COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
`http://www.cse.unsw.edu.au/~cs3131`
`http://www.cse.unsw.edu.au/~cs9102`

Week 2 (2nd Lecture): Top-Down Parsing: Recursive-Descent

Write a predictive (or non-backtracking) top-down parser

Grammar G Assignment 2!

Eliminating left recursion & common prefixes

The Transformed Grammar G'

Constructing First, Follow and Select Sets for G'

A Recursive-Descent Parser

The LL(1) Parsing Table

The LL(1) Table-Driving Parser

Lookahead Token(s)

- **Lookahead Token(s)**: The currently scanned token(s) in the input.
- In Recogniser.java, currentToken represents the lookahead token
- For most programming languages, one token lookahead only.
- Initially, the lookahead token is the leftmost token in the input.

currentChar is the lookahead character used in the scanner.

Top-Down Parsing

- Build the parse tree starting with the start symbol (i.e., the root) towards the sentence being analysed (i.e., leaves).
- Use one token of lookahead, in general
- Discover the leftmost derivation

  I.e, the productions used in expanding the parse tree represent a leftmost derivation

Predictive (Non-Backtracking) Top-Down Parsing

- To expand a nonterminal, the parser always predict (choose) the right alternative for the nonterminal by looking at the lookahead symbol only.
- Flow-of-control constructs, with their distinguishing keywords, are detectable this way, e.g., in the VC grammar:

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \langle \text{compound-stmt} \rangle \\
& | & \textbf{if} \text{ ”(” } \langle \text{expr} \rangle \text{ ”)” } (\textbf{ELSE } \langle \text{stmt} \rangle)? \\
& | & \textbf{break} \text{ ”;”} \\
& | & \textbf{continue} \text{ ”;”} \\
& \ldots &
\end{array}
$$

- Prediction happens before the actual match begins.

Which of the Two Alternatives on S to Choose?

- Grammar:

$$\begin{aligned} S &\to aA \mid bB \\ A &\to \cdots \\ B &\to \cdots \end{aligned}$$

- Sentence: $a \cdots$
- The leftmost derivation:

$$\begin{aligned} S &\Longrightarrow_{\mathrm{lm}} aA \\ &\Longrightarrow_{\mathrm{lm}} \cdots \end{aligned}$$

Select the first alternative aA

Which of the Two Alternatives on S to Choose?

- Grammar:

$$
\begin{aligned}
S &\to Ab \mid Bc \\
A &\to Df \mid CA \\
B &\to gA \mid e \\
C &\to dC \mid c \\
D &\to h \mid i
\end{aligned}
$$

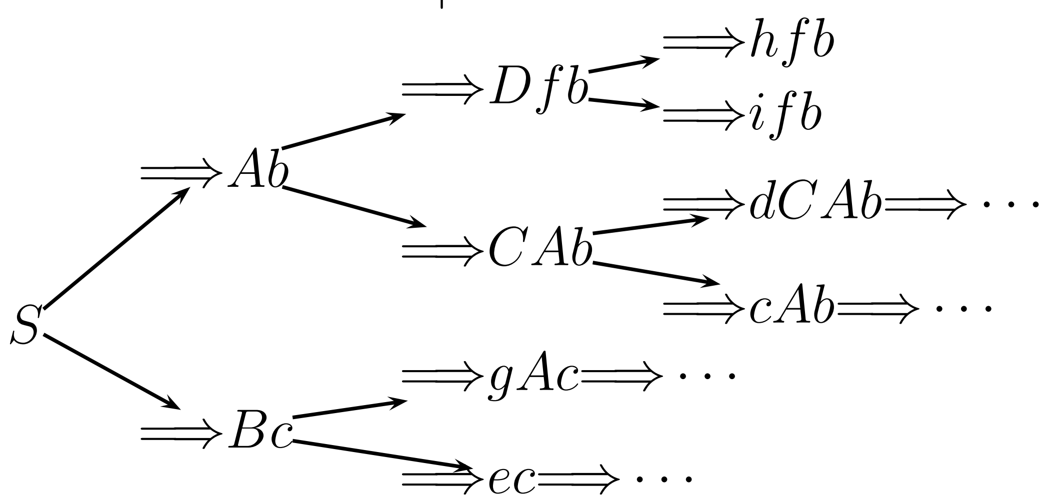
- Sentence: $gchfc$
- The leftmost derivation:

$$
\begin{array}{lllllll}
S & \Longrightarrow_{\text{lm}} & Bc & \Longrightarrow_{\text{lm}} & gAc & \Longrightarrow_{\text{lm}} & gCAc \\
 & \Longrightarrow_{\text{lm}} & gcAc & \Longrightarrow_{\text{lm}} & gcDfc & \Longrightarrow_{\text{lm}} & gchfc
\end{array}
$$

