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 → Aα | β becomes
A → βR
R → αR | ε

```plaintext
expr → expr + term | expr - term | term
term → id

expr → term rest
rest → + term rest | - term rest | ε
term → id
```
A left recursive grammar has rules that support the derivation: $A \Rightarrow A\alpha$, for some $\alpha$.

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

$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \ldots \text{etc.}$  

$A \rightarrow A\alpha | \beta$

Take left recursive grammar:

$A \rightarrow A\alpha | \beta$

To the following:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon$
Resolving Difficulties : Left Recursion (2)

Informal Discussion:

Take all productions for $A$ 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 $\beta_i$ begins with $A$.

Now apply concepts of previous slide:

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

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

For our example:

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

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

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

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid id$$
Resolving Difficulties: Left Recursion (3)

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

\[
S \rightarrow Aa \mid b \quad \{ \quad S \rightarrow Aa \Rightarrow Sda \\
A \rightarrow Ac \mid Sd \mid \epsilon \quad \}
\]

Algorithm:

*Input:* Grammar G with ordered Non-Terminals A₁, ..., Aₙ

*Output:* An equivalent grammar with no left recursion

1. Arrange the non-terminals in some order A₁=start NT,A₂,...,Aₙ
2. for i := 1 to n do begin
   for j := 1 to i - 1 do begin
      for k := 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 \( A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k \) are all current \( A_j \) productions;
      end
   end
   eliminate the immediate left recursion among \( A_i \) productions
end
Using the Algorithm

Apply the algorithm to:  

\[ A_1 \rightarrow A_2 a \mid b \mid \in \]
\[ A_2 \rightarrow A_2 c \mid A_1 d \]

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

\[ A_2 \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma \]

where  \( A_1 \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k \)  are \( A_1 \) productions

in our case  \( A_2 \rightarrow A_1 d \)  becomes  \( A_2 \rightarrow A_2 ad \mid bd \mid d \)

What’s left:  \( A_1 \rightarrow A_2 a \mid b \mid \in \)

\[ A_2 \rightarrow A_2 c \mid A_2 ad \mid bd \mid d \]

Are we done?
Using the Algorithm (2)

No! We must still remove $A_2$ left recursion!

$$A_1 \rightarrow A_2a \mid b \mid \varepsilon$$

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

Recall:

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

$$A \rightarrow \beta_1A' \mid \beta_2A' \mid \ldots \mid \beta_nA'$$

$$A' \rightarrow \alpha_1A' \mid \alpha_2A' \mid \ldots \mid \alpha_mA' \mid \varepsilon$$

Apply to above case. What do you get?
Removing Difficulties : Left Factoring

Problem : Uncertain which of 2 rules to choose:

\[ stmt \rightarrow \text{if} \ expr \ \text{then} \ stmt \ \text{else} \ stmt \]

\[ /\text{if} \ expr \ \text{then} \ stmt \]

When do you know which one is valid ?

What’s the general form of \( stmt \)?

\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \]

\[ \alpha : \text{if} \ expr \ \text{then} \ stmt \]

\[ \beta_1 : \text{else} \ stmt \]

\[ \beta_2 : \in \]

Transform to:

\[ A \rightarrow \alpha \ A' \]

\[ A' \rightarrow \beta_1 \mid \beta_2 \]

EXAMPLE:

\[ stmt \rightarrow \text{if} \ expr \ \text{then} \ stmt \ rest \]

\[ rest \rightarrow \text{else} \ stmt \ / \in \]
Motivating Table-Driven Parsing

1. Left to right scan input

2. Find leftmost derivation

Grammar:

\[ E \rightarrow TE' \]
\[ E' \rightarrow +TE' | \epsilon \]
\[ T \rightarrow id \]

Input: \( id + id \) $

Terminator

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) ⇔ when $A \rightarrow \alpha | \beta$
  1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$; besides, only one of $\alpha$ or $\beta$ can derive $\epsilon$
  2. if $\alpha$ derives $\epsilon$, then $\text{Follow}(A) \cap \text{First}(\beta) = \emptyset$

**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 \neq \$ $, 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 $\rightarrow$ call recovery routine
   if $M[X,a] = \{X \rightarrow UVW\}$, 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 $ip$;

if $X$ is terminal or $\$$ then

if $X=a$ then

pop $X$ from the stack and advance $ip$

else  $\text{error()}$

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 $X \rightarrow Y_1 Y_2 \ldots Y_k$

end

else  $\text{error()}$

until $X=\$$ /* stack is empty */

Input pointer

May also execute other code based on the production used
Example

E → TE'
E' → + TE' | ∈
T → FT'
T' → * FT' | ∈
F → ( E ) | id

Our well-worn example!

