Compilerconstructie

najaar 2013

http://www.liacs.nl/home/rvvliet/coco/

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs(dot)nl

college 3, dinsdag 17 september 2013

Syntax Analysis (1)

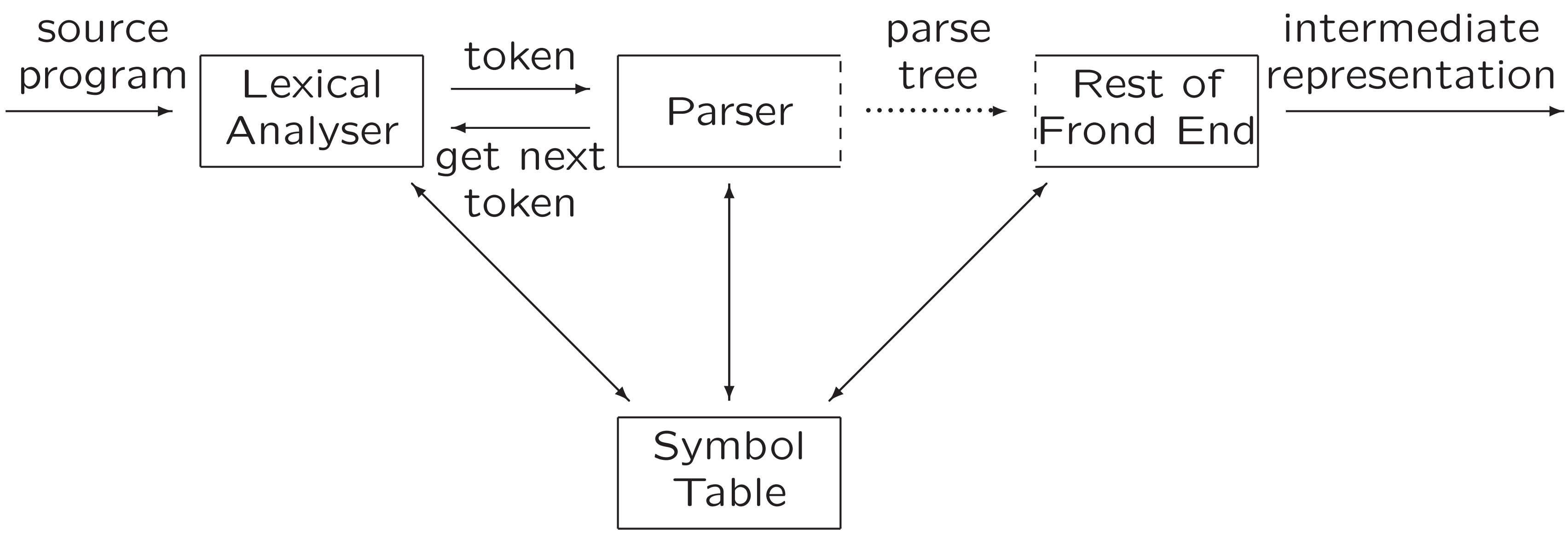
4 Syntax Analysis

- 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

- Syntax of programming-language constructs can be described by CFG
  - 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

- Syntax analyis is performed by parser

4.1 Parser's Position in a Compiler



- 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:

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

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:
   $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k \quad \Rightarrow \quad A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$
   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:

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

Derivations

Example grammar:

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

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

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

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

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 $L(G) = \{w \mid w$ is sentence of $G\}$
- Leftmost derivation: $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$
- If $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$, then α is left sentential form of G
- Rightmost derivation: $\gamma Aw \underset{rm}{\Rightarrow} \gamma\delta w$, $\quad \underset{rm}{\overset{*}{\Rightarrow}}$

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 college 1)

(derivation tree in FI2)

