Problems with Top Down Parsing

- Left Recursion in CFG May Cause Parser to Loop Forever.
- Indeed:
  - In the production $A \rightarrow A\alpha$ we write the program procedure A

    {

    if lookahead belongs to First($A\alpha$) then

    call the procedure A

    }

- Solution: Remove Left Recursion...
  - without changing the Language defined by the Grammar.

Dealing with Left recursion

- Solution: Algorithm to Remove Left Recursion:

BASIC IDEA:

$A \rightarrow A\alpha | \beta$ becomes

$A \rightarrow \beta R$

$R \rightarrow \alpha R | \epsilon$

expr* → *expr* + *term* | *expr* - *term* | *term

***term* → id**

expr* → *term rest

***rest* → + *term rest* | - *term rest* | ϵ**

***term* → id**

Resolving Difficulties : Left Recursion

A left recursive grammar has rules that support the derivation : $A \overset{+}{\Rightarrow} A\alpha$, for some α.

Top-Down parsing can't reconcile this type of grammar, since it could consistently make choice which wouldn't allow termination.

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \ \dots \text{ etc.} \quad A \rightarrow A\alpha \mid \beta$$

Take left recursive grammar:

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

To the following:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Resolving Difficulties : Left Recursion (2)

Informal Discussion:

Take all productions for <u>A</u> and order as:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Where no β_i begins with A.

Now apply concepts of previous slide:

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$$

For our example:

$E \rightarrow E + T \mid T$ $\longrightarrow$ $\begin{cases} E \rightarrow TE' \\ E' \rightarrow + TE' \mid \epsilon \end{cases}$

$T \rightarrow T * F \mid F$ $\longrightarrow$ $\begin{cases} T \rightarrow FT' \\ T' \rightarrow * FT' \mid \epsilon \end{cases}$

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

Resolving Difficulties : Left Recursion (3)

Problem: If left recursion is two-or-more levels deep, this isn't enough

$$\left.\begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \in \end{array}\right\}. \qquad S \Rightarrow Aa \Rightarrow Sda$$

Algorithm:

Input: **Grammar G with ordered Non-Terminals A_1, ..., A_n**

Output: **An equivalent grammar with no left recursion**

1. **Arrange the non-terminals in some order A_1=start NT,A_2,…A_n**
2. **for *i* := 1 to *n* do begin**

   for *j* := 1 to *i* – 1 do begin

   replace each production of the form $A_i \rightarrow A_j\gamma$

   by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

   where $A_j \rightarrow \delta_1|\delta_2|\ldots|\delta_k$ are all current A_j productions;

   end

   eliminate the immediate left recursion among A_i productions

   end

Using the Algorithm

Apply the algorithm to: $A_1 \rightarrow A_2a \mid b \mid \epsilon$

$A_2 \rightarrow A_2c \mid A_1d$

i = 1

For A_1 there is no left recursion

i = 2

for j=1 to 1 do

Take productions: $A_2 \rightarrow A_1\gamma$ and replace with

$A_2 \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma \mid$

where $A_1 \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ **are A_1 productions**

in our case $A_2 \rightarrow A_1d$ becomes $A_2 \rightarrow A_2ad \mid bd \mid d$

What's left: $A_1 \rightarrow A_2a \mid b \mid \epsilon$

$A_2 \rightarrow A_2c \mid A_2ad \mid bd \mid d$

Are we done ?

Using the Algorithm (2)

No ! We must still remove A_2 left recursion !

$$A_1 \rightarrow A_2 a \mid b \mid \in$$

$$A_2 \rightarrow A_2 c \mid A_2 ad \mid bd \mid d$$

Recall:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in$$

Apply to above case. What do you get ?

Removing Difficulties : Left Factoring

Problem : Uncertain which of 2 rules to choose:

$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$
$$\mid \textbf{if}\ expr\ \textbf{then}\ stmt$$

When do you know which one is valid ?

What's the general form of *stmt* ?