Intuition behind First Sets

- Grammar:

$$
\begin{aligned}
S &\to Ab \mid Bc \\
A &\to Df \mid CA \\
B &\to gA \mid e \\
C &\to dC \mid c \\
D &\to h \mid i
\end{aligned}
$$



- All possible leftmost derivations:

  First$(Ab) = \{c, d, h, i\}$ First$(Bc) = \{e, g\}$

Definition of First Sets

First(α):

- The set of all terminals that can begin any strings derived from α.
- if $\alpha \Longrightarrow^* \epsilon$, then ϵ is also in First(α)

Nullable Nonterminals

A nonterminal A is nullable if $A \Longrightarrow^* \epsilon$.

A Procedure to Compute $\mathsf{First}(\alpha)$

1. Case 1: α is a single symbol or ϵ:

   If α is a terminal a, then $\mathsf{First}(\alpha) = \mathsf{First}(a) = \{a\}$

   else if α is ϵ, then $\mathsf{First}(\alpha) = \mathsf{First}(\epsilon) = \{\epsilon\}$

   else if α is a nonterminal and $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots$ then

   $$\mathsf{First}(\alpha) = \cup_k \mathsf{First}(\beta_k)$$

2. Case 2: $\alpha = X_1 X_2 \cdots X_n$:

   If $X_1 X_2 \ldots X_i$ is nullable but X_{i+1} is not, then

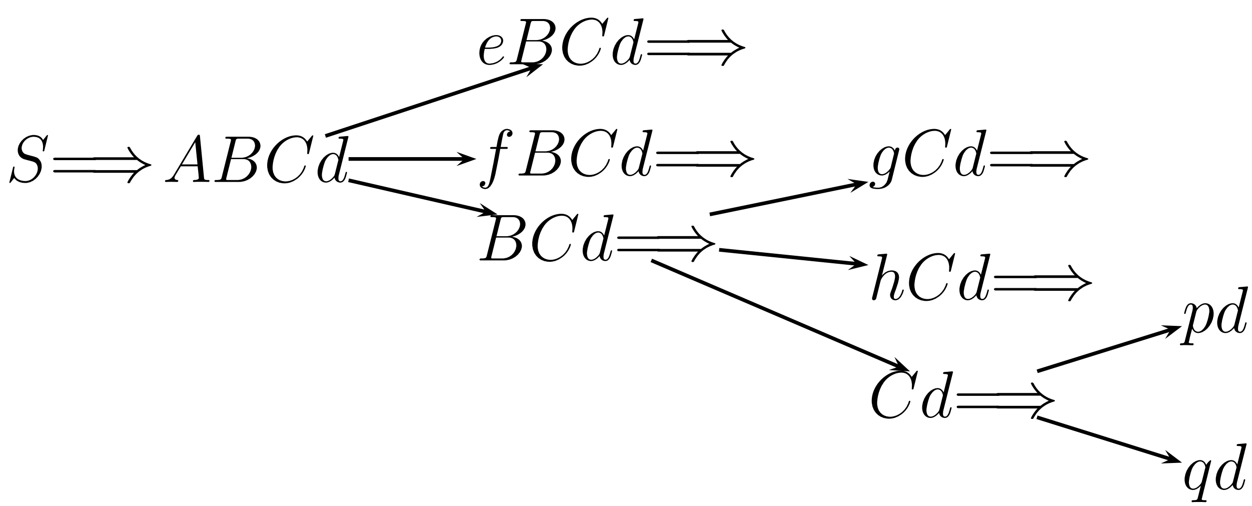
   $$\mathsf{First}(\alpha) = \mathsf{First}(X_1) \cup \mathsf{First}(X_2) \cup \cdots \cup \mathsf{First}(X_{i+1})$$

   Add ϵ to $\mathsf{First}(\alpha)$ if and only if α, i.e., $X_1 X_2 \cdots X_n$ is nullable

   (Note that $\mathsf{First}(X_1)$ must always be added to $\mathsf{First}(\alpha)$.)

