

# Compilerconstructie

najaar 2015

<http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/>

**Rudy van Vliet**

kamer 124 Snellius, tel. 071-527 5777

rvvliet(at)liacs(dot)nl

college 3, woensdag 16 september 2015

+ werkcollege

Syntax Analysis (1)

## 4 Syntax Analysis

- Every language has rules prescribing the syntactic structure of the programs:
  - functions, made up of declarations and statements
  - statements made up of expressions
  - expressions made up of tokens
- CFG can describe (part of) syntax of programming-language constructs.
  - Precise syntactic specification
  - Automatic construction of parsers for certain classes of grammars
  - Structure imparted to language by grammar is useful for translating source programs into object code
  - New language constructs can be added easily
- Parser checks/determines syntactic structure

## 4.3.5 Non-CF Language Constructs

- Declaration of identifiers before their use

$$L_1 = \{w c w \mid w \in \{a, b\}^*\}$$

- Number of formal parameters in function declaration equals number of actual parameters in function call  
Function call may be specified by

$$\begin{aligned} stmt &\rightarrow \mathbf{id} (expr\_list) \\ expr\_list &\rightarrow expr\_list, expr \mid expr \end{aligned}$$

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

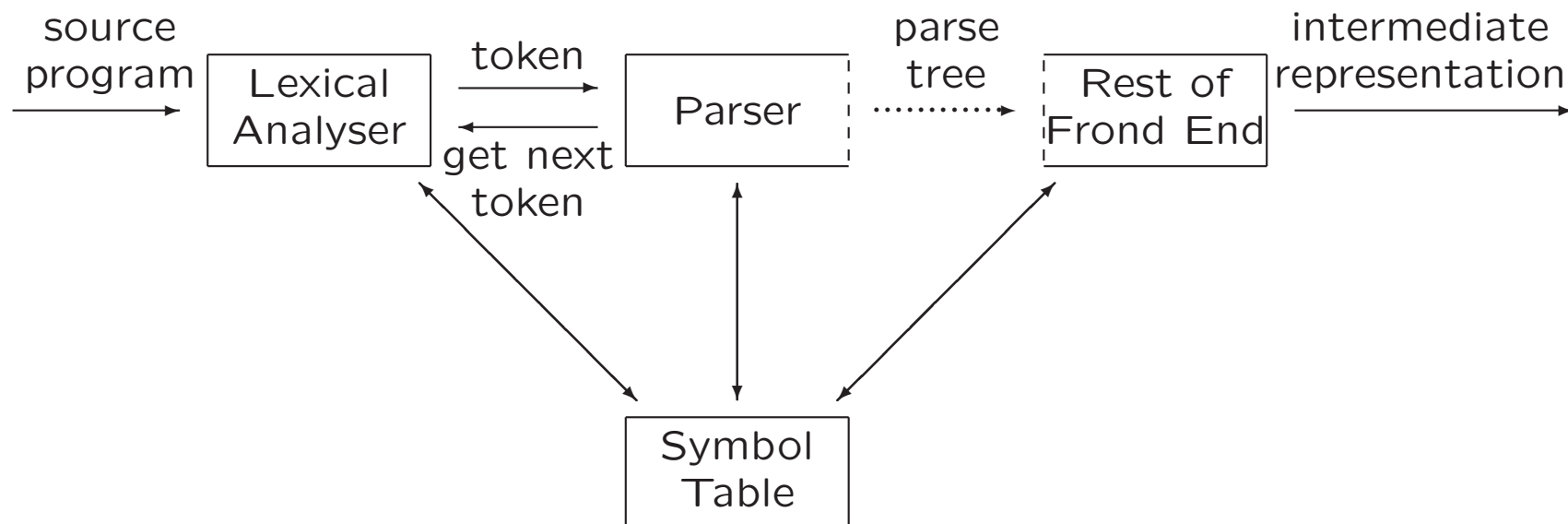
Such checks are performed during semantic-analysis phase

## 2.4 Parsing

- Process of determining if a string of tokens can be generated by a grammar
- For any context-free grammar, there is a parser that takes at most  $\mathcal{O}(n^3)$  time to parse a string of  $n$  tokens
- Linear algorithms sufficient for parsing programming languages
- Two methods of parsing:
  - **Top-down** constructs parse tree from root to leaves
  - **Bottom-up** constructs parse tree from leaves to root

Cf. top-down PDA and bottom-up PDA in FI2

## 4.1.1 The Role of the Parser



- Obtain string of tokens
- Verify that string can be generated by the grammar
- Report and recover from syntax errors

# Parsing

Finding parse tree for given string

- Universal (any CFG)
  - Cocke-Younger-Kasami
  - Earley
- Top-down (CFG with restrictions)
  - Predictive parsing
  - LL (Left-to-right, Leftmost derivation) methods
  - LL(1): LL parser, needs only one token to look ahead
- Bottom-up (CFG with restrictions)

Today: top-down parsing

Next week: bottom-up parsing

## 4.2 Context-Free Grammars

Context-free grammar is a 4-tuple with

- A set of *nonterminals* (syntactic variables)
- A set of tokens (*terminal* symbols)
- A designated *start* symbol (nonterminal)
- A set of *productions*: rules how to decompose nonterminals

Example: CFG for simple arithmetic expressions:

$$G = (\{expr, term, factor\}, \{\mathbf{id}, +, -, *, /, (, )\}, expr, P)$$

with productions  $P$ :

$$\begin{aligned} expr &\rightarrow expr + term \mid expr - term \mid term \\ term &\rightarrow term * factor \mid term / factor \mid factor \\ factor &\rightarrow (expr) \mid \mathbf{id} \end{aligned}$$

## 4.2.2 Notational Conventions

1. Terminals:

$a, b, c, \dots$ ; specific terminals:  $+, *, (, ), 0, 1, \mathbf{id}, \mathbf{if}, \dots$