Table M

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>E</td>
<td>E→TE’</td>
</tr>
<tr>
<td>E’</td>
<td>E’→+TE’</td>
</tr>
<tr>
<td>T</td>
<td>T→FT’</td>
</tr>
<tr>
<td>T’</td>
<td>T’→ε</td>
</tr>
<tr>
<td>F</td>
<td>F→id</td>
</tr>
</tbody>
</table>
Trace of Example

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Trace of Example

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id + id * id$</td>
<td></td>
</tr>
<tr>
<td>$E'T$</td>
<td>id + id * id$</td>
<td>E→TE’</td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>id + id * id$</td>
<td>T→FT’</td>
</tr>
<tr>
<td>$E’T’id$</td>
<td>id + id * id$</td>
<td>F→id</td>
</tr>
<tr>
<td>$E’T’$</td>
<td>+ id * id$</td>
<td></td>
</tr>
<tr>
<td>$E’$</td>
<td>+ id * id$</td>
<td>T’→ε</td>
</tr>
<tr>
<td>$E’T+$</td>
<td>+ id * id$</td>
<td>E’→+TE’</td>
</tr>
<tr>
<td>$E’T$</td>
<td>id * id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>id * id$</td>
<td>T→FT’</td>
</tr>
<tr>
<td>$E’T’F*$</td>
<td>* id$</td>
<td>T’→*FT’</td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’id$</td>
<td>id$</td>
<td>F→id</td>
</tr>
<tr>
<td>$E’T’$</td>
<td>* id$</td>
<td></td>
</tr>
<tr>
<td>$E’T$</td>
<td>id * id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F*$</td>
<td>* id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’id$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E’T$</td>
<td>$</td>
<td>T’→ε</td>
</tr>
<tr>
<td>$E’$</td>
<td>$</td>
<td>E’→ε</td>
</tr>
</tbody>
</table>

**Expend Input**
The leftmost derivation for the example is as follows:

\[ E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow id \ T'E' \Rightarrow id \ E' \Rightarrow id + TE' \Rightarrow id + FT'E' \]
\[ \Rightarrow id + id \ T'E' \Rightarrow id + id \times FT'E' \Rightarrow id + id \times id \ T'E' \]
\[ \Rightarrow id + id \times id \ E' \Rightarrow id + id \times id \]
What’s the Missing Puzzle Piece?

Constructing the Parsing Table M!

1\textsuperscript{st}: Calculate First & Follow for Grammar

2\textsuperscript{nd}: Apply Construction Algorithm for Parsing Table
    (We’ll see this shortly)

Basic Tools:

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

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. ($S \Rightarrow \alpha Aa\beta$, for some $\alpha$ and $\beta$).
    NOTE: If $S \Rightarrow \alpha A$, 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($\alpha$)? Add $A \rightarrow \alpha$ to $M[A, a]$

3. $\varepsilon$ in First($\alpha$)? 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

<table>
<thead>
<tr>
<th>Grammar Production</th>
<th>First(S)</th>
<th>Follow(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → i E t SS’</td>
<td>{ i, a }</td>
<td>{ e, $ }</td>
</tr>
<tr>
<td>S’ → eS</td>
<td>{ e, $ }</td>
<td>{ e, $ }</td>
</tr>
<tr>
<td>E → b</td>
<td>{ b }</td>
<td>{ t }</td>
</tr>
</tbody>
</table>
### Constructing Parsing Table – Example 1

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a</td>
</tr>
<tr>
<td>S</td>
<td>S → a</td>
</tr>
<tr>
<td>S’</td>
<td>S → iEtSS’</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>S’</td>
<td>S’ → e</td>
</tr>
<tr>
<td>S’</td>
<td>S’ → eS</td>
</tr>
<tr>
<td>E</td>
<td>E → b</td>
</tr>
</tbody>
</table>

First(S) = \{ i, a \}  
First(S’) = \{ e, \epsilon \}  
First(E) = \{ b \}  
Follow(S) = \{ e, \$ \}  
Follow(S’) = \{ e, \$ \}  
Follow(E) = \{ t \}
Constructing Parsing Table – Example 2

