Compilerconstructie

najaar 2018

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

Rudy van Vliet
kamer 140 Snellius, tel. 071-527 2876
rvvliet(at)liacs(dot)nl

college 3, vrijdag 21 september 2018

\+ werkcollege

Syntax Analysis (1)

LKP

https://defles.ch/lkp

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

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

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

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

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

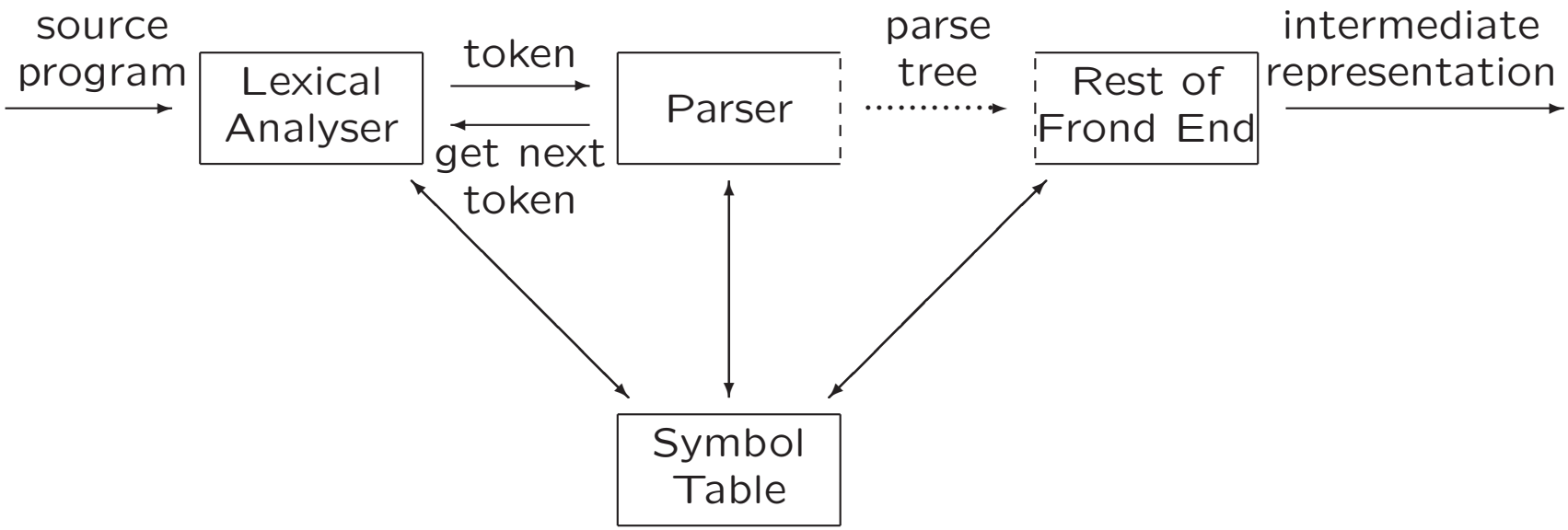
Such checks are performed during semantic-analysis phase

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:

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

with productions P:

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

4.2.2 Notational Conventions

1. Terminals:
   $a, b, c, \ldots$; specific terminals: $+, *, (,), 0, 1,$ **id**, **if**, $\ldots$
2. Nonterminals:
   $A, B, C, \ldots$; specific nonterminals: S, *expr*, *stmt*, $\ldots, E, \ldots$
3. Grammar symbols: X, Y, Z
4. Strings of terminals: u, v, w, x, y, z
5. Strings of grammar symbols: $\alpha, \beta, \gamma, \ldots$
   Hence, generic production: $A \to \alpha$
6. A-productions:
   $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k \qquad \Rightarrow \qquad A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$
   Alternatives for A
7. By default, head of first production is start symbol

Notational Conventions (Example)

CFG for simple arithmetic expressions:

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

with productions P:

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

Can be rewritten concisely as:

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

4.2.3 Derivations

Example grammar:

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

- 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 $A \to \gamma$ is production in grammar
- Derivation in zero or more steps: $\overset{*}{\Rightarrow}$
- Derivation in one or more steps: $\overset{+}{\Rightarrow}$

Derivations

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

Example of leftmost derivation:

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

Parse Tree

(from lecture 1)

(derivation tree in FI2)