2. Nonterminals:

$A, B, C, \dots$ ; specific nonterminals:  $S, expr, stmt, \dots, E, \dots$

3. Grammar symbols:  $X, Y, Z$

4. Strings of terminals:  $u, v, w, x, y, z$

5. Strings of grammar symbols:  $\alpha, \beta, \gamma, \dots$

Hence, generic production:  $A \rightarrow \alpha$

6.  $A$ -productions:

$A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k \quad \Rightarrow \quad A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$   
*Alternatives for  $A$*

7. By default, head of first production is start symbol



# Notational Conventions (Example)

CFG for simple arithmetic expressions:

$$G = (\{expr, term, factor\}, \{\mathbf{id}, +, -, *, /, (, )\}, expr, P)$$

with productions  $P$ :

$$\begin{aligned} expr &\rightarrow expr + term \mid expr - term \mid term \\ term &\rightarrow term * factor \mid term / factor \mid factor \\ factor &\rightarrow (expr) \mid \mathbf{id} \end{aligned}$$

Can be rewritten concisely as:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

## 4.2.3 Derivations

Example grammar:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

- In each step, a nonterminal is replaced by body of one of its productions, e.g.,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

- One-step derivation:  
 $\alpha A \beta \Rightarrow \alpha \gamma \beta$ , where  $A \rightarrow \gamma$  is production in grammar
- Derivation in zero or more steps:  $\Rightarrow^*$
- Derivation in one or more steps:  $\Rightarrow^+$

# Derivations

- If  $S \xRightarrow{*} \alpha$ , then  $\alpha$  is **sentential form** of  $G$
- If  $S \xRightarrow{*} \alpha$  and  $\alpha$  has no nonterminals, then  $\alpha$  is **sentence** of  $G$
- **Language generated by  $G$**  is  $L(G) = \{w \mid w \text{ is sentence of } G\}$
- **Leftmost derivation**:  $wA\gamma \xRightarrow{lm} w\delta\gamma$
- If  $S \xRightarrow{lm}^* \alpha$ , then  $\alpha$  is **left sentential form** of  $G$
- **Rightmost derivation**:  $\gamma Aw \xRightarrow{rm} \gamma\delta w, \xRightarrow{rm}^*$

Example of leftmost derivation:

$$E \xRightarrow{lm} -E \xRightarrow{lm} -(E) \xRightarrow{lm} -(E + E) \xRightarrow{lm} -(\mathbf{id} + E) \xRightarrow{lm} -(\mathbf{id} + \mathbf{id})$$

# Parse Tree

(from lecture 1)

(derivation tree in FI2)

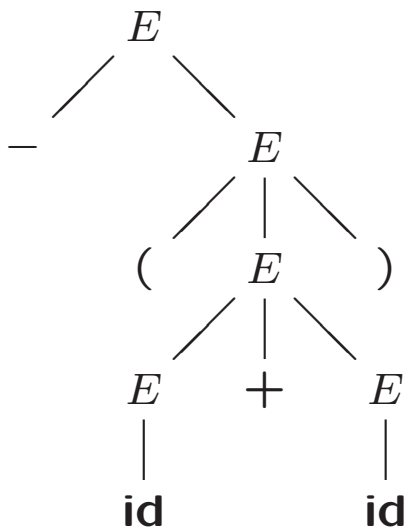
- The root of the tree is labelled by the start symbol
- Each leaf of the tree is labelled by a terminal (=token) or  $\epsilon$  (=empty)
- Each interior node is labelled by a nonterminal
- If node  $A$  has children  $X_1, X_2, \dots, X_n$ , then there must be a production  $A \rightarrow X_1 X_2 \dots X_n$

**Yield** of the parse tree: the sequence of leafs (left to right)

## 4.2.4 Parse Trees and Derivations

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

$$E \xRightarrow{lm} -E \xRightarrow{lm} -(E) \xRightarrow{lm} -(E + E) \xRightarrow{lm} -(\mathbf{id} + E) \xRightarrow{lm} -(\mathbf{id} + \mathbf{id})$$



Many-to-one relationship between derivations and parse trees. . .

## 4.2.5 Ambiguity

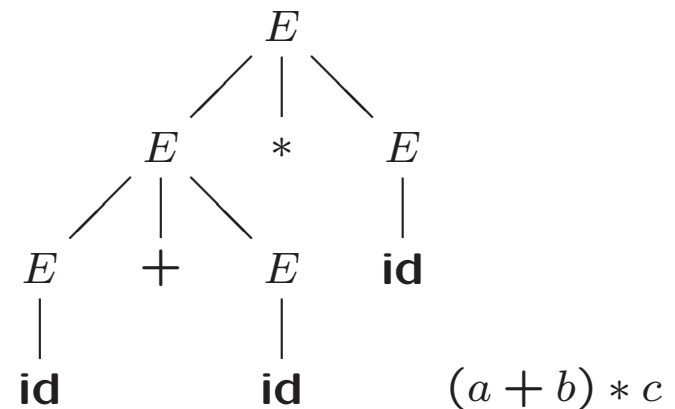
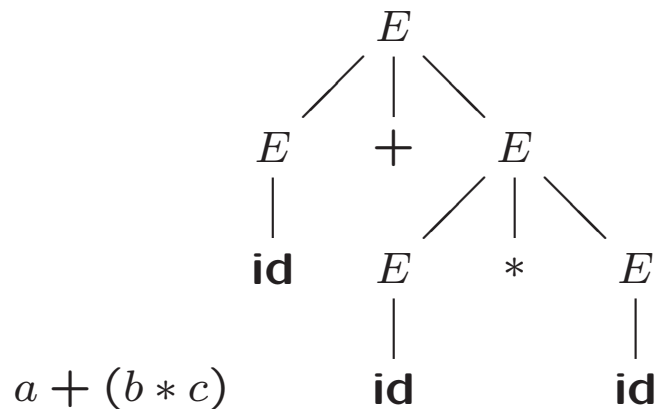
More than one leftmost/rightmost derivation for same sentence