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>First(E,F,T)</th>
<th>Follow(E,E’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → TE’</td>
<td>{ (, id }</td>
<td></td>
</tr>
<tr>
<td>E’ → + TE’</td>
<td>+, ∈</td>
<td></td>
</tr>
<tr>
<td>T → FT’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’ → * FT’</td>
<td>*, ∈</td>
<td></td>
</tr>
<tr>
<td>F → ( E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Follow(F) = { *, +, ), $ }
Follow(T,T’) = { +, ), $}
Constructing Parsing Table – Example 2

<table>
<thead>
<tr>
<th>Production</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow TE' )</td>
<td>{ (, id }</td>
<td>{ (, ) }</td>
</tr>
<tr>
<td>( E' \rightarrow + TE' \mid \in )</td>
<td>{ +, \in }</td>
<td>{ * , + , (, $ }</td>
</tr>
<tr>
<td>( T \rightarrow FT' )</td>
<td>{ * , \in }</td>
<td>{ * , + , (, $ }</td>
</tr>
<tr>
<td>( T' \rightarrow * FT' \mid \in )</td>
<td>{ * , \in }</td>
<td>{ + , ) , $ }</td>
</tr>
<tr>
<td>( F \rightarrow ( E ) \mid \text{id} )</td>
<td>{ (, id }</td>
<td>{ (, ) }</td>
</tr>
</tbody>
</table>

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

\[
\text{by rule 2}
\]

\[
M[E, ( ] : E \rightarrow TE' \\
M[E, \text{id} ] : E \rightarrow TE'
\]

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

(by rule 3) \( E' \rightarrow \in : \in \) in First(\(\in\)) 

\[
M[E’, )] : E’ \rightarrow \in (3) \\
M[E’, $] : E’ \rightarrow \in (3)
\]

(by rule 3) \( T’ \rightarrow \in : \in \) in First(\(\in\)) 

\[
M[T’, +] : T’ \rightarrow \in (3) \\
M[T’, ] : T’ \rightarrow \in (3)
\]

(Due to Follow(E’)) 

\[
M[T’, $] : T’ \rightarrow \in (3)
\]
Resolving Problems: Ambiguous Grammars

Consider the following grammar segment:

\[ stmt \rightarrow \text{if } expr \text{ then } stmt \]

\[ \mid \text{if } expr \text{ then } stmt \text{ else } stmt \]

\[ \mid \text{other } (\text{any other statement}) \]

What’s problem here?

Let’s consider a simple parse tree:

Else must match to previous then.
Parse Trees for Example

Form 1:

```
stmt
  /  \  
expr  then
  /    
E_1   stmt
      /  |
     then else
      /    |
E_2   stmt
      /  |
     then S_1
      /    |
S_2   stmt
```

Form 2:

```
stmt
  /  |
expr  then
  /    |
E_1   stmt
      /  |
     then else
      /    |
E_2   stmt
      /  |
     then S_1
      /    |
S_2   stmt
```

What’s the issue here?
Removing Ambiguity

Take Original Grammar:

\[
stmt \rightarrow \text{if expr then stmt} \\
| \text{if expr then stmt else stmt} \\
| \text{other (any other statement)}
\]

Or to write more simply:

\[
S \rightarrow i E t S \\
| i E t S e S \\
| S \\
E \rightarrow a
\]

The problem string: \text{i a t i a t s e s}
Revise to remove ambiguity:

\[ S \rightarrow i\ E\ t\ S \]
\[ \quad \mid i\ E\ t\ S\ e\ S \]
\[ \quad \mid s \]
\[ E \rightarrow a \]

\[ S \rightarrow M \mid U \]
\[ M \rightarrow i\ E\ t\ M\ e\ M \mid s \]
\[ U \rightarrow i\ E\ t\ S \mid i\ E\ t\ M\ e\ U \]
\[ E \rightarrow a \]