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

α : **if** *expr* **then** *stmt*

β_1: **else** *stmt* β_2 : $\in$

Transform to:

$A \rightarrow \alpha A'$

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

EXAMPLE:

$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ rest$

$rest \rightarrow \textbf{else}\ stmt \mid \in$

Motivating Table-Driven Parsing

1. **Left to right scan input**

2. **Find leftmost derivation**

Terminator

Grammar: $E \rightarrow TE'$

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

$T \rightarrow id$

Input : $id + id\ \$$

Derivation: $E \Rightarrow$

Processing Stack:

LL(1) Grammars

L : Scan input from Left to Right

L : Construct a Leftmost Derivation

1 : Use "1" input symbol as lookahead in conjunction with stack to decide on the parsing action

LL(1) grammars == they have no multiply-defined entries in the parsing table.

Properties of LL(1) grammars:

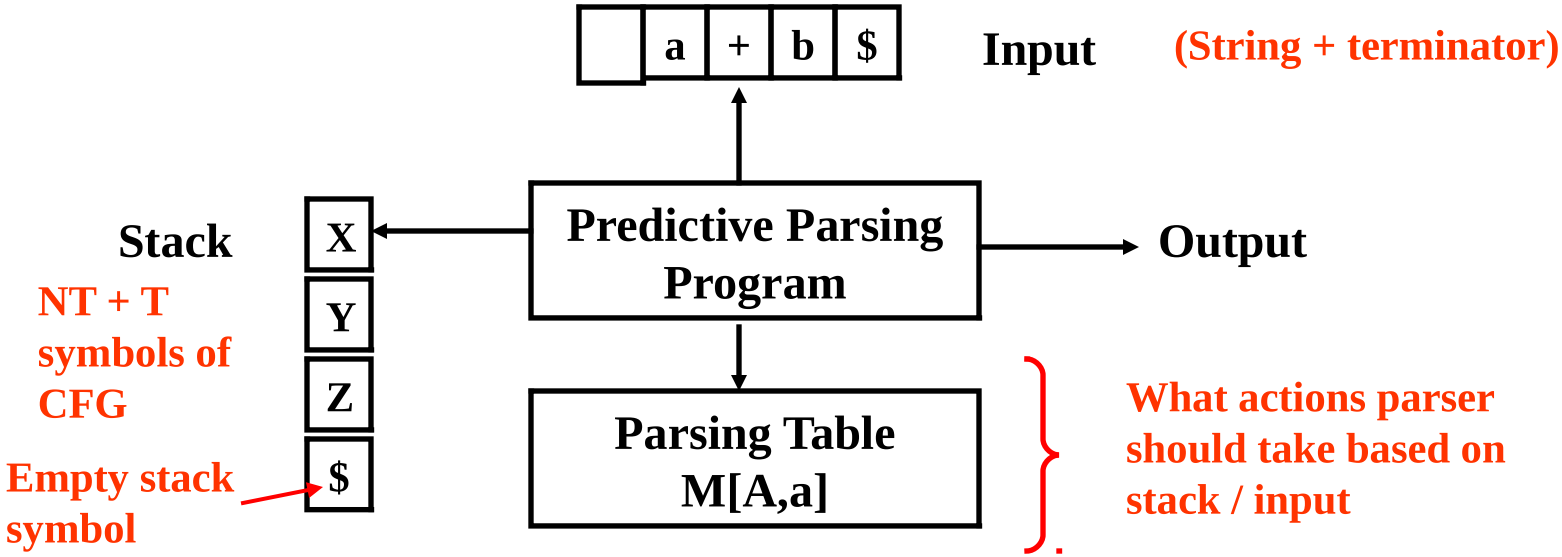
- **Grammar can't be ambiguous or left recursive**
- **Grammar is LL(1) $\Leftrightarrow$ when $A \rightarrow \alpha \mid \beta$**

  1. α & β do not derive strings starting with the same terminal a

  2. Either α or β can derive $\in$, but not both.

Note: It may not be possible for a grammar to be manipulated into an LL(1) grammar