Example:

$a + b * c$

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

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



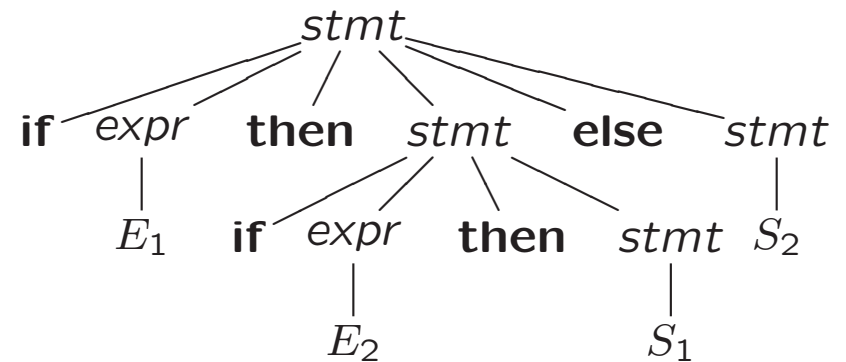
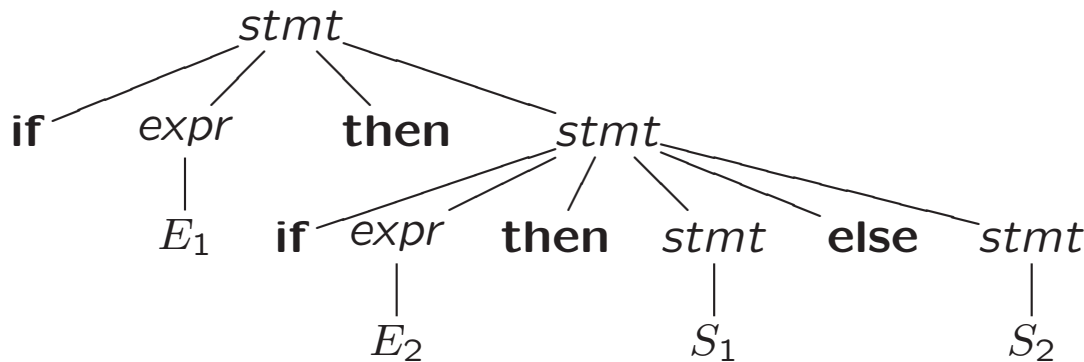
## 4.3.2 Eliminating ambiguity

- Sometimes ambiguity can be eliminated
- Example: “dangling-else”-grammar

$$\begin{array}{l} stmt \rightarrow \text{if } expr \text{ then } stmt \\ \quad | \text{if } expr \text{ then } stmt \text{ else } stmt \\ \quad | \text{other} \end{array}$$

Here, **other** is any other statement

**if**  $E_1$  **then** **if**  $E_2$  **then**  $S_1$  **else**  $S_2$



# Eliminating ambiguity

Example: ambiguous “dangling-else”-grammar

```
stmt → if expr then stmt  
      | if expr then stmt else stmt  
      | other
```

Only matched statements between **then** and **else**...



# Eliminating ambiguity

Example: ambiguous “dangling-else”-grammar

$$\begin{aligned} stmt &\rightarrow \mathbf{if\ expr\ then\ stmt} \\ &\quad | \mathbf{if\ expr\ then\ stmt\ else\ stmt} \\ &\quad | \mathbf{other} \end{aligned}$$

Equivalent unambiguous grammar

$$\begin{aligned} stmt &\rightarrow \mathit{matchedstmt} \\ &\quad | \mathit{openstmt} \\ \mathit{matchedstmt} &\rightarrow \mathbf{if\ expr\ then\ matchedstmt\ else\ matchedstmt} \\ &\quad | \mathbf{other} \\ \mathit{openstmt} &\rightarrow \mathbf{if\ expr\ then\ stmt} \\ &\quad | \mathbf{if\ expr\ then\ matchedstmt\ else\ openstmt} \end{aligned}$$

Only one parse tree for

**if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$**

Associates each **else** with closest previous unmatched **then**

## 2.4.1 Top-Down Parsing (Example)

$stmt \rightarrow$  **expr ;**  
| **if (expr )stmt**  
| **for (optexpr ; optexpr ; optexpr )stmt**  
| **other**

$optexpr \rightarrow$   $\epsilon$   
| **expr**

How to determine parse tree for

**for (; expr ; expr )other**

Use lookahead: current terminal in input...

## 2.4.2 Predictive Parsing

- Recursive-descent parsing is a top-down parsing method:
  - Executes a set of recursive procedures to process the input
  - Every nonterminal has one (recursive) procedure parsing the nonterminal's syntactic category of input tokens
- Predictive parsing . . .

## 4.4.1 Recursive Descent Parsing

Recursive procedure for each nonterminal

```
void A()  
1) { Choose an A-production,  $A \rightarrow X_1X_2 \dots X_k$ ;  
2)  for ( $i = 1$  to  $k$ )  
3)    { if ( $X_i$  is nonterminal)  
4)      call procedure  $X_i()$ ;  
5)      else if ( $X_i$  equals current input symbol  $a$ )  
6)        advance input to next symbol;  
7)      else /* an error has occurred */;  
    }  
}
```

Pseudocode is nondeterministic

# Recursive-Descent Parsing

- One may use backtracking:
  - Try each  $A$ -production in some order
  - In case of failure at line 7 (or call in line 4), return to line 1 and try another  $A$ -production
  - Input pointer must then be reset, so store initial value input pointer in local variable
- Example in book
- Backtracking is rarely needed: predictive parsing