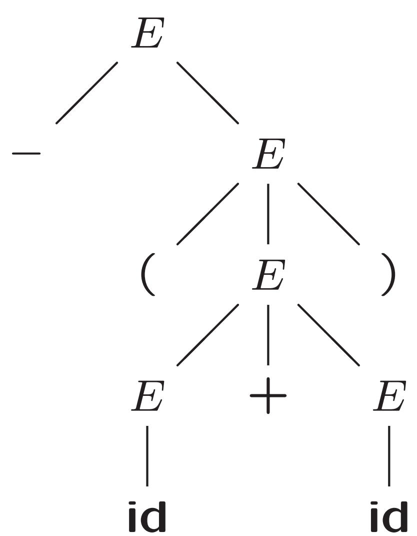
- The root of the tree is labelled by the start symbol
- Each leaf of the tree is labelled by a terminal (=token) or ϵ (=empty)
- Each interior node is labelled by a nonterminal
- If node A has children $X_1, X_2, \ldots, X_n$, then there must be a production $A \to X_1 X_2 \ldots X_n$

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

Parse Trees and Derivations

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

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



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

4.3.1 Why Regular Expressions For Lexical Syntax?

- Convenient way to modularize front end
  $\approx$ simplifies design

- Regular expressions powerful enough for lexical syntax

- Regular expressions easier to understand than grammars

- More efficient lexical analysers can be constructed automatically from regular expressions than from arbitrary grammars

Ambiguity

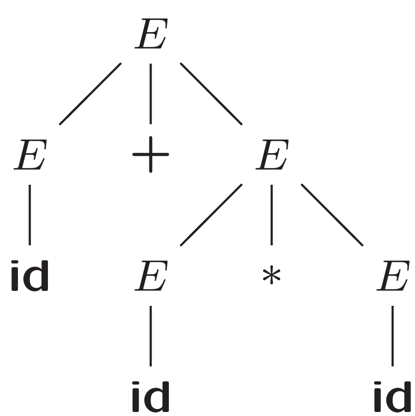
More than one leftmost/rightmost derivation for same sentence

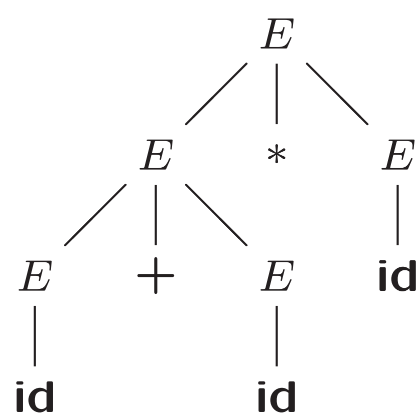
Example: $a + b * c$

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

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

$a + (b * c)$





$(a + b) * c$

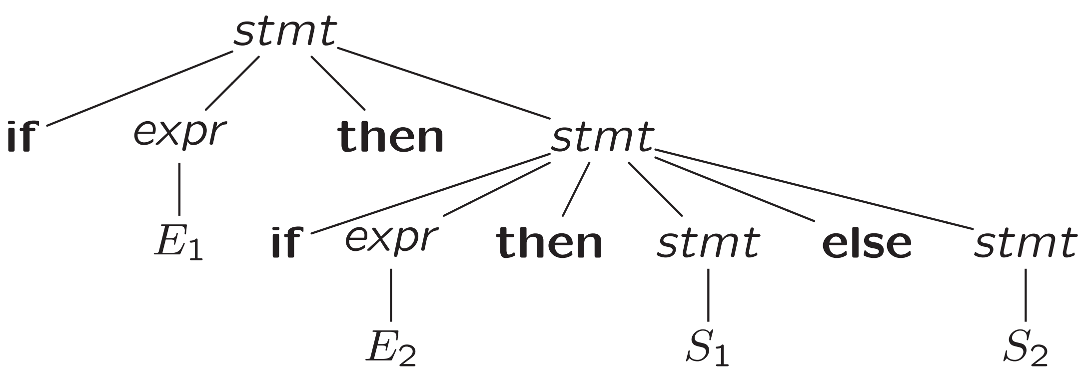
Eliminating ambiguity

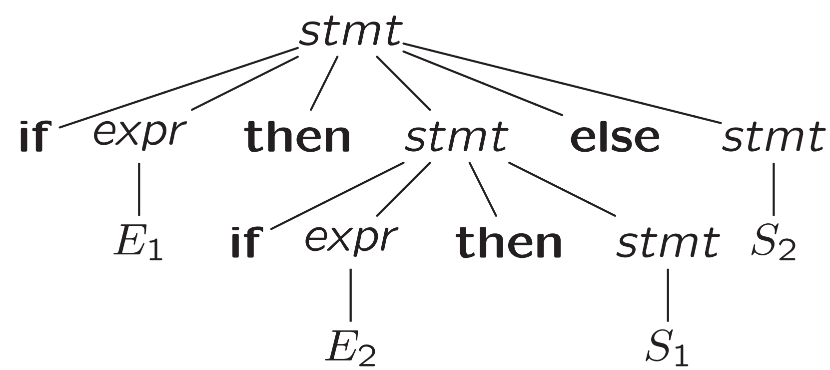
- Sometimes ambiguity can be eliminated
- Example: "dangling-else"-grammar