Non-Recursive / Table Driven



General parser behavior: X : top of stack   a : current input

1. When X=a = $ halt, accept, success
2. When X=a ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X,a]
   if it is an error → call recovery routine
   if M[X,a] = {X → UVW}, POP X, PUSH W,V,U
   DO NOT expend any input

Algorithm for Non-Recursive Parsing

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*;

Input pointer

if X is terminal or $ **then**

if X=a **then**

pop X from the stack and advance *ip*

else *error()*

else /* X is a non-terminal */

if $M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ **then begin**

pop X from stack;

push $Y_k, Y_{k-1}, \ldots, Y_1$ onto stack, with Y_1 on top

output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$

May also execute other code based on the production used

end

else *error()*

until X=$ /* stack is empty */

Example

E → TE'

E' → + TE' | ∈

T → FT'

T' → * FT' | ∈

F → (E) | id

Our well-worn example !

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'			T→FT'		
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id			F→(E)		

Trace of Example

STACK	INPUT	OUTPUT

Trace of Example

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E’T	id + id * id$	$E \rightarrow TE'$
$E’T’F	id + id * id$	$T \rightarrow FT'$
$E’T’id	id + id * id$	$F \rightarrow id$
$E’T’	+ id * id$	
$E’	+ id * id$	$T' \rightarrow \epsilon$
$E’T+	+ id * id$	$E' \rightarrow \underline{+}TE'$
$E’T	id * id$	
$E’T’F	id * id$	$T \rightarrow FT'$
$E’T’id	id * id$	$F \rightarrow id$
$E’T’	* id$	
$E’T’F*	* id$	$T' \rightarrow \underline{*}FT'$
$E’T’F	id$	
$E’T’id	id$	$F \rightarrow id$
$E’T’	$	
$E’	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Expend Input

Leftmost Derivation for the Example

The leftmost derivation for the example is as follows:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \mathbf{id}\ T'E' \Rightarrow \mathbf{id}\ E' \Rightarrow \mathbf{id} + TE' \Rightarrow \mathbf{id} + FT'E'$$

$$\Rightarrow \mathbf{id} + \mathbf{id}\ T'E' \Rightarrow \mathbf{id} + \mathbf{id} * FT'E' \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}\ T'E'$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}\ E' \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

What's the Missing Puzzle Piece ?

Constructing the Parsing Table M !

1st : Calculate First & Follow for Grammar

2nd: Apply Construction Algorithm for Parsing Table
(We'll see this shortly)

Basic Tools:

First: **Let α be a string of grammar symbols. First(α) is the set that includes every terminal that appears leftmost in α or in any string originating from α.**
NOTE: If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is First(α).

Follow: **Let A be a non-terminal. Follow(A) is the set of terminals a that can appear directly to the right of A in some sentential form. (S $\overset{*}{\Rightarrow}$ αAaβ, for some α and β).**
NOTE: If S $\overset{*}{\Rightarrow}$ αA, then \$ is Follow(A).

Constructing Parsing Table

Algorithm:

Table has one row per non-terminal / one column per terminal (incl. $)

1. **Repeat Steps 2 & 3 for each rule $A \rightarrow \alpha$**
2. **Terminal a in First(α)? Add $A \rightarrow \alpha$ to M[A, a]**
3. **$\in$ in First(α)? Add $A \rightarrow \alpha$ to M[A, b] for all terminals b in Follow(A).**
4. **All undefined entries are errors.**

Constructing Parsing Table – Example 1

$S \rightarrow$ i E t SS' | a	First(S) = { i, a }	Follow(S) = { e, $ }
$S' \rightarrow$ eS | $\in$	First(S') = { e, $\in$ }	Follow(S') = { e, $ }
$E \rightarrow$ b	First(E) = { b }	Follow(E) = { t }

Constructing Parsing Table – Example 1

$S \to iEtSS' \mid a$	First(S) = { i, a }	Follow(S) = { e, $ }
$S' \to eS \mid \epsilon$	First(S') = { e, ϵ }	Follow(S') = { e, $ }
$E \to b$	First(E) = { b }	Follow(E) = { t }