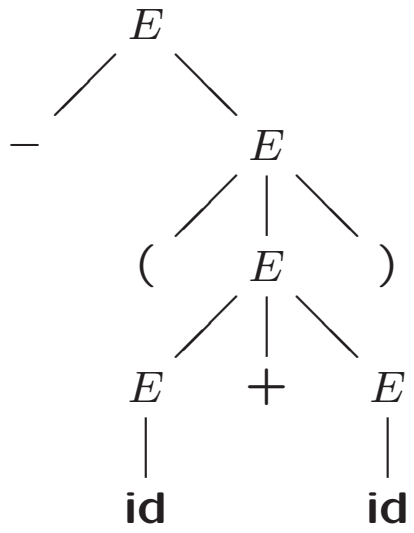
- The root of the tree is labelled by the start symbol
- Each leaf of the tree is labelled by a terminal (=token) or ϵ (=empty)
- Each interior node is labelled by a nonterminal
- If node A has children $X_1, X_2, \ldots, X_n$, then there must be a production $A \to X_1 X_2 \ldots X_n$

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

4.2.4 Parse Trees and Derivations

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

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



(E)

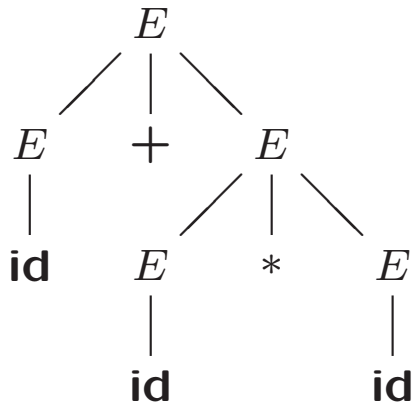
Many-to-one relationship between derivations and parse trees. . .

