rightarrow ab \mid a$

Given string : cad



LL(1)
- → 1 token look ahead
- → left to right scan
- → leftmost derivation

- Works for a restrictive set of grammars
- ~~First step~~
- Algorithm ~~to prepare a correct grammar~~:
  1. Eliminate all ambiguity
  2. Eliminate Left Recursion
  3. Left factoring

  } prepares a usable grammar
  4. Constr. of FIRST and FOLLOW

Left recursion:

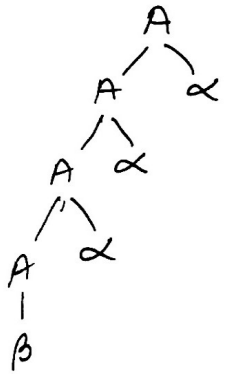
$A \rightarrow A\alpha \mid \beta$

(this is left recursion: first char on RHS = LHS)

we do not know which rule to apply in LL(1) because we don't know how many α follow β

($\beta\alpha\alpha\alpha\alpha$)

Left recursion

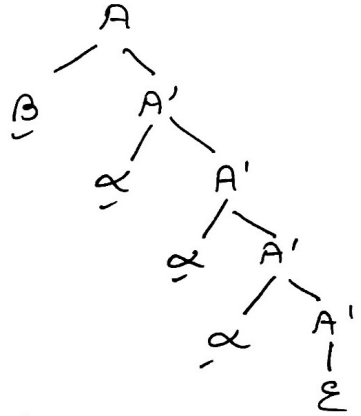


Converting Immediate Left recursion rules to right rec. $\Longrightarrow$

Right recursion

$A \rightarrow \beta A'$

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



This works fine in LL(1)

Non immediate left recursion

$S \rightarrow A\alpha \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

$S \Rightarrow A\alpha \Rightarrow Sd\alpha$

Removing left recursion

for $i = 1$ to n do

  for $j = 1$ to $i-1$ do

    replace each grammar rule choice of the form $A_i \rightarrow A_j\beta$ by the rule

    $A_i \rightarrow \alpha_1\beta \mid \alpha_2\beta \mid \alpha_3\beta \mid \ldots \mid \alpha_k\beta$, ~~where~~

    where, $A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ is the current rule for A_j

Eliminate all immediate left recursion of A_i

~~A →~~

Eg of removal of Left recursion

immediate

① $A \rightarrow B\alpha \mid A\alpha \mid c$

② $B \rightarrow Bb \mid Ab \mid d$

immediate

non immediate

iteration

① outer loop : $i=1$ ($A_i = A$ ~~A →~~ ...)

inner loop ~~i=0~~ : exits

~~A → B~~

removal of imm. Left recursion

$$\left.\begin{aligned} A &\rightarrow B\alpha A' \mid CA' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}\right\} \text{new rules ①}$$

[2 new rules are formed but the numbering of rules will not change)

~~removal of~~

② outer loop : $i=2$ ($A_i = B$ ~~B~~ ...)

inner loop : $j=1$ ($A_j = A$)

$A_i \rightarrow A_j B$ ($B \rightarrow Ab$)

Now, $A \rightarrow B\alpha A' \mid CA'$

$B \rightarrow Bb \mid \underbrace{B\alpha A'}b \mid \underbrace{CA'}b \mid d$

replaced

exits inner loop

2 immediate left recursions

$$\left.\begin{aligned} A &\rightarrow B\alpha A' \mid CA' \\ A' &\rightarrow \alpha A' \mid \varepsilon \\ B &\rightarrow CA'bB' \mid dB' \\ B' &\rightarrow bB' \mid \alpha A'bB' \mid \varepsilon \end{aligned}\right\} \text{final set of rules}$$

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

Left factoring

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

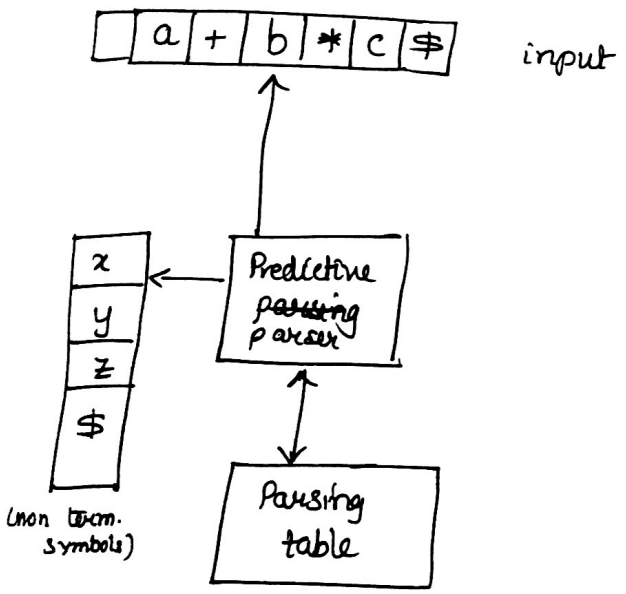
Since we only have one-token look-up, there is ambiguity

∴ We change the rule to

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$ ← this decision is effectively deferred

Working of the parser



We construct 2 sets

FIRST set (of α, say) which may appear
↳ ~~Each ter~~ All terminal symbols at beginning of α