$S \to iEtSS'$ — First(i E t SS')={i}

$S \to a$ — First(a) = {a}

$E \to b$ — First(b) = {b}

$S' \to eS$ — First(eS) = {e}

$S' \to \epsilon$ — First(ϵ) = {ϵ}

Follow(S') = { e, $ }

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S \to \epsilon$
E		$E \to b$				

Constructing Parsing Table – Example 2

E → TE’

E’ → + TE’ | ∈

T → FT’

T’ → * FT’ | ∈

F → (E) | id

First(E,F,T) = { (, id }	**Follow(E,E’) = {), $}**
First(E’) = { +, ∈ }	**Follow(F) = { *, +,), $ }**
First(T’) = { *, ∈ }	**Follow(T,T’) = { +,) , $}**

Constructing Parsing Table – Example 2

$E \to TE'$ $E' \to +TE' \mid \epsilon$ $T \to FT'$ $T' \to *FT' \mid \epsilon$ $F \to (E) \mid id$	First(E,F,T) = { (, id } First(E') = { +, ϵ } First(T') = { *, ϵ }	Follow(E,E') = {), \$} Follow(F) = { *, +,), \$ } Follow(T,T') = { +,) , \$}

Expression Example: $E \to TE'$: First(TE') = First(T) = { (, id }

M[E, ( ] : $E \to TE'$

M[E, id] : $E \to TE'$

by rule 2

(by rule 2) $E' \to +TE'$: First(+TE') = + : M[E', +] : $E' \to +TE'$

(by rule 3) $E' \to \epsilon$: ϵ in First(ϵ)	$T' \to \epsilon$: ϵ in First(ϵ)
M[E',)] : $E' \to \epsilon$ (3)	M[T', +] : $T' \to \epsilon$ (3)
M[E', \$] : $E' \to \epsilon$ (3)	M[T',)] : $T' \to \epsilon$ (3)
(Due to Follow(E'))	M[T', \$] : $T' \to \epsilon$ (3)

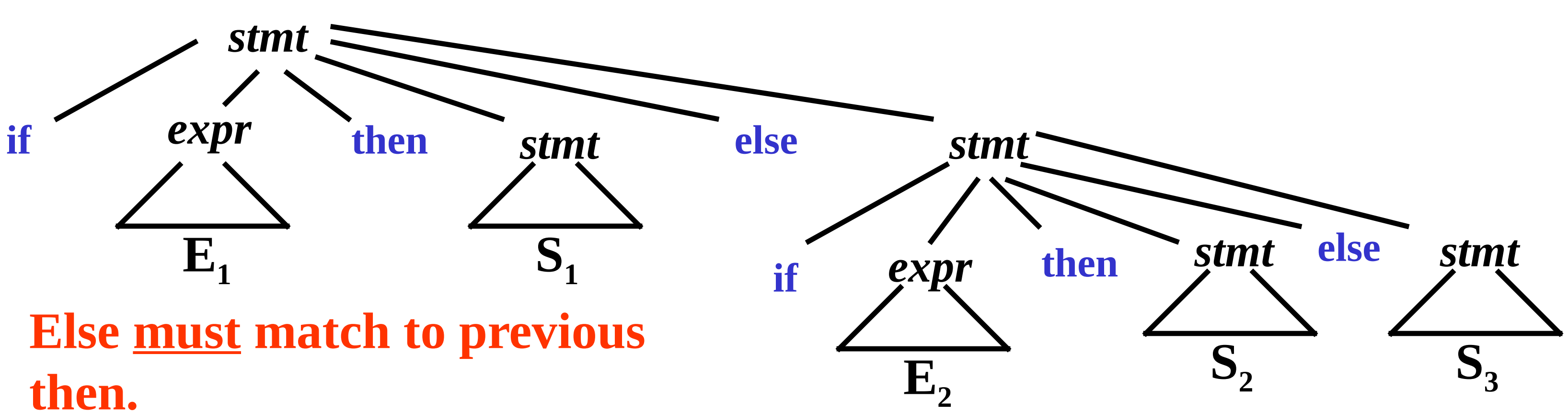
Resolving Problems: Ambiguous Grammars

Consider the following grammar segment:

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt$$
$$\mid \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$\mid \textbf{other} \text{ (any other statement)}$$

What's problem here ?