4.2.5 Ambiguity

More than one leftmost/rightmost derivation for same sentence

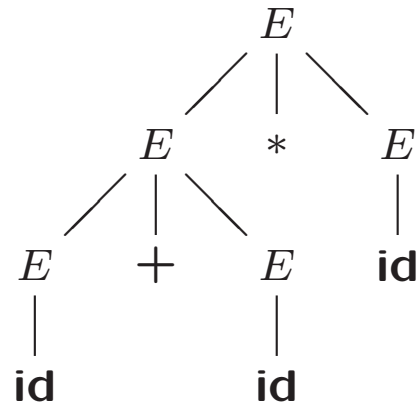
Example: $a + b * c$

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



$a + (b * c)$

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



$(a + b) * c$

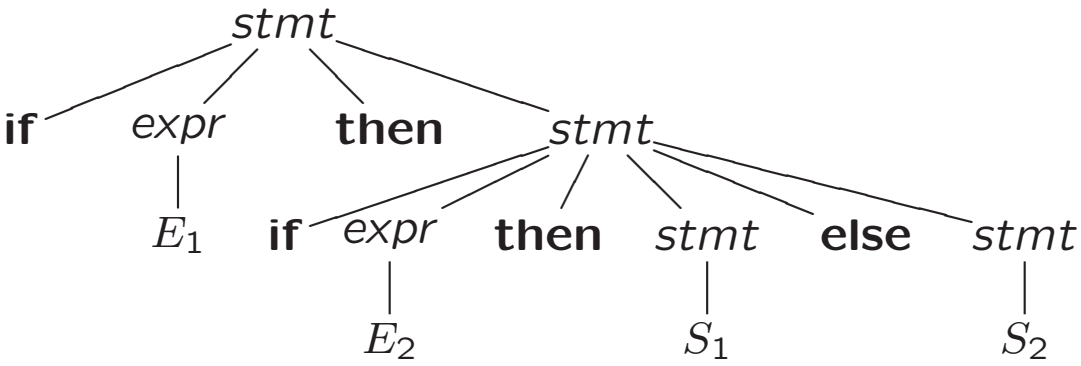
4.3.2 Eliminating ambiguity

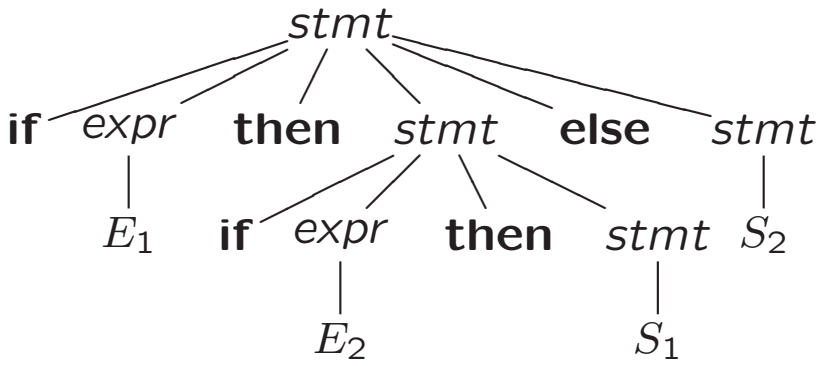
- Sometimes ambiguity can be eliminated
- Example: "dangling-else"-grammar

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
& | & \textbf{other}
\end{array}
$$

Here, **other** is any other statement

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





Preferred...

Eliminating ambiguity

Example: ambiguous "dangling-else"-grammar

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
& | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
& | & \textbf{other}
\end{array}
$$

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

Eliminating ambiguity

Example: ambiguous "dangling-else"-grammar

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
& | & \textbf{other}
\end{array}
$$

Equivalent unambiguous grammar

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matchedstmt} \\
& | & \textit{openstmt} \\
\textit{matchedstmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matchedstmt} \textbf{ else } \textit{matchedstmt} \\
& | & \textbf{other} \\
\textit{openstmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{matchedstmt} \textbf{ else } \textit{openstmt}
\end{array}
$$

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)

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{expr}\ ; \\
& | & \textbf{if}\ (\textbf{expr})\ \textit{stmt} \\
& | & \textbf{for}\ (\textit{optexpr}\ ;\textit{optexpr}\ ;\textit{optexpr})\ \textit{stmt} \\
& | & \textbf{other} \\
\textit{optexpr} & \rightarrow & \epsilon \\
& | & \textbf{expr}
\end{array}
$$

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 → X1X2...Xk;
2)   for (i = 1 to k)
3)   { if (Xi is nonterminal)
4)       call procedure Xi();
5)     else if (Xi equals current input symbol a)
6)             advance input to next symbol; /* match */
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:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{expr}\ ; \\
 & | & \textbf{if}\ (\textbf{expr})\ \textit{stmt} \\
 & | & \textbf{for}\ (\textit{optexpr}\ ;\textit{optexpr}\ ;\textit{optexpr})\ \textit{stmt} \\
 & | & \textbf{other}
\end{array}
$$

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 α be string of grammar symbols
- FIRST(α) = set of terminals/tokens that appear as first symbols of strings derived from α

Simple example:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{expr}\ ; \\
& | & \textbf{if}\ (\textbf{expr})\ \textit{stmt} \\
& | & \textbf{for}\ (\textit{optexpr}\ ;\textit{optexpr}\ ;\textit{optexpr})\ \textit{stmt} \\
& | & \textbf{other}
\end{array}
$$

Right-hand side may start with nonterminal. . .

Using FIRST (simple case)

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

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

  then FIRST(α) and FIRST(β) must be disjoint in order for predictive parsing to work

2.4.3 When to Use ϵ-Productions (simple solution)

Simple example:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{expr}\ ; \\
& | & \textbf{if}\ (\textbf{expr})\ \textit{stmt} \\
& | & \textbf{for}\ (\textit{optexpr}\ ;\textit{optexpr}\ ;\textit{optexpr})\ \textit{stmt} \\
& | & \textbf{other} \\
\textit{optexpr} & \rightarrow & \textbf{expr} \\
& | & \epsilon
\end{array}
$$