## 2.4.2 Predictive Parsing

- Recursive-descent parsing ...
- Predictive parsing is a special form of recursive-descent parsing:
  - The lookahead symbol(s) unambiguously determine(s) the production for each nonterminal

Simple example:

```
stmt → expr ;  
      | if (expr) stmt  
      | for (optexpr ; optexpr ; optexpr) stmt  
      | other
```

# Predictive Parsing (Example)

```
void stmt()
{ switch (lookahead)
  { case expr:
      match(expr); match(';'); break;
    case if:
      match(if); match('('); match(expr); match(')'); stmt();
      break;
    case for:
      match(for); match('(');
      optexpr(); match(';'); optexpr(); match(';'); optexpr();
      match(')'); stmt(); break;
    case other:
      match(other); break;
    default:
      report("syntax error");
  }
}

void match(terminal t)
{ if (lookahead==t) lookahead = nextTerminal;
  else report("syntax error");
}
```

## Using FIRST (simple case)

- Let  $\alpha$  be string of grammar symbols
- $\text{FIRST}(\alpha)$  = set of terminals/tokens that appear as first symbols of strings derived from  $\alpha$

Simple example:

```
stmt → expr ;  
      | if (expr) stmt  
      | for (optexpr ; optexpr ; optexpr) stmt  
      | other
```

Right-hand side may start with nonterminal...



## Using FIRST (simple case)

- Let  $\alpha$  be string of grammar symbols
- $\text{FIRST}(\alpha)$  = set of terminals/tokens that appear as first symbols of strings derived from  $\alpha$
- When a nonterminal has multiple productions, e.g.,

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

then  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  must be disjoint in order for predictive parsing to work

## 2.4.3 When to Use $\epsilon$ -Productions (simple solution)

Simple example:

```
stmt  → expr ;  
      | if (expr) stmt  
      | for (optexpr ; optexpr ; optexpr) stmt  
      | other  
optexpr → expr  
      |  $\epsilon$ 
```

# Predictive Parsing (Example)

```
void stmt()
{ switch (lookahead)
  { case expr: ...
    case if: ...
    case for:
      match(for); match('(');
      optexpr(); match(';'); optexpr(); match(';'); optexpr();
      match(')'); stmt(); break;
    case other; ...
    default: ...
  }
}
```

```
void optexpr()
{ if (lookahead==expr)
  match (expr);
}
```

```
void match(terminal t)
{ if (lookahead==t) lookahead = nextTerminal;
  else report("syntax error");
}
```

## 2.4.5 Left Recursion

- Productions of the form  $A \rightarrow A\alpha \mid \beta$  are left-recursive
  - $\beta$  does not start with  $A$
  - Example:

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

$$T \rightarrow \mathbf{id}$$

- $\text{FIRST}(E + T) \cap \text{FIRST}(T) = \{\mathbf{id}\} \neq \emptyset$
- Top-down parser may loop forever if grammar has left-recursive productions
- Left-recursive productions can be eliminated by rewriting productions

## 4.3.3 Elimination of Left Recursion

### Immediate left recursion

- Productions of the form  $A \rightarrow A\alpha \mid \beta$
- Can be eliminated by replacing the productions by

$$\begin{array}{ll} A \rightarrow \beta A' & (A' \text{ is new nonterminal}) \\ A' \rightarrow \alpha A' \mid \epsilon & (A' \rightarrow \alpha A' \text{ is right recursive}) \end{array}$$

- Procedure:

1. Group  $A$ -productions as

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

2. Replace  $A$ -productions by

$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}$$

# Elimination of Left Recursion

## Immediate left recursion

- Productions of the form  $A \rightarrow A\alpha \mid \beta$
- Can be eliminated by replacing the productions by

$$\begin{array}{ll} A \rightarrow \beta A' & (A' \text{ is new nonterminal}) \\ A' \rightarrow \alpha A' \mid \epsilon & (A' \rightarrow \alpha A' \text{ is right recursive}) \end{array}$$

Example:

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow \mathbf{id} \end{array}$$

- New grammar...
- Derivation trees for  $\mathbf{id}_1 + \mathbf{id}_2 + \mathbf{id}_3 + \mathbf{id}_4 \dots$

# Elimination of Left Recursion

## General left recursion

- Left recursion involving two or more steps

$$\begin{aligned} S &\rightarrow Ba \mid b \\ B &\rightarrow AA \mid a \\ A &\rightarrow Ac \mid Sd \end{aligned}$$

- $S$  is left-recursive because

$$S \Rightarrow Ba \Rightarrow AAa \Rightarrow SdAa \quad (\text{not immediately left-recursive})$$

# Elimination of General Left Recursion

$$\begin{aligned} S &\rightarrow Ba \mid b \\ B &\rightarrow AA \mid a \\ A &\rightarrow Ac \mid Sd \end{aligned}$$

- We order nonterminals:  $S, B, A$  ( $n = 3$ )
- Variables may only 'point forward'
- $i = 1$  and  $i = 2$ : nothing to do
- $i = 3$ :
  - substitute  $A \rightarrow Sd$
  - substitute  $A \rightarrow Bad$
  - eliminate immediate left-recursion in  $A$ -productions



# Elimination of General Left Recursion

Algorithm for  $G$  with **no cycles or  $\epsilon$ -productions**

- 1) arrange nonterminals in some order  $A_1, A_2, \dots, A_n$
- 2) **for** ( $i = 1$  to  $n$ )
- 3) { **for** ( $j = 1$  to  $i - 1$ )
- 4) { replace each production of form  $A_i \rightarrow A_j\gamma$   
by the productions  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$ , where  
 $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all current  $A_j$ -productions
- 5) }
- 6) eliminate immediate left recursion among  $A_i$ -productions
- 7) }

Example with  $A \rightarrow \epsilon$  (well/wrong.....)