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

Here, **other** is any other statement

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





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

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

Equivalent unambiguous grammar

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

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 Parsing (Top-Down Example)

from college 1

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

How to determine parse tree for

$$\textbf{for}\ (;\textbf{expr}\ ;\textbf{expr}\ )\textbf{other}$$

Use lookahead: current terminal in input

Predictive Parsing

from college 1

- 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 . . .

Recursive Descent Parsing

Recursive procedure for each nonterminal

```
   void A()
1) { Choose an A-production, A → X1 X2 ... 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;
7)             else /* an error has occurred */;
     }
   }
```

Pseudocode is nondeterministic

Recursive Descent

- 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

Predictive Parsing

from college 1

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

Simple example:

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

Predictive Parsing (Example)

from college 1

```
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

from college 1

- Let α be string of grammar symbols
- FIRST(α) is the set of terminals that appear as first symbols of strings generated 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

from college 1

- Let α be string of grammar symbols
- FIRST(α) is the set of terminals that appear as first symbols of strings generated from α
- When a nontermimal 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

Left Recursion

- Productions of the form $A \to A\alpha \mid \beta$ are left-recursive
  - β does not start with A
  - Example: $E \to E + T \mid T$
- Top-down parser may loop forever if grammar has left-recursive productions
- Left-recursive productions can be eliminated by rewriting productions

Left Recursion Elimination

Immediate left recursion

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

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

- 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

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

Left Recursion Elimination

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)

General Left Recursion Elimination

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

- 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

General Left Recursion Elimination

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) $\quad\{$ replace each production of form $A_i \rightarrow A_j\gamma$
by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, where
$A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_j-productions
5) $\quad\}$
6) $\quad$eliminate immediate left recursion among A_i-productions
7) $\}$

Example with $A \rightarrow \epsilon$

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 α
  How to expand A? To $\alpha\beta_1$ or to $\alpha\beta_2$?

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

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

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)

What is result of left factoring for

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

Non-Context-Free 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

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