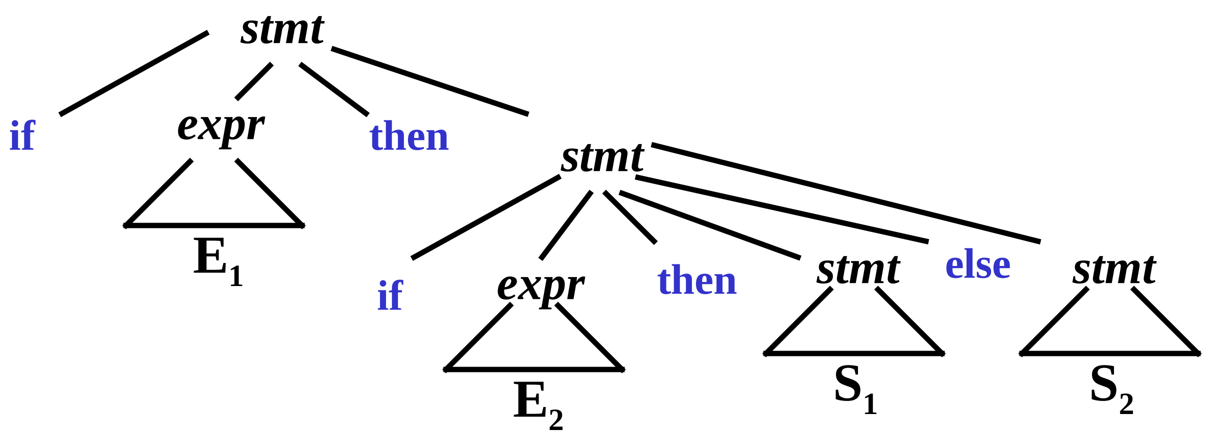
Let's consider a simple parse tree:



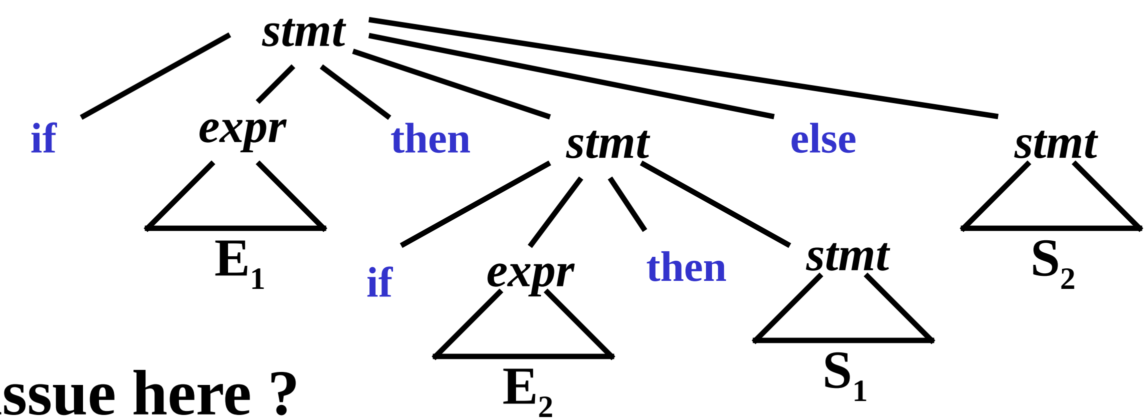
Else <u>must</u> match to previous then.

Parse Trees for Example

Form 1:



Form 2:



What's the issue here ?