## 4.3.4 Left Factoring

Another transformation to produce grammar suitable for predictive parsing

- If  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$  and input begins with nonempty string derived from  $\alpha$   
How to expand  $A$ ? To  $\alpha\beta_1$  or to  $\alpha\beta_2$ ?

## 4.3.4 Left Factoring

Another transformation to produce grammar suitable for predictive parsing

- If  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$  and input begins with nonempty string derived from  $\alpha$   
How to expand  $A$ ? To  $\alpha\beta_1$  or to  $\alpha\beta_2$ ?
- Solution: left-factoring  
Replace two  $A$ -productions by

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

- $|\alpha|$  may be  $\geq 2$

## Left Factoring (Example)

- Which production to choose when input token is **if**?

$$\begin{array}{l} stmt \rightarrow \mathbf{if} \ expr \ \mathbf{then} \ stmt \\ \quad | \ \mathbf{if} \ expr \ \mathbf{then} \ stmt \ \mathbf{else} \ stmt \\ \quad | \ \mathbf{other} \\ expr \rightarrow b \end{array}$$

- Or abstract:

$$\begin{array}{l} S \rightarrow iEtS \mid iEtSeS \mid a \\ E \rightarrow b \end{array}$$

- Left-factored: . . .

# Left Factoring (Example)

- Which production to choose when input token is **if**?  
Abstract:

$$\begin{aligned} S &\rightarrow iEtS \mid iEtSeS \mid a \\ E &\rightarrow b \end{aligned}$$

- Left-factored:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow \epsilon \mid eS \\ E &\rightarrow b \end{aligned}$$

Of course, still ambiguous...

## Left Factoring (Example)

What is result of left factoring for

$$S \rightarrow abS \mid abcA \mid aaa \mid aab \mid aA$$

## 4.4 Top-Down Parsing

- Construct parse tree,
  - starting from the root
  - creating nodes in preorder

Corresponds to finding leftmost derivation

# Top-Down Parsing (Example)

- 

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

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

$$F \rightarrow (E) \mid \mathbf{id}$$

- Non-left-recursive variant: ...



# Top-Down Parsing (Example)

- 

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

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

$$F \rightarrow (E) \mid \mathbf{id}$$

- Non-left-recursive variant:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \mathbf{id}$$

- Top-down parse for input **id + id \* id ...**
- At each step: determine production to be applied

# Top-Down Parsing

- Recursive-descent parsing
- Predictive parsing
  - Eliminate left-recursion from grammar
  - Left-factor the grammar
  - Compute FIRST and FOLLOW
  - Two variants:
    - \* Recursive (recursive calls)
    - \* Non-recursive (explicit stack)

## 4.4.2 FIRST (and Follow)

- Let  $\alpha$  be string of grammar symbols
- $\text{FIRST}(\alpha)$  = set of terminals/tokens that appear as first symbols of strings derived from  $\alpha$
- If  $\alpha \xRightarrow{*} \epsilon$ , then  $\epsilon \in \text{FIRST}(\alpha)$
- Example

$$F \rightarrow (E) \mid \mathbf{id}$$

$$\text{FIRST}(FT') = \{(\, \mathbf{id}\}$$

- When nonterminal has multiple productions, e.g.,

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

and  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  are disjoint,  
we can choose between these  $A$ -productions by looking at  
next input symbol

# Computing FIRST

Compute  $\text{FIRST}(X)$  for **all** grammar symbols  $X$ :

- If  $X$  is **terminal**, then  $\text{FIRST}(X) = \{X\}$
- If  $X \rightarrow \epsilon$  is production, then add  $\epsilon$  to  $\text{FIRST}(X)$
- Repeat adding symbols to  $\text{FIRST}(X)$  by looking at productions

$$X \rightarrow Y_1 Y_2 \dots Y_k$$

(see book) until all FIRST sets are stable

# FIRST (Example)

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

| nonterminal $A$ | FIRST( $A$ ) |
|-----------------|--------------|
| $E$             | ...          |
| $E'$            | ...          |
| $T$             | ...          |
| $T'$            | ...          |
| $F$             | ...          |

Fill in bottom-up...

# FIRST (Example)

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

| nonterminal $A$ | FIRST( $A$ )        |
|-----------------|---------------------|
| $E$             | $\{(\mathbf{id})\}$ |
| $E'$            | $\{+, \epsilon\}$   |
| $T$             | $\{(\mathbf{id})\}$ |
| $T'$            | $\{*, \epsilon\}$   |
| $F$             | $\{(\mathbf{id})\}$ |

## 4.4.2 (First and) FOLLOW

- Let  $A$  be **nonterminal**
- $\text{FOLLOW}(A)$  = set of terminals/tokens that can appear immediately to the right of  $A$  in sentential form:

$$\text{FOLLOW}(A) = \{a \mid S \xRightarrow{*} \alpha A a \beta\}$$

- Example

$$F \rightarrow (E) \mid \mathbf{id}$$

# Computing FOLLOW

Compute FOLLOW( $A$ ) for all nonterminals  $A$ :

- Place \$ in FOLLOW( $S$ )
- For production  $A \rightarrow \alpha B \beta$ ,  
add everything in FIRST( $\beta$ ) to FOLLOW( $B$ ) (except  $\epsilon$ )
- – For production  $A \rightarrow \alpha B$ ,  
add everything in FOLLOW( $A$ ) to FOLLOW( $B$ )
- For production  $A \rightarrow \alpha B \beta$  with  $\epsilon \in \text{FIRST}(\beta)$ ,  
add everything in FOLLOW( $A$ ) to FOLLOW( $B$ )

until all FOLLOW sets are stable



# FIRST and FOLLOW (Example)

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

| nonterminal $A$ | FIRST( $A$ )     | FOLLOW( $A$ ) |
|-----------------|------------------|---------------|
| $E$             | {(, <b>id</b> }  | ...           |
| $E'$            | {+, $\epsilon$ } | ...           |
| $T$             | {(, <b>id</b> }  | ...           |
| $T'$            | {*, $\epsilon$ } | ...           |
| $F$             | {(, <b>id</b> }  | ...           |

Fill in top-down...