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");
}
```

4.4.2 FIRST (and Follow)

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

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

  FIRST(FT') = $\{(, \textbf{id}\}$
- When nonterminal has multiple productions, e.g.,

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

  and FIRST(α) and FIRST(β) are disjoint,
  we can choose between these A-productions by looking at next input symbol

Computing FIRST (Example)

$$
\begin{array}{lcll}
S & \rightarrow & Ab & | \quad c \\
A & \rightarrow & aS & | \quad \epsilon
\end{array}
$$

nonterminal X	FIRST(X)
S	...
A	...

Computing FIRST (Example)

$$
\begin{array}{lcll}
S & \rightarrow & Ab & \mid \quad c \\
A & \rightarrow & aS & \mid \quad \epsilon
\end{array}
$$

nonterminal X	FIRST(X)
S	$\{a, b, c\}$
A	$\{a, \epsilon\}$

Computing FIRST (Example)

$$
\begin{aligned}
S &\rightarrow ABb \mid c \\
A &\rightarrow aS \mid \epsilon \\
B &\rightarrow cA \mid \epsilon
\end{aligned}
$$

nonterminal X	FIRST(X)
S	...
A	...
B	...

Computing FIRST (Example)

$$
\begin{aligned}
S &\rightarrow ABb \mid c \\
A &\rightarrow aS \mid \epsilon \\
B &\rightarrow cA \mid \epsilon
\end{aligned}
$$

nonterminal X	FIRST(X)
S	$\{a, b, c\}$
A	$\{a, \epsilon\}$
B	$\{c, \epsilon\}$

Computing FIRST

Compute FIRST(X) for all grammar symbols X:

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

  $$X \to Y_1 Y_2 \ldots Y_k$$

  (see book) until all FIRST sets are stable

FIRST (Example)

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

nonterminal A	FIRST(A)
E	...
E'	...
T	...
T'	...
F	...

Fill in bottom-up...

FIRST (Example)

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

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

2.4.5 Left Recursion

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

$$\begin{aligned} E &\to E + T \mid T \\ T &\to \textbf{id} \end{aligned}$$

- $\mathsf{FIRST}(E + T) \cap \mathsf{FIRST}(T) = \{\textbf{id}\} \neq \emptyset$

- 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 $A \to A\alpha \mid \beta$
- Can be eliminated by replacing the productions by

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

- Procedure:

  1. Group A-productions as

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

  2. Replace A-productions by

$$\begin{array}{lcl} A & \to & \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' & \to & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

Elimination of Left Recursion

Immediate left recursion

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

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

Example:

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & \mathbf{id} \end{array}$$

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

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

$$S \Rightarrow Ba \Rightarrow AAa \Rightarrow SdAa$$ (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 ϵ-productions

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

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

4.3.4 Left Factoring

Another transformation to produce grammar suitable for predictive parsing

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

- Solution: left-factoring
  Replace two A-productions by

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

- $|\alpha|$ may be ≥ 2

Left Factoring (Example)

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

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr } \textbf{then } \textit{stmt} \\