Case 2 of the Procedure for Computing First

$$
\begin{aligned}
S &\to ABCd \\
A &\to e \mid f \mid \epsilon \\
B &\to g \mid h \mid \epsilon \\
C &\to p \mid q
\end{aligned}
$$



First$(ABCd) = \{e, f, g, h, p, q\}$

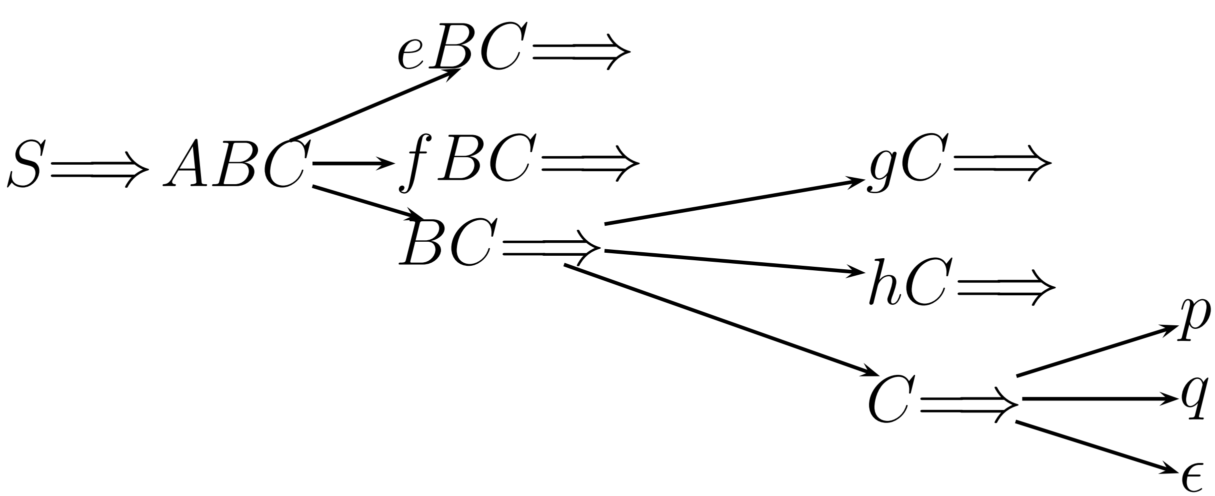
Case 2 of the Procedure for Computing First Again

$S \to ABC$ $\longleftarrow$ d deleted from the grammar in slide 189

$A \to e \mid f \mid \epsilon$

$B \to g \mid h \mid \epsilon$

$C \to p \mid q \mid \epsilon$ $\longleftarrow$ ϵ added to the grammar in slide 189



$\text{First}(ABC) = \{e, f, g, h, p, q, \epsilon\}$

The Expression Grammar

- The grammar with left recursion:

$$\text{Grammar 1:} \quad \begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T/F \mid F \\ F &\rightarrow \textbf{INT} \mid (E) \end{aligned}$$

- The transformed grammar without left recursion:

$$\text{Grammar 2:} \quad \begin{aligned} E &\rightarrow TQ \\ Q &\rightarrow +TQ \mid -TQ \mid \epsilon \\ T &\rightarrow FR \\ R &\rightarrow *FR \mid /FR \mid \epsilon \\ F &\rightarrow \textbf{INT} \mid (E) \end{aligned}$$

First Sets for Grammar 2 (without Recursion)

$$
\begin{array}{rcll}
\text{First}(E) & = & \text{First}(TQ) & = \{(, \textbf{INT}\} \\
\text{First}(T) & = & \text{First}(FR) & = \{(, \textbf{INT}\} \\
 & & \text{First}(Q) & = \{+, -, \epsilon\} \\
 & & \text{First}(R) & = \{*, /, \epsilon\} \\
 & & \text{First}(F) & = \{\textbf{INT}, (\} \\
 & & \text{First}(+TQ) & = \{+\} \\
 & & \text{First}(-TQ) & = \{-\} \\
 & & \text{First}(*FR) & = \{*\} \\
 & & \text{First}(/FR) & = \{/\} \\
 & & \text{First}((E)) & = \{(\} \\
 & & \text{First}(\textbf{INT}) & = \{\textbf{INT}\}
\end{array}
$$

Why Follow Sets?