# FIRST and FOLLOW (Example)

$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow +TE' \mid \epsilon \\
 T &\rightarrow FT' \\
 T' &\rightarrow *FT' \mid \epsilon \\
 F &\rightarrow (E) \mid \mathbf{id}
 \end{aligned}$$

| nonterminal $A$ | FIRST( $A$ )     | FOLLOW( $A$ )         |
|-----------------|------------------|-----------------------|
| $E$             | {(, <b>id</b> }  | {), <b>\$</b> }       |
| $E'$            | {+, $\epsilon$ } | {), <b>\$</b> }       |
| $T$             | {(, <b>id</b> }  | {+, ), <b>\$</b> }    |
| $T'$            | {*, $\epsilon$ } | {+, ), <b>\$</b> }    |
| $F$             | {(, <b>id</b> }  | {*, +, ), <b>\$</b> } |

## 4.4.3 LL(1) Grammars

When next input symbol is  $a$  (terminal or input endmarker \$), we may choose  $A \rightarrow \alpha$

- if  $a \in \text{FIRST}(\alpha)$
- if  $(\alpha = \epsilon \text{ or } \alpha \xRightarrow{*} \epsilon)$  and  $a \in \text{FOLLOW}(A)$

Algorithm to construct parsing table  $M[A, a]$

```
for (each production  $A \rightarrow \alpha$ )  
{ for (each  $a \in \text{FIRST}(\alpha)$ )  
  add  $A \rightarrow \alpha$  to  $M[A, a]$ ;  
  if ( $\epsilon \in \text{FIRST}(\alpha)$ )  
  { for (each  $a \in \text{FOLLOW}(A)$ )  
    add  $A \rightarrow \alpha$  to  $M[A, a]$ ;  
  }  
}  
}  
If  $M[A, a]$  is empty, set  $M[A, a]$  to error.
```

# Top-Down Parsing Table (Example)

$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow +TE' \mid \epsilon \\
 T &\rightarrow FT' \\
 T' &\rightarrow *FT' \mid \epsilon \\
 F &\rightarrow (E) \mid \mathbf{id}
 \end{aligned}$$

| nonterminal $A$ | FIRST( $A$ )     | FOLLOW( $A$ ) |
|-----------------|------------------|---------------|
| $E$             | {(, <b>id</b> }  | {), \$}       |
| $E'$            | {+, $\epsilon$ } | {), \$}       |
| $T$             | {(, <b>id</b> }  | {+, ), \$}    |
| $T'$            | {*, $\epsilon$ } | {+, ), \$}    |
| $F$             | {(, <b>id</b> }  | {*, +, ), \$} |

| Non-terminal | Input Symbol                |                           |                       |                     |                           |                           |
|--------------|-----------------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
|              | <b>id</b>                   | +                         | *                     | (                   | )                         | \$                        |
| $E$          | $E \rightarrow TE'$         |                           |                       | $E \rightarrow TE'$ |                           |                           |
| $E'$         |                             | $E' \rightarrow +TE'$     |                       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$          | $T \rightarrow FT'$         |                           |                       | $T \rightarrow FT'$ |                           |                           |
| $T'$         |                             | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$          | $F \rightarrow \mathbf{id}$ |                           |                       | $F \rightarrow (E)$ |                           |                           |

# LL(1) Grammars

- LL(1)  
Left-to-right scanning of input, Leftmost derivation,  
1 token to look ahead suffices for predictive parsing
- Grammar  $G$  is LL(1),  
if and only if for two distinct productions  $A \rightarrow \alpha \mid \beta$ ,
  - $\alpha$  and  $\beta$  do not both derive strings beginning with same terminal  $a$
  - at most one of  $\alpha$  and  $\beta$  can derive  $\epsilon$
  - if  $\beta \xRightarrow{*} \epsilon$ , then  $\alpha$  does not derive strings beginning with terminal  $a \in \text{FOLLOW}(A)$
- In other words, . . .
- Grammar  $G$  is LL(1), if and only if parsing table uniquely identifies production or signals error

# LL(1) Grammars (Example)

- Not LL(1):

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

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

$$F \rightarrow (E) \mid \mathbf{id}$$

- Non-left-recursive variant, LL(1):

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \mathbf{id}$$

# Left Factoring (Example)

- Abstract if-then-else-grammar:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

- Left-factored:

$$S \rightarrow iEtSS' \mid a$$

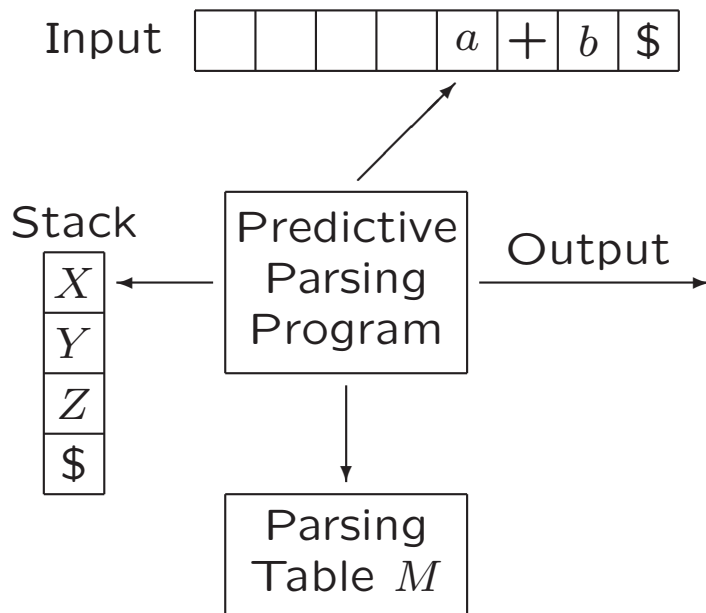
$$S' \rightarrow \epsilon \mid eS$$

$$E \rightarrow b$$

Not LL(1)...