Removing Ambiguity

Take Original Grammar:

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ |\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ |\ & \textbf{other}\ \ \textbf{(any other statement)} \end{aligned}$$

Or to write more simply:

$$\begin{aligned} S \rightarrow\ & \mathbf{i}\ E\ \mathbf{t}\ S \\ |\ & \mathbf{i}\ E\ \mathbf{t}\ S\ \mathbf{e}\ S \\ |\ & \mathbf{s} \\ E \rightarrow\ & \mathbf{a} \end{aligned}$$

The problem string: i a t i a t s e s

Revise to remove ambiguity:

$$
\begin{aligned}
S &\rightarrow \mathrm{i}\ E\ \mathrm{t}\ S \\
&\quad \mid \ \mathrm{i}\ E\ \mathrm{t}\ S\ \mathrm{e}\ S \\
&\quad \mid \ \mathrm{s} \\
E &\rightarrow \mathrm{a}
\end{aligned}
$$

$$
\begin{aligned}
S &\rightarrow M \mid U \\
M &\rightarrow \mathrm{i}\ E\ \mathrm{t}\ M\ \mathrm{e}\ M \mid \mathrm{s} \\
U &\rightarrow \mathrm{i}\ E\ \mathrm{t}\ S \mid \mathrm{i}\ E\ \mathrm{t}\ M\ \mathrm{e}\ U \\
E &\rightarrow \mathrm{a}
\end{aligned}
$$

Try the above on **i a t i a t s e s**

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Error Processing

Syntax Error Identification / Handling

Recall typical error types:

- **Lexical : Misspellings**
- **Syntactic : Omission, wrong order of tokens**
- **Semantic : Incompatible types**
- **Logical : Infinite loop / recursive call**

Majority of error processing occurs during syntax analysis

NOTE: Not all errors are identifiable !! Which ones?

Error Processing

- Detecting errors
- Finding position at which they occur
- Clear / accurate presentation
- Recover (pass over) to continue and find later errors
- Don’t impact compilation of “correct” programs

Error Recovery Strategies

Panic Mode– Discard tokens until a "synchronizing" token is found (end, ";", "}", etc.)

-- Decision of designer

-- Problems:

skip input $\Rightarrow$ miss declaration – causing more errors

$\Rightarrow$ miss errors in skipped material

-- Advantages:

simple $\Rightarrow$ suited to 1 error per statement

Phrase Level – Local correction on input

-- "," $\Rightarrow$ ";" – Delete "," – insert ";"

-- Also decision of designer

-- Not suited to all situations

-- Used in conjunction with panic mode to allow less input to be skipped

Error Recovery Strategies – (2)

Error Productions:

-- Augment grammar with rules

-- Augment grammar used for parser construction / generation

-- example: add a rule for
:= in C assignment statements
Report error but continue compile

-- Self correction + diagnostic messages

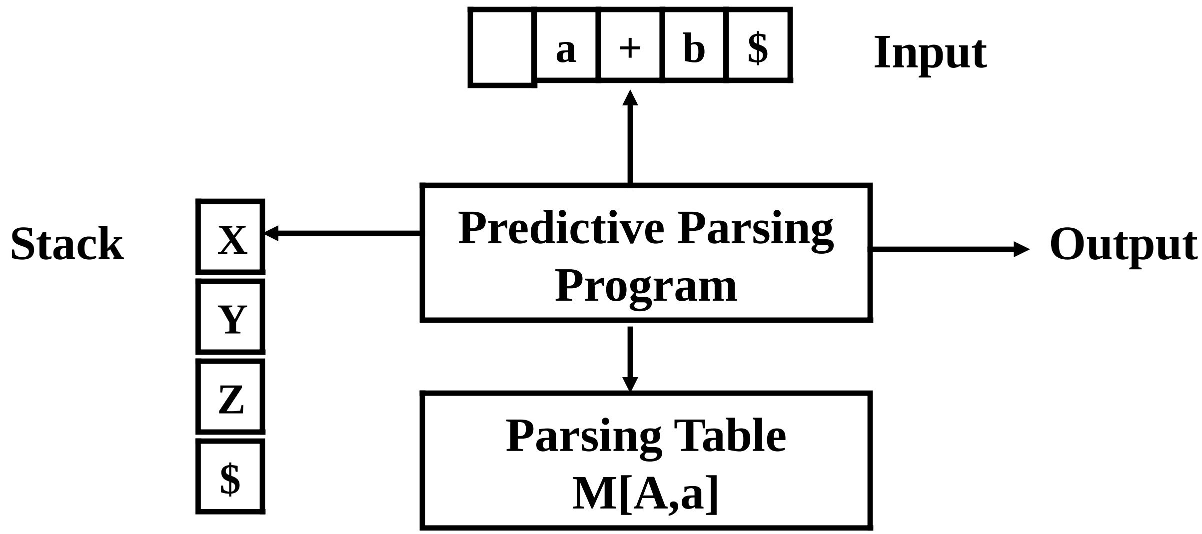
Global Correction:

-- Adding / deleting / replacing symbols is chancy – may <u>do many</u> changes !

-- Algorithms available to minimize changes costly - key issues

Error Recovery

When Do Errors Occur? Recall Predictive Parser Function:



1. **If X is a terminal and it doesn't match input.**
2. **If M[X, Input] is empty – No allowable actions**

Consider two recovery techniques:

A. **Panic Mode**

B. **Phrase-level Recovery**

Panic-Mode Recovery

- Assume a non-terminal on the top of the stack.
- Idea: skip symbols on the input until a token in a selected set of *synchronizing* tokens is found.
- The choice for a synchronizing set is important.
  - some ideas:
  - define the synchronizing set of A to be FOLLOW(A). then skip input until a token in FOLLOW(A) appears and then pop A from the stack. Resume parsing...
  - add symbols of FIRST(A) into synchronizing set. In this case we skip input and once we find a token in FIRST(A) we resume parsing from A.
  - Productions that lead to ϵ if available might be used.
- If a terminal appears on top of the stack and does not match to the input == pop it and and continue parsing (issuing an error message saying that the terminal was inserted).

Panic Mode Recovery, II

General Approach: Modify the empty cells of the Parsing Table.

1. **if M[A,a] = {empty} and a belongs to Follow(A) then we set M[A,a] = "synch"**

Error-recovery Strategy :

If A=top-of-the-stack and a=current-input,

1. **If A is NT and M[A,a] = {empty} then skip a from the input.**
2. **If A is NT and M[A,a] = {synch} then pop A.**
3. **If A is a terminal and A!=a then pop token (essentially inserting it).**

Revised Parsing Table / Example

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→∈	E’→∈
T	T→FT’			T→FT’		
T’		T’→∈	T’→*FT’		T’→∈	T’→∈
F	F→id			F→(E)		

From Follow sets. Pop top of stack NT

“synch” action

Skip input symbol

Revised Parsing Table / Example(2)

STACK	INPUT	Remark
$E	+ id * + id$	error, skip +
$E	id * + id$	
$E'T	id * + id$	
$E'T'F	id * + id$	
$E'T'id	id * + id$	
$E'T'	* + id$	
$E'T'F*	* + id$	
$E'T'F	+ id$	error, M[F,+] = synch
$E'T'	+ id$	F has been popped
$E'	+ id$	
$E'T+	+ id$	
$E'T	id$	
$E'T'F	id$	
$E'T'id	id$	
$E'T'	$	
$E'	$	
$	$	

Possible
Error Msg:
"Misplaced +
I am skipping it"

Possible
Error Msg:
"Missing Term"

Writing Error Messages

- Keep input counter(s)
- Recall: every non-terminal symbolizes an abstract language construct.
- Examples of Error-messages for our usual grammar
  - E = means expression.
    - top-of-stack is E, input is +
      "Error at location i, expressions cannot start with a '+'" or
      "error at location i, invalid expression"
    - Similarly for E, *
  - E'= expression ending.
    - Top-of-stack is E', input is * or id
      "Error: expression starting at j is badly formed at location i"
    - Requires: every time you pop an 'E' remember the location