FOLLOW set (of A, say)
↳ All terminal symbols which can follow non-terminal A

Construction of sets

① FIRST (α)

(i) if X is a terminal, FIRST(X) = X

(ii) if $X \rightarrow \varepsilon$ is a production, Add ε to FIRST(X)

(iii) If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \ldots Y_n$ is a production, then place a in FIRST(X), ~~where~~ if for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1) .... FIRST(Y_{i-1})

Example:

$E \rightarrow E+T \mid T$ ⇒ contains left recursion

$T \rightarrow T*F \mid F$ ⇒ also " " "

$F \rightarrow (E) \mid id$

$\xRightarrow{\text{after eliminating left recursion}}$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Now,

FIRST(+) = {+}

FIRST (*) = {*}

~~FIRST (e)~~ =

FIRST (() = {(}

FIRST ()) = {)}

FIRST (id) = {id}

non-terminals

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

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

FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

② FOLLOW (A): set of terminals a that can appear immediately to the right of A in some sentencial form

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

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

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

FOLLOW (E) = {$} ~~∪ {FOLLOW~~ ,)}

FOLLOW (T) = ~~{~~ FIRST (E') ∪ FOLLOW (E) = {+,), $}

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

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

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

Compiler Design

10-2-15

① For each production $A \to \alpha$ of the grammar, do the foll. steps –

ⓐ For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, \alpha a]$

ⓑ If ε is in ~~first of alpha~~ FIRST(α), add $A \to \alpha$ to $M[A, \alpha b]$ for each terminal b in FOLLOW(A)
If ε is in FIRST(α) and \$ is in FOLLOW(A) add $A \to \alpha$ to $M[A, \$]$

② Mark each undefined entry of M as "error"

Table: ($M[A, b]$)

terminals →

non terminals ↓

	id	+	*	(	)	\$
E	E→TE'	scan	scan	E→TE'	pop	pop
E'	scan	E'→+TE'	scan	scan	E'→ε	E'→ε
T	T→FT'	pop	scan	T→FT'	~~T→FT'~~ pop	pop
T'	scan	T'→ε	T'→*FT'	scan	T'→ε	T'→ε
F	F→id	pop	pop	F→(E)	pop	pop

~~Ero~~ Erroneous entries (blank entries) are in blue
~~Fu~~ Go ahead a few pages for ~~the~~ details about pop and scan actions

Stack

Stack	Input ~~Symbol~~	Output	Comments
$E	id + id * id $ (↑ current input marker)	~~E → TE'~~ E → TE'	from M[E, id], we get E → TE'. E popped. TE' pushed (right-most in stack)
$E'T	id + id * id $	T → FT'	
$E'T'F	id + id * id $	~~F → id~~ F → id	
$E'T'id	id + id * id $	matched	id matched. Input marker progresses
$E'T'	+ id * id $	T' → ε	
$E'	+ id * id $	E' → +TE'	
$E'T+	+ id * id $	matched	+ matched
$E'T	id * id $		
⋮			
$	$	matched	

The derivation is as follows:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow id\,T'E' \Rightarrow id\,E' \ldots$$

This gives the leftmost derivation

This parsing technique does not work for some rules

- $A \rightarrow \alpha \mid \beta$
  FIRST(α) & FIRST(β) contain same terminal symbol
- if both α & β produce an empty string
- either α or β derives an empty string, then α derives any string beginning with a terminal in FOLLOW(A)
  ($\beta \overset{*}{\Rightarrow} \varepsilon$)

* In these cases, the grammar is not in LL(1)

Def^n of an LL(1) grammar

A grammar is an LL(1) grammar if all productions conform to the foll. LL(1) conditions:

1) For each production $A \rightarrow \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n$,

~~FIRST(σ_1) ∩ FIRST(σ_2) ... ∩ FIRST(σ_n)~~

$$FIRST(\sigma_i) \cap FIRST(\sigma_j) = \Phi \quad \forall\ i,j,\ i \neq j$$

(FIRST should never contain the same terminal)

2) If non-terminal X can derive an empty string, then $FIRST(X) \cap FOLLOW(X) = \Phi$

Consider the grammar

$Z \rightarrow d$

$Z \rightarrow XYZ$

$Y \rightarrow \varepsilon$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

Construct the parsing table

Note your observations

Error-entry (blank cell) in parsing table

Say id ++ id $ ~~as~~ is an input

This causes an error. The compiler behaves as follows:

① (As soon as possible), an error is detected ~~and~~ as soon as possible

② compiler picks a likely place from where it can resume parsing again ~~and while doing this~~ (As much as possible)

③ Avoid error cascading

④ Avoid infinite loops on error

There are 2 rules:

uses synchronizing tokens for this

① Given a non-terminal A on top of the stack and an input token that is not in FIRST(A) (or FOLLOW(A) if ε is in FIRST(A)) then there are the following 3 alternatives

ⓐ Pop A from the stack, if current input token is $ or is in FOLLOW(A) – [Pop action]

ⓑ Successively pop tokens from the input until a token is seen for which we can restart the parse – if the current input token is not $ and is not in FIRST(A) ∪ FOLLOW(A)

[Scan action]

ⓒ Push a new non-terminal symbol onto the stack

Class Test-1

25th – 10:20 – 11:10 (A1)

11:10 – 12 (A2)

Syllabus – till here