## 4.4.4 Nonrecursive Predictive Parsing

Cf. top-down PDA from FI2



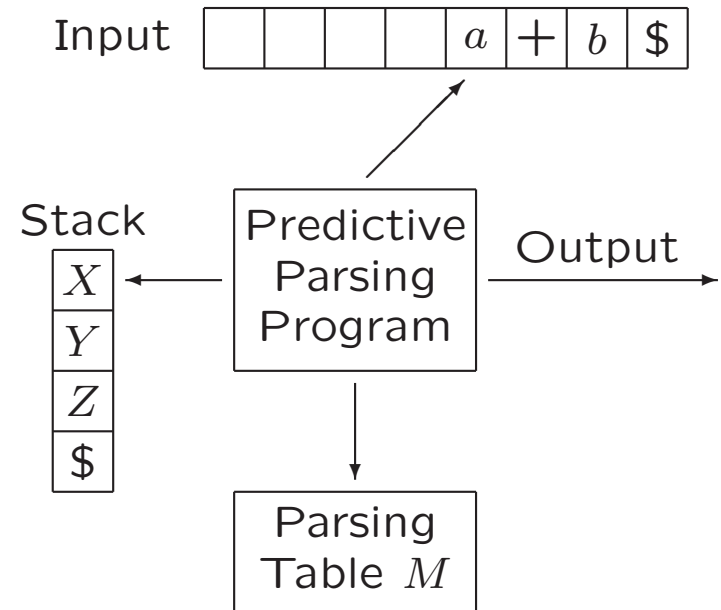


# Nonrecursive Predictive Parsing

```

push $ onto stack;
push S onto stack;
let  $a$  be first symbol of input  $w$ ;
let  $X$  be top stack symbol;
while ( $X \neq \$$ ) /* stack is not empty */
{ if ( $X = a$ )
  { pop stack;
    let  $a$  be next symbol of  $w$ ;
  }
  else if ( $X$  is terminal)
    error();
  else if ( $M[X, a]$  is error entry)
    error();
  else if ( $M[X, a] = X \rightarrow Y_1Y_2 \dots Y_k$ )
    { output production  $X \rightarrow Y_1Y_2 \dots Y_k$ ;
      pop stack;
      push  $Y_k, Y_{k-1}, \dots, Y_1$  onto stack, with  $Y_1$  on top;
    }
let  $X$  be top stack symbol;
}

```



# Nonrec. Predictive Parsing (Example)

| Non-terminal | Input Symbol                |                           |                       |                     |                           |                           |
|--------------|-----------------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
|              | id                          | +                         | *                     | (                   | )                         | \$                        |
| $E$          | $E \rightarrow TE'$         |                           |                       | $E \rightarrow TE'$ |                           |                           |
| $E'$         |                             | $E' \rightarrow +TE'$     |                       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$          | $T \rightarrow FT'$         |                           |                       | $T \rightarrow FT'$ |                           |                           |
| $T'$         |                             | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$          | $F \rightarrow \mathbf{id}$ |                           |                       | $F \rightarrow (E)$ |                           |                           |

| Matched | Stack | Input                      | Action |
|---------|-------|----------------------------|--------|
|         | $E\$$ | $\mathbf{id + id * id \$}$ | ...    |
| ...     | ...   | ...                        | ...    |

# Nonrec. Predictive Parsing (Example)

| Non-terminal | Input Symbol                |                           |                       |                     |                           |                           |
|--------------|-----------------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
|              | id                          | +                         | *                     | (                   | )                         | \$                        |
| $E$          | $E \rightarrow TE'$         |                           |                       | $E \rightarrow TE'$ |                           |                           |
| $E'$         |                             | $E' \rightarrow +TE'$     |                       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$          | $T \rightarrow FT'$         |                           |                       | $T \rightarrow FT'$ |                           |                           |
| $T'$         |                             | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$          | $F \rightarrow \mathbf{id}$ |                           |                       | $F \rightarrow (E)$ |                           |                           |

| Matched    | Stack              | Input                  | Action                             |
|------------|--------------------|------------------------|------------------------------------|
|            | $E\$$              | <b>id + id * id \$</b> | output $E \rightarrow TE'$         |
|            | $TE'\$$            | <b>id + id * id \$</b> | output $T \rightarrow FT'$         |
|            | $FT'E'\$$          | <b>id + id * id \$</b> | output $F \rightarrow \mathbf{id}$ |
|            | <b>id</b> $T'E'\$$ | <b>id + id * id \$</b> | match <b>id</b>                    |
| <b>id</b>  | $T'E'\$$           | <b>+ id * id \$</b>    | output $T' \rightarrow \epsilon$   |
| <b>id</b>  | $E'\$$             | <b>+ id * id \$</b>    | output $E' \rightarrow +TE'$       |
| <b>id</b>  | $+TE'\$$           | <b>+ id * id \$</b>    | match <b>+</b>                     |
| <b>id+</b> | $TE'\$$            | <b>id * id \$</b>      | output $T \rightarrow FT'$         |
| ...        | ...                | ...                    | ...                                |

Note shift up of last column

## 4.1.3 Syntax Error Handling

- Good compiler should assist in identifying and locating errors
  - **Lexical errors:** compiler can easily detect and continue
  - **Syntax errors:** compiler can detect and often recover
  - **Semantic errors:** compiler can sometimes detect
  - **Logical errors:** hard to detect
- Three goals. The error handler should
  - Report errors clearly and accurately
  - Recover quickly to detect subsequent errors
  - Add minimal overhead to processing of correct programs