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

Such checks are performed during semantic-analysis phase

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)

-

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

- 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 $\textbf{id} + \textbf{id} * \textbf{id}$ . . .
- 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)

FIRST

- Let α be string of grammar symbols
- FIRST(α) = set of terminals/tokens which begin 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

Compute FIRST(X) for all grammar symbols X:

- If X is terminal, then FIRST(X) $= \{X\}$

- If $X \rightarrow \epsilon$ is production, then add ϵ to FIRST(X)

- Repeat adding symbols to FIRST(X) by looking at productions

  $$X \rightarrow Y_1 Y_2 \ldots Y_k$$

  (see book) until all FIRST sets are stable

FIRST (Example)

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

$$
\begin{aligned}
\text{FIRST}(E) &= \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\} \\
\text{FIRST}(E') &= \{+, \epsilon\} \\
\text{FIRST}(T') &= \{*, \epsilon\}
\end{aligned}
$$

FOLLOW

- Let A be nonterminal

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

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

- Compute FOLLOW(A) for all nonterminals A
  See book

FIRST and FOLLOW (Example)

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

$$
\begin{aligned}
\text{FIRST}(E) &= \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\} \\
\text{FIRST}(E') &= \{+, \epsilon\} \\
\text{FIRST}(T') &= \{*, \epsilon\} \\
\text{FOLLOW}(E) &= \text{FOLLOW}(E') = \{), \$\} \\
\text{FOLLOW}(T) &= \text{FOLLOW}(T') = \{+,), \$\} \\
\text{FOLLOW}(F) &= \{*, +,), \$\}
\end{aligned}
$$

Parsing Tables

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 $b \in \text{FOLLOW}(A)$)
      add $A \to \alpha$ to $M[A, b]$;
  }
}
If $M[A, a]$ is empty, set $M[A, a]$ to **error**.

Top-Down Parsing Table (Example)

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

$$
\begin{aligned}
\text{FIRST}(E) &= \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\} \\
\text{FIRST}(E') &= \{+, \epsilon\} \\
\text{FIRST}(T') &= \{*, \epsilon\} \\
\text{FOLLOW}(E) &= \text{FOLLOW}(E') = \{), \$\} \\
\text{FOLLOW}(T) &= \text{FOLLOW}(T') = \{+,), \$\} \\
\text{FOLLOW}(F) &= \{*, +,), \$\}
\end{aligned}
$$

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	$F \rightarrow$ **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):

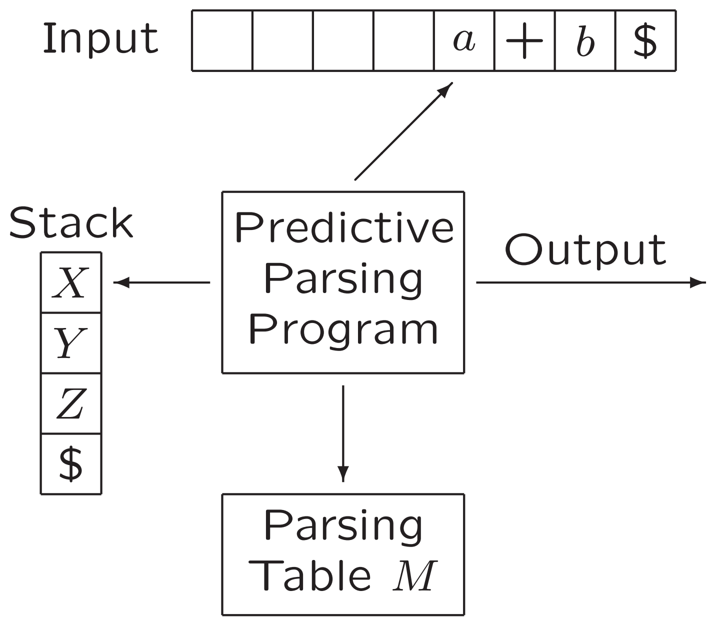
$$
\begin{array}{rcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

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

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

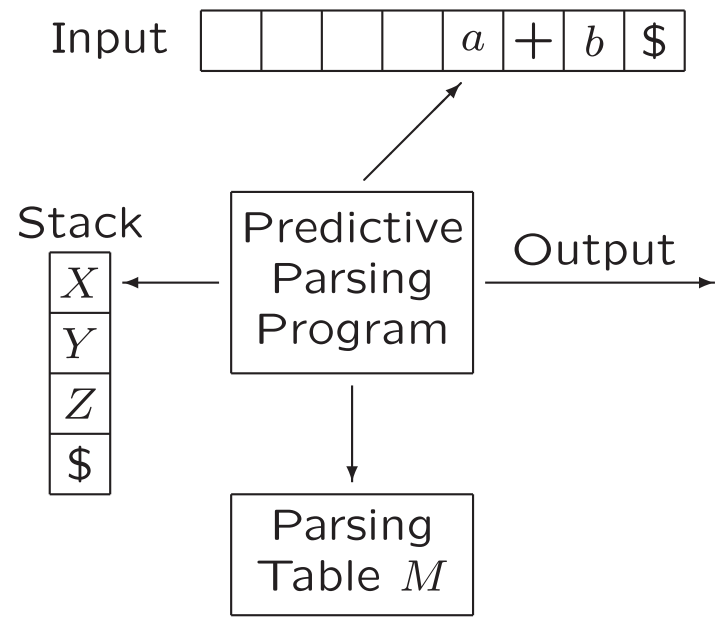
Nonrecursive Predictive Parsing

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 ≠ $) /* 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 → Y1Y2...Yk)
                      { output production X → Y1Y2...Yk;
                        pop stack;
                        push Yk, Yk−1, ..., Y1 onto stack, with Y1 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** \$	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

Error Recovery in Predictive 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

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 automatic parser generators

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

Compiler constructie

college 3

Syntax Analysis (1)

Chapters for reading: 4.1–4.4

Next week: also werkcollege