& | & \textbf{if } \textit{expr } \textbf{then } \textit{stmt } \textbf{else } \textit{stmt} \\
& | & \textbf{other} \\
\textit{expr} & \rightarrow & b
\end{array}
$$

- Or abstract:

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

- 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

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

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)

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

- Non-left-recursive variant: . . .

Top-Down Parsing (Example)

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

- Non-left-recursive variant:

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

- 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 A be nonterminal

- FOLLOW(A) = set of terminals/tokens that can appear immediately to the right of A in sentential form:

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

- Example

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

Computing FOLLOW

Compute FOLLOW(A) for all nonterminals A:

- Place \$ in FOLLOW(S)

- For production $A \to \alpha B \beta$,
  add everything in FIRST(β) to FOLLOW(B) (except ϵ)

- – For production $A \to \alpha B$,
    add everything in FOLLOW(A) to FOLLOW(B)

  – For production $A \to \alpha B \beta$ with $\epsilon \in$ FIRST(β),
    add everything in FOLLOW(A) to FOLLOW(B)

  until all FOLLOW sets are stable

FIRST and FOLLOW (Example)

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

nonterminal A	FIRST(A)	FOLLOW(A)
E	$\{(, \textbf{id}\}$	$\ldots$
E'	$\{+, \epsilon\}$	$\ldots$
T	$\{(, \textbf{id}\}$	$\ldots$
T'	$\{*, \epsilon\}$	$\ldots$
F	$\{(, \textbf{id}\}$	$\ldots$

Fill in top-down...

FIRST and FOLLOW (Example)

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

nonterminal A	FIRST(A)	FOLLOW(A)
E	$\{(, \textbf{id}\}$	$\{), \$\}$
E'	$\{+, \epsilon\}$	$\{), \$\}$
T	$\{(, \textbf{id}\}$	$\{+,), \$\}$
T'	$\{*, \epsilon\}$	$\{+,), \$\}$
F	$\{(, \textbf{id}\}$	$\{*, +,), \$\}$

4.4.3 LL(1) Grammars

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

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

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

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

Top-Down Parsing Table (Example)

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

nonterminal A	FIRST(A)	FOLLOW(A)
E	$\{(, \textbf{id}\}$	$\{), \$\}$
E'	$\{+, \epsilon\}$	$\{), \$\}$
T	$\{(, \textbf{id}\}$	$\{+,), \$\}$
T'	$\{*, \epsilon\}$	$\{+,), \$\}$
F	$\{(, \textbf{id}\}$	$\{*, +,), \$\}$

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Top-Down Parsing Table (Example)

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

nonterminal A	FIRST(A)	FOLLOW(A)
E	$\{(, \textbf{id}\}$	$\{), \$\}$
E'	$\{+, \epsilon\}$	$\{), \$\}$
T	$\{(, \textbf{id}\}$	$\{+,), \$\}$
T'	$\{*, \epsilon\}$	$\{+,), \$\}$
F	$\{(, \textbf{id}\}$	$\{*, +,), \$\}$

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	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

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 $A \to \alpha \mid \beta$,
  - α and β do not both derive strings beginning with same terminal a
  - at most one of α and β can derive ϵ
  - if $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α 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):

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

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

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

Left Factoring (Example)

- Abstract if-then-else-grammar:

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

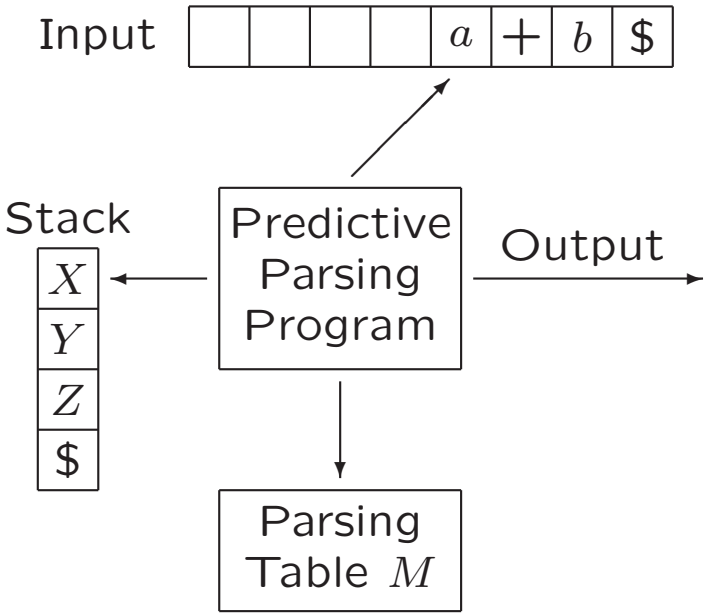
- Left-factored:

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

  Not LL(1)...

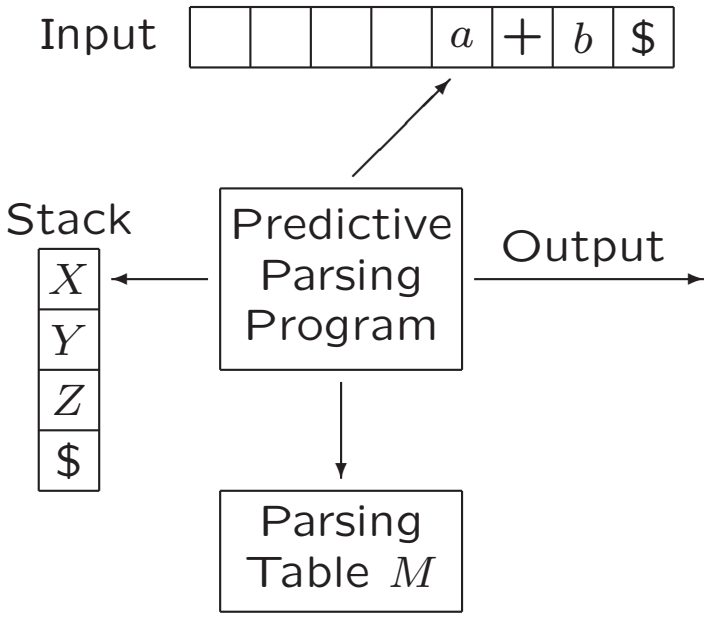
4.4.4 Nonrecursive Predictive Parsing

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 ($M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$)
              { output production $X \rightarrow Y_1 Y_2 \ldots Y_k$;
                pop stack;
                push $Y_k, Y_{k-1}, \ldots, Y_1$ onto stack, with Y_1 on top;
              }
  let X be top stack symbol;
}



Nonrec. Predictive Parsing (Example)

Non-terminal	Input Symbol					
	id	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Matched	Stack	Input	Action
	E\$	**id** $+$ **id** $*$ **id** \$	...
...	...	...	...

Nonrec. Predictive Parsing (Example)

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Matched	Stack	Input	Action
	$E\$$	**id** + **id** * **id** $	output $E \to TE'$
	$TE'\$$	**id** + **id** * **id** $	output $T \to FT'$
	$FT'E'\$$	**id** + **id** * **id** $	output $F \to$ **id**
	id$T'E'\$$	**id** + **id** * **id** $	match **id**
id	$T'E'\$$	+ **id** * **id** $	output $T' \to \epsilon$
id	$E'\$$	+ **id** * **id** $	output $E' \to +TE'$
id	$+TE'\$$	+ **id** * **id** $	match +
id+	$TE'\$$	**id** * **id** $	output $T \to FT'$
...	...	...	...

Note shift up of last column

The next eight slides (on error handling) have not been discussed in class. Therefore, the topic does not have to be known for the exam.

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"

4.1.4 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 $A \stackrel{*}{\Rightarrow} \epsilon$, use production deriving ϵ as default
  - Add tokens to synchronizing sets of all other tokens

Adding Synchronizing Tokens

nonterminal A	FIRST(A)	FOLLOW(A)
E	$\{(, \textbf{id}\}$	$\{), \$\}$
E'	$\{+, \epsilon\}$	$\{), \$\}$
T	$\{(, \textbf{id}\}$	$\{+,), \$\}$
T'	$\{*, \epsilon\}$	$\{+,), \$\}$
F	$\{(, \textbf{id}\}$	$\{*, +,), \$\}$

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

Adding Synchronizing Tokens

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

Parsing ( ) + (**id** (* **id** :

Matched	Stack	Input	Action
	E\$	( ) + (**id** (* **id** \$	...
...	...	...	...

Adding Synchronizing Tokens

Parsing ( ) + (**id** ($*$ **id** :

Matched	Stack	Input	Action
	$E\$$	$()+(\mathbf{id}(*\mathbf{id}\$$	($\in$ FIRST(TE'), output $E \to TE'$
...	...	...	...
	$(E)T'E'\$$	$()+(\mathbf{id}(*\mathbf{id}\$$	match (
(	$E)T'E'\$$	$)+(\mathbf{id}(*\mathbf{id}\$$	error, synch
$(\underline{E}$	$)T'E'\$$	$)+(\mathbf{id}(*\mathbf{id}\$$	match)
...	...	...	...
$(\underline{E})+($	$\mathbf{id}T'E')T'E'\$$	$\mathbf{id}(*\mathbf{id}\$$	match **id**
$(\underline{E})+(\mathbf{id}$	$T'E')T'E'\$$	$(*\mathbf{id}\$$	error, skip (
$(\underline{E})+(\mathbf{id}$	$T'E')T'E'\$$	$*\mathbf{id}\$$	$*$ $\in$ FIRST($*FT'$), output $T' \to *FT'$
...	...	...	...
$(\underline{E})+(\mathbf{id}*\mathbf{id}$	$E')T'E'\$$	$\$$	$ $\in$ FOLLOW(E'), output $E' \to \epsilon$
$(\underline{E})+(\mathbf{id}*\mathbf{id}$	$)T'E'\$$	$\$$	error, pop)
$(\underline{E})+(\mathbf{id}*\mathbf{id})$	$T'E'\$$	$\$$	$ $\in$ FOLLOW(T'), output $T' \to \epsilon$
$(\underline{E})+(\mathbf{id}*\mathbf{id}\underline{)}$	$E'\$$	$\$$	$ $\in$ FOLLOW(E'), output $E' \to \epsilon$
$(\underline{E})+(\mathbf{id}*\mathbf{id}\underline{)}$	$\$$	$\$$	

Underlined nonterminal in column 'Matched' indicates that it has been popped from stack by synch-action

Underlined terminal indicates that it has been inserted into input

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