- First sets do not tell us when to apply $A \rightarrow \alpha$ such that $\alpha \Longrightarrow^{*} \epsilon$ (the important special case is $A \rightarrow \epsilon$)
- Follow sets do
- Follow sets constructed only for nonterminals
- By convention, assume every input is terminated by a special end marker (i.e., the EOF marker), denoted $
- Follow sets do not contain ϵ

Definition of Follow Sets

Let A be a nonterminal. Define $\textsf{Follow}(A)$ to be the set of terminals that can appear immediately to the right of A in some sentential form. That is,

$$\textsf{Follow}(A) \;=\; \{a \mid S \Longrightarrow^* \cdots Aa \cdots\}$$

where S is the start symbol of the grammar.

A Procedure to Compute Follow Sets

1. If A is the start symbol, add \$ to $\mathsf{Follow}(A)$.
2. Look through the grammar for all occurrences of A on the right of productions. Let a typical production be:

$$B \rightarrow \alpha A \beta$$

   There are two cases – both may be applicable:

   (a) $\mathsf{Follow}(A)$ includes $\mathsf{First}(\beta) - \{\epsilon\}$.

   (b) If $\beta \Longrightarrow^* \epsilon$, then include $\mathsf{Follow}(B)$ in $\mathsf{Follow}(A)$.

Follow Sets for Grammar 2 (without Recursion)

$$\begin{array}{rcl}
\textsf{Follow}(E) & = & \{), \$\} \\
\textsf{Follow}(Q) & = & \{), \$\} \\
\textsf{Follow}(T) & = & \{+, -,), \$\} \\
\textsf{Follow}(R) & = & \{+, -,), \$\} \\
\textsf{Follow}(F) & = & \{+, -, *, /,), \$\}
\end{array}$$

Select Sets for Productions

- One Select set for every production in the grammar:
- The Select set for a production of the form $A \rightarrow \alpha$ is:
  - If $\epsilon \in \text{First}(\alpha)$, then

$$\text{Select}(A \rightarrow \alpha) = (\text{First}(\alpha) - \{\epsilon\}) \cup \text{Follow}(A)$$

  - Otherwise:

$$\text{Select}(A \rightarrow \alpha) = \text{First}(\alpha)$$

- The Select set predicts $A \rightarrow \alpha$ to be used in a derivation.
- Thus, the Select not needed if A has has one alternative

Select Sets for Grammar 2

$$
\begin{array}{lll}
\text{Select}(E \to TQ) & = \text{First}(TQ) & = \{(, \textbf{INT}\} \\
\text{Select}(Q \to +TQ) & = \text{First}(+TQ) & = \{+\} \\
\text{Select}(Q \to -TQ) & = \text{First}(-TQ) & = \{-\} \\
\text{Select}(Q \to \epsilon) & = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(Q) & = \{), \$\} \\
\text{Select}(T \to FR) & = \text{First}(FR) & = \{(, \textbf{INT}\} \\
\text{Select}(R \to *FR) & = \text{First}(+FR) & = \{*\} \\
\text{Select}(R \to /FR) & = \text{First}(/FR) & = \{/\} \\
\text{Select}(R \to \epsilon) & = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(R) & = \{+, -,), \$\} \\
\text{Select}(F \to \textbf{INT}) & = \text{First}(\textbf{INT}) & = \{\textbf{INT}\} \\
\text{Select}(F \to (E)) & = \text{First}((E)) & = \{(\}
\end{array}
$$

Lecture 4: Top-Down Parsing: Recursive-Descent

1. Write a predictive (or non-backtracking) top-down parser

Grammar G Assignment 2!