Writing Error-Messages, II

- Messages for Synch Errors.
  - Top-of-stack is F input is +
    - “error at location i, expected summation/multiplication term missing”
  - Top-of-stack is E input is)
    - “error at location i, expected expression missing”

Writing Error Messages, III

- When the top-of-the stack is a terminal that does not match…
  - E.g. top-of-stack is id and the input is +
    - “error at location i: identifier expected”
  - Top-of-stack is) and the input is terminal other than)
    - Every time you match an ‘(‘ push the location of ‘(‘ to a “left parenthesis” stack.
      - – this can also be done with the symbol stack.
    - When the mismatch is discovered look at the left parenthesis stack to recover the location of the parenthesis.
    - “error at location i: left parenthesis at location m has no closing right parenthesis”
      - – E.g. consider (id * + (id id) $

Incorporating Error-Messages to the Table

- Empty parsing table entries can now fill with the appropriate error-reporting techniques.

Phrase-Level Recovery

- **Fill in blanks entries of parsing table with error handling routines that do not only report errors but may also:**
  - **change/ insert / delete / symbols into the stack and / or input stream**
  - **+ issue error message**
- **Problems:**
  - **Modifying stack has to be done with care, so as to not create possibility of derivations that aren't in language**
  - **infinite loops must be avoided**
- **Essentially extends panic mode to have more complete error handling**

How Would You Implement TD Parser

- **Stack – Easy to handle. Write ADT to manipulate its contents**
- **Input Stream – Responsibility of lexical analyzer**
- **Key Issue – How is parsing table implemented ?**

One approach: Assign unique IDS

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id	synch	synch	F→(E)	synch	synch

All rules have unique IDs

Ditto for synch actions

Also for blanks which handle errors

Revised Parsing Table:

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	1	18	19	1	9	10
E’	20	2	21	22	3	3
T	4	11	23	4	12	13
T’	24	6	5	25	6	6
F	8	14	15	7	16	17

1 E→TE’
2 E’→+TE’
3 E’→∈
4 T→FT’
5 T’→*FT’
6 T’→∈
7 F→(E)
8 F→id

9 – 17 : Sync Actions

18 – 25 : Error Handlers

Resolving Grammar Problems

Note: Not all aspects of a programming language can be represented by context free grammars / languages.

<u>Examples:</u>

1. Declaring ID before its use

2. Valid typing within expressions

3. Parameters in definition vs. in call

These features are called context-sensitive and define yet another language class, CSL.

Context-Sensitive Languages - Examples

Examples:

$L_1 = \{ wcw \mid w \text{ is in } (a \mid b)^* \}$: Declare before use

$L_2 = \{ a^n b^m c^n d^m \mid n \geq 1, m \geq 1 \}$

$a^n b^m$: formal parameter

$c^n d^m$: actual parameter

How do you show a Language is a CFL?

$L_3 = \{ w\, c\, w^R \mid w \text{ is in } (a \mid b)^* \}$

$L_4 = \{ a^n b^m c^m d^n \mid n \geq 1,\ m \geq 1 \}$

$L_5 = \{ a^n b^n c^m d^m \mid n \geq 1,\ m \geq 1 \}$

$L_6 = \{ a^n b^n \mid n \geq 1 \}$

Solutions

$L_3 = \{ w\,c\,w^R \mid w \text{ is in } (a \mid b)^* \}$

$S \rightarrow a\,S\,a \mid b\,S\,b \mid c$

$L_4 = \{ a^n b^m c^m d^n \mid n \geq 1,\ m \geq 1 \}$

$S \rightarrow a\,S\,d \mid a\,A\,d$

$A \rightarrow b\,A\,c \mid bc$

$L_5 = \{ a^n b^n c^m d^m \mid n \geq 1,\ m \geq 1 \}$

$S \rightarrow XY$

$X \rightarrow a\,X\,b \mid ab$

$Y \rightarrow c\,Y\,d \mid cd$

$L_6 = \{ a^n b^n \mid n \geq 1 \}$

$S \rightarrow a\,S\,b \mid ab$