Try the above on \texttt{i a t i a t s e s}

\[ \text{stmt} \rightarrow \text{matched_stmt} | \text{unmatched_stmt} \]

\[ \text{matched_stmt} \rightarrow \text{if}\ \text{expr}\ \text{then}\ \text{matched_stmt}\ \text{else}\ \text{matched_stmt} / \text{other} \]

\[ \text{unmatched_stmt} \rightarrow \text{if}\ \text{expr}\ \text{then}\ \text{stmt} \]
\[ \quad \mid \text{if}\ \text{expr}\ \text{then}\ \text{matched}\ \text{stmt}\ \text{else}\ \text{unmatched}\ \text{stmt} \]
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 ⇒ miss declaration – causing more errors
⇒ miss errors in skipped material

-- Advantages:
simple ⇒ suited to 1 error per statement

Phrase Level – Local correction on input
-- “,” ⇒ “;” – 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 do many changes!
-- Algorithms available to minimize changes costly - key issues
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 $\epsilon$ 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

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>E</td>
<td>E→TE'</td>
</tr>
<tr>
<td>E'</td>
<td>___</td>
</tr>
<tr>
<td>T</td>
<td>T→FT'</td>
</tr>
<tr>
<td>T'</td>
<td>___</td>
</tr>
<tr>
<td>F</td>
<td>F→id</td>
</tr>
</tbody>
</table>

From Follow sets. Pop top of stack NT

“synch” action

Skip input symbol
### Revised Parsing Table / Example(2)

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>+ id * + id$</td>
<td>error, skip +</td>
</tr>
<tr>
<td>$E$</td>
<td>id * + id$</td>
<td></td>
</tr>
<tr>
<td>$E’T$</td>
<td>id * + id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>id * + id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’id$</td>
<td>id * + id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’$</td>
<td>* + id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F*$</td>
<td>* + id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>+ id$</td>
<td>error, M[F,+] = synch F has been popped</td>
</tr>
<tr>
<td>$E’T$</td>
<td>+ id$</td>
<td></td>
</tr>
<tr>
<td>$E’$</td>
<td>+ id$</td>
<td></td>
</tr>
<tr>
<td>$E’T+$</td>
<td>+ id$</td>
<td></td>
</tr>
<tr>
<td>$E’T$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’F$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’id$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E’T’$</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$E’$</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

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
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”
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

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E→TE’</td>
<td></td>
<td></td>
<td></td>
<td>E→TE’</td>
<td>synch</td>
<td>synch</td>
</tr>
<tr>
<td>E’</td>
<td>E’→+TE’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E’→ε</td>
<td>E’→ε</td>
</tr>
<tr>
<td>T</td>
<td>T→FT’</td>
<td>synch</td>
<td></td>
<td></td>
<td>T→FT’</td>
<td>synch</td>
<td>synch</td>
</tr>
<tr>
<td>T’</td>
<td>T’→ε</td>
<td>synch</td>
<td>T’→*FT’</td>
<td></td>
<td>T’→ε</td>
<td>T’→ε</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F→id</td>
<td>synch</td>
<td>synch</td>
<td>F→(E)</td>
<td>synch</td>
<td>synch</td>
<td></td>
</tr>
</tbody>
</table>

All rules have unique IDs

Ditto for synch actions

Also for blanks which handle errors
Revised Parsing Table:

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E’</td>
<td>20</td>
</tr>
<tr>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>T’</td>
<td>24</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
</tr>
</tbody>
</table>

1 E→TE’
2 E’→+TE’
3 E’→ε
4 T→FT’
5 T’→*FT’
6 T’→ε
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.

Examples:

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:

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

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

$a^n b^m : \text{formal parameter}$

$c^n d^m : \text{actual parameter}$
How do you show a Language is a CFL?

\[ L_3 = \{ \ w \ c \ w^R \ | \ w \text{ is in } (a \mid b)^* \} \]

\[ L_4 = \{ \ a^n b^m c^m d^n \ | \ n \geq 1, \ m \geq 1 \} \]

\[ L_5 = \{ \ a^n b^n c^m d^m \ | \ n \geq 1, \ m \geq 1 \} \]

\[ L_6 = \{ \ a^n b^n \ | \ n \geq 1 \} \]
Solutions

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

\[
S \rightarrow aS\ a \mid bS\ b \mid c
\]

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

\[
S \rightarrow aS\ d \mid a\ A\ d \\
A \rightarrow b\ A\ c \mid bc
\]

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

\[
S \rightarrow XY \\
X \rightarrow a\ X\ b \mid ab \\
Y \rightarrow c\ Y\ d \mid cd
\]

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

\[
S \rightarrow a\ S\ b \mid ab
\]