$\Downarrow$

Eliminating left recursion & common prefixes

$\Downarrow$

The Transformed Grammar G'

$\Downarrow$

Constructing First, Follow and Select Sets for G'

A Recursive-Descent Parser

The LL(1) Parsing Table

$\Downarrow$

The LL(1) Table-Driving Parser

Writing a Predictive Recursive-Descent Parser

- The variable currentToken is the lookahead token, which is initialised to the leftmost token in the program
- A method, called match, for matching the tokens at production right-hand sides

```
void match(int tokenExpected) {
  if (currentToken.kind == tokenExpected) {
    currentToken = scanner.getToken();
  } else {
    error: "tokenExpected" expected
                  but "currentToken" found
  }
}
```

- A method, called parseA, for every nonterminal A

parseA for $A \to \alpha_1 \mid \cdots \mid \alpha_n$

void parseA() {
  if (currentToken.kind in $\mathsf{Select}(A \to \alpha_1)$)
    parse α_1
  else if (currentToken.kind in $\mathsf{Select}(A \to \alpha_2)$)
    parse α_2
  $\ldots$
  else if (currentToken.kind in $\mathsf{Select}(A \to \alpha_n)$)
    parse α_n
  else
    syntacticError(...);
}

ParsingA for $A \rightarrow \alpha$ When A Has a Single Alternative

```
void parseA() {
   parse α
}
```

Coding parse α_i

- Suppose $\alpha_i = aABbC$, where A, B and C are nonterminals
- parse α_i implemented as:

```
match("a");
parseA();
parseB();
match("b");
parseC();
```

- If $\alpha_i = \epsilon$, then parse α_i implemented as:

```
/* empty statement */
```

Coding parse α_i: A Concrete Example

```
void parseWhileStmt() throws SyntaxError {
   match(Token.WHILE);
   match(Token.LPAREN);
   parseExpr();
   match(Token.RPAREN);
   parseStmt();
 }
```

Parsing Method for the Start Symbol

- If the start symbol S does not appear anywhere else: then

```
void parseS() {
    code for the alternatives of S
    match(Token.EOF);
}
```

- Otherwise, introduce a new start symbol, Goal:

```
void parseGoal() {
    parseS();
    match(Token.EOF);
}
```

Term (Predictive) Recursive Descent?

- Predictive (or non-backtracking): the parser always predicts the right production to use at every derivation step
- Recursive, a parsing method may call itself recursively either directly or indirectly.
- Descent: the parser builds the parse tree (or AST) by descending through it as it parses the program (Assignment 3).

Outline for the Rest of the Lecture

1. Definition of LL(1) grammar
2. One simplification in the presence of a nullable alternative
3. Eliminate left recursion and common prefixes
4. Write parsing methods in the presence of regular operators
5. Assignment 2

Definition of LL(1) Grammar

- A grammar is LL(1) if for every nonterminal of the form:

$$A \;\to\; \alpha_1 \mid \cdots \mid \alpha_n$$

  the select sets are pairwise disjoint, i.e.:

$$\textsf{Select}(A{\to}\alpha_i) \cap \textsf{Select}(A{\to}\alpha_j) \;=\; \emptyset$$

  for all i and j such that $i \neq j$.

- This implies there can be at most one nullable alternative

One Simplification When A Has a Nullable Alternative

void parseA() {
  if (currentToken.kind in $\mathsf{First}(A \to \alpha_1)$) {
    parse α_1
  $\dots$
  else if (currentToken.kind in $\mathsf{First}(A \to \alpha_{n-1})$){
    parse α_{n-1}
  else /* $A \to \alpha_n$ as the default (item 4, p. 193/229, Red/Purple Dragon */
    parse α_n
  }
}

- Suppose $\alpha_n \Longrightarrow^* \epsilon$ is the only nullable alternative
- Then $\mathsf{Select}(A \to \alpha_i) = \mathsf{First}(\alpha_i)$ for $1 \leqslant i < n$
- In fact, the coding still correct even if all are not nullable!

The Simplified Parsing Method Illustrated

- The grammar:

```
S --> A b
A --> a | ε
```

- The language: $\{b, ab\}$

$$\text{Select}(A \to a) = \text{First}(a) = \{a\}$$

$$\text{Select}(A \to \epsilon) = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(A) = \{b\}$$

V1:

```
void parseS() {
    parseA();
    match("b");
    match("EOF");
}

void parseA() {
    if (lookahead is "a")
        match("a");
    else if (lookahead is "b")
        /* do nothing */
    else
        print an error message
}
```

V2:

```
void parseS() {
    parseA();
    match("b");
    match("EOF");
}

void parseA() {
    if (lookahead is "a")
        match("a");
    // applies A -> ε otherwise
}
/* Some error detection postponed
but cannot miss any error */
```

Left-Recursive Grammars Are Not LL(1)

The parsing method for E of Grammar 1 in Slide 171:

```
void parseE() {
  switch (currentToken.kind) {
  case Token.INT: case Token.LPAREN:
    parseE();
    break;
  case Token.INT: case Token.LPAREN:
    parseE();
    match(Token.PLUS);
    parseT();
    break;
  case Token.INT: case Token.LPAREN:
    parseE();
    match(Token.MINUS);
    parseT();
    break;
  default:
    syntacticError(...);
    break;
  }
} /* this does not work */
```

Left Recursion

- Direct left-recursion:

$$A \rightarrow A\alpha$$

- Non-direct left-recursion:

$$\begin{array}{rcl} A & \rightarrow & B\alpha \\ B & \rightarrow & A\beta \end{array}$$

  – Algorithm 4.1 of text eliminates both kinds of left recursion
  – In real programming languages, non-direct left-recursion is rare
  – Not required

- A grammar with left recursion is not LL(1)

Eliminating Direct Left Recursion: Grammar Rewriting

- The grammar G_1:

$$
\begin{array}{lcll}
A & \rightarrow & \alpha & \quad // \ \alpha \text{ does not begin with } A \\
A & \rightarrow & A\beta_1 \mid A\beta_2 &
\end{array}
$$

- The transformed grammar G_2:

$$
\begin{array}{lcl}
A & \rightarrow & \alpha A' \\
A' & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \epsilon
\end{array}
$$

Eliminating Direct Left Recursion Using Regular Operators

- The grammar G_1:

$$\begin{array}{lcl} A & \rightarrow & \alpha \\ A & \rightarrow & A\beta_1 \mid A\beta_2 \end{array}$$

- The transformed grammar G_2:

$$A \rightarrow \alpha(\beta_1 \mid \beta_2)^*$$

- G_1 and G_2 define the same language: $L(G_1) = L(G_2)$
- Recommended to use in Assignment 2

The Expression Grammar

- The grammar with left recursion:

$$\text{Grammar 1}: \quad \begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T/F \mid F \\ F &\rightarrow \textbf{INT} \ \mid (E) \end{aligned}$$

- Eliminating left recursion using the Kleene Closure

$$\text{Grammar 3}: \quad \begin{aligned} E &\rightarrow T \ (\text{“+”} \ T \mid \text{“-”} \ T)^* \\ T &\rightarrow F \ (\text{“*”} \ F \mid \ \text{“/”} \ F)^* \\ F &\rightarrow \textbf{INT} \ \mid \text{“(”} \ E \ \text{“)”} \end{aligned}$$

All tokens are enclosed in double quotes to distinguish them for the regular operators: (,) and *

- Compare with Slide 171

Grammars with Common Prefixes Are Not LL(1)

- The dangling-else grammar:

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \textbf{ ELSE } \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{other}
\end{array}
$$

- The parsing method according to Slide 181:

```
void parseStmt() {
switch (currentToken.kind) {
case Token.IF:
  accept();
  match(Token.LPAREN);
  parseExpr();
  match(Token.RPAREN);
  parseStmt();
  match(Token.ELSE);
  parseStmt();
  break;
case Token.IF:
  accept();
  match(Token.LPAREN);
  parseExpr();
  match(Token.RPAREN);
  parseStmt();
  break;
...
```

Eliminating Common Prefixes: Left-Factoring

- The grammar G_1:

$$\begin{array}{rcl} A & \to & \alpha\beta_1 \mid \alpha\beta_2 \\ A & \to & \gamma \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{rcl} A & \to & \alpha A' \\ A & \to & \gamma \\ A' & \to & \beta_1 \mid \beta_2 \end{array}$$

- $L(G_1) = L(G_2)$
- A grammar with common prefixes is not LL(1)

Example: Eliminating Common Prefixes

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF} \text{ ''('' } \langle \text{expr} \rangle \text{ '')'' } \langle \text{stmt} \rangle \ \textbf{ELSE} \ \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF} \text{ ''('' } \langle \text{expr} \rangle \text{ '')'' } \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{other}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF} \text{ ''('' } \langle \text{expr} \rangle \text{ '')'' } \langle \text{stmt} \rangle \ \langle \text{else-clause} \rangle \\
\langle \text{else-clause} \rangle & \rightarrow & \textbf{ELSE} \ \langle \text{stmt} \rangle \ \mid \epsilon \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{other}
\end{array}
$$

Eliminating Common Prefixes using Choice Operator

- The grammar G_1:

$$\begin{array}{rcl} A & \rightarrow & \alpha\beta_1 \mid \alpha\beta_2 \\ A & \rightarrow & \gamma \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{rcl} A & \rightarrow & \alpha(\beta_1 \mid \beta_2) \\ A & \rightarrow & \gamma \end{array}$$

- Recommended to use in Assignment 2

Example: Eliminating Common Prefixes

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF } ”(” \ \langle \text{expr} \rangle \ ”)” \ \langle \text{stmt} \rangle \ \textbf{ELSE} \ \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF } ”(” \ \langle \text{expr} \rangle \ ”)” \ \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{other}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF } ”(” \ \langle \text{expr} \rangle \ ”)” \ \langle \text{stmt} \rangle \ (\ \textbf{ELSE} \ \langle \text{stmt} \rangle)? \\
\langle \text{stmt} \rangle & \rightarrow & \textbf{other}
\end{array}
$$

Compare with Slide 198

Coding parse α_i in the Presence of Regular Operators

- Suppose $\alpha_i = a(A)^*(B)^+b(C)?$, where A, B and C are nonterminals or a sequence of terminals and nonterminals
- parse α_i implemented as:

```
match("a");
while (currentToken.kind is in First(A))
  parse A;
do {
  parse B;
} while (currentToken.kind is in First(B))
match("b");
if (currentToken.kind is in First(C))
  parse C;
```

Assignment 2

- A subset of VC already implemented for you
- For expressions, you need to eliminate left-recursion on several nonterminals as illustrated in Slide 195
- You also need to eliminate some common prefixes (e.g., one for $\langle$primary-expr$\rangle$) as illustrated in Slide 200.
- A simple left-factoring can fix the LL(2) construct:

$$\langle \text{prog} \rangle \rightarrow (\ \langle \text{func-decl} \rangle \mid \langle \text{var-decl} \rangle \ )^*$$

- Everything else should be quite straightforward

Reading

- Chapter 2
- Red Dragon: Pages 176 – 178 and 188 – 190
- Purple Dragon: $4.3.3 – 4.3.4 and Pages 217 – 226

Next Class: Top-Down Parsing Revisited

First Sets for Grammar 1 (with Recursion)

$$
\begin{array}{lclclcl}
\text{First}(E) & = & \text{First}(E+T) & = & \text{First}(E-T) & = \\
\text{First}(T) & = & \text{First}(T*F) & = & \text{First}(T/F) & = \\
\text{First}(F) & = & \{(, \textbf{INT}\} \\
\text{First}((E)) & = & \{(\} \\
\text{First}(\textbf{INT}) & = & \{\textbf{INT}\}
\end{array}
$$

The explicit construction is left as an exercise.

Follow Sets for Grammar 1 (with Recursion)

$$\begin{aligned} \textsf{Follow}(E) &= \{+, -,), \$\} \\ \textsf{Follow}(T) = \textsf{Follow}(F) &= \{+, -, *, /,), \$\} \end{aligned}$$

The explicit construction is left as an exercise.

Select Sets for Grammar 1

Follow sets not used since the grammar has no ϵ-productions

$$
\begin{array}{lll}
\text{Select}(E \to E + T) & = \text{First}(E + T) & = \{(, \textbf{INT}\} \\
\text{Select}(E \to E - T) & = \text{First}(E - T) & = \{(, \textbf{INT}\} \\
\text{Select}(E \to T) & = \text{First}(T) & = \{(, \textbf{INT}\} \\
\text{Select}(T \to T * F) & = \text{First}(T * F) & = \{(, \textbf{INT}\} \\
\text{Select}(T \to T / F) & = \text{First}(T / F) & = \{(, \textbf{INT}\} \\
\text{Select}(T \to F) & = \text{First}(F) & = \{(, \textbf{INT}\} \\
\text{Select}(F \to \textbf{INT}) & = \text{First}(\textbf{INT}) & = \{\textbf{INT}\} \\
\text{Select}(F \to (E)) & = \text{First}((