# Error Detection and Reporting

- **Viable-prefix property** of LL/LR parsers allow detection of syntax errors as soon as possible, i.e., as soon as prefix of input does not match prefix of any string in language (valid program)
- Reporting an error:
  - At least report line number and position
  - Print diagnostic message, e.g.,  
“semicolon missing at this position”

# Error-Recovery Strategies

- Continue after error detection, restore to state where processing may continue, but...
- No universally acceptable strategy, but some useful strategies:
  - **Panic-mode recovery**: discard input until token in designated set of *synchronizing* tokens is found
  - **Phrase-level recovery**: perform local correction on the input to repair error, e.g., insert missing semicolon  
Has actually been used
  - **Error productions**: augment grammar with productions for erroneous constructs
  - **Global correction**: choose minimal sequence of changes to obtain correct string  
Costly, but yardstick for evaluating other strategies

## 4.4.5 Error Recovery in Pred. Parsing

### Panic-mode recovery

- Discard input until token in set of designated synchronizing tokens is found
- Heuristics
  - Put all symbols in  $FOLLOW(A)$  into synchronizing set for  $A$  (and remove  $A$  from stack)
  - Add symbols based on hierarchical structure of language constructs
  - Add symbols in  $FIRST(A)$
  - If  $A \xRightarrow{*} \epsilon$ , use production deriving  $\epsilon$  as default
  - Add tokens to synchronizing sets of all other tokens

# Adding Synchronizing Tokens

| nonterminal $A$ | FIRST( $A$ )     | FOLLOW( $A$ ) |
|-----------------|------------------|---------------|
| $E$             | {(, <b>id</b> }  | {), \$}       |
| $E'$            | {+, $\epsilon$ } | {), \$}       |
| $T$             | {(, <b>id</b> }  | {+, ), \$}    |
| $T'$            | {*, $\epsilon$ } | {+, ), \$}    |
| $F$             | {(, <b>id</b> }  | {*, +, ), \$} |

| Non-terminal | Input Symbol                |                           |                       |                     |                           |                           |
|--------------|-----------------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
|              | <b>id</b>                   | +                         | *                     | (                   | )                         | \$                        |
| $E$          | $E \rightarrow TE'$         |                           |                       | $E \rightarrow TE'$ | synch                     | synch                     |
| $E'$         |                             | $E' \rightarrow +TE'$     |                       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$          | $T \rightarrow FT'$         | synch                     |                       | $T \rightarrow FT'$ | synch                     | synch                     |
| $T'$         |                             | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$          | $F \rightarrow \mathbf{id}$ | synch                     | synch                 | $F \rightarrow (E)$ | synch                     | synch                     |



# Adding Synchronizing Tokens

| Non-terminal | Input Symbol                |                           |                       |                     |                           |                           |
|--------------|-----------------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
|              | id                          | +                         | *                     | (                   | )                         | \$                        |
| $E$          | $E \rightarrow TE'$         |                           |                       | $E \rightarrow TE'$ | synch                     | synch                     |
| $E'$         |                             | $E' \rightarrow +TE'$     |                       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$          | $T \rightarrow FT'$         | synch                     |                       | $T \rightarrow FT'$ | synch                     | synch                     |
| $T'$         |                             | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$          | $F \rightarrow \mathbf{id}$ | synch                     | synch                 | $F \rightarrow (E)$ | synch                     | synch                     |

Parsing **+id\*)id**:

| Matched | Stack | Input             | Action |
|---------|-------|-------------------|--------|
|         | $E\$$ | <b>+id*)id</b> \$ | ...    |
| ...     | ...   | ...               | ...    |

# Adding Synchronizing Tokens

Parsing **+id\*) + id**:

| Matched    | Stack              | Input                | Action   |
|------------|--------------------|----------------------|--|
|            | $E\$$              | <b>+id*) + id</b> \$ | error, skip +  |
|            | $E\$$              | <b>id*) + id</b> \$  | <b>id</b> $\in$ FIRST( $E$ ), output $E \rightarrow TE'$ |
|            | $TE'\$$            | <b>id*) + id</b> \$  | output $T \rightarrow FT'$                               |
|            | $FT'E'\$$          | <b>id*) + id</b> \$  | output $F \rightarrow \mathbf{id}$                       |
|            | <b>id</b> $T'E'\$$ | <b>id*) + id</b> \$  | match <b>id</b>  |
| <b>id</b>  | $T'E'\$$           | <b>*) + id</b> \$    | output $T' \rightarrow *FT'$                             |
| <b>id</b>  | <b>*</b> $FT'E'\$$ | <b>*) + id</b> \$    | match <b>*</b>   |
| <b>id*</b> | $FT'E'\$$          | <b>) + id</b> \$     | error, synch   |
| <b>id*</b> | $T'E'\$$           | <b>+ id</b> \$       | output $T' \rightarrow \epsilon$                         |
| ...        | ...                | ...                  | ...  |

# Error Recovery in Predictive Parsing

## Phrase-level recovery

- Local correction on remaining input that allows parser to continue
- Pointer to error routines in blank table entries
  - Change symbols
  - Insert symbols
  - Delete symbols
  - Print appropriate message
- Make sure that we do not enter infinite loop

# Predictive Parsing Issues

- What to do in case of multiply-defined entries?
  - Transform grammar
    - \* Left-recursion elimination
    - \* Left factoring
  - Not always applicable
- Designing grammar suitable for top-down parsing is hard
  - Left-recursion elimination and left factoring make grammar hard to read and to use in translation

Therefore: try to use LR parser generators

# Compilerconstructie

college 3

Syntax Analysis (1)

Chapters for reading: 2.4, 4.1–4.4

Next week: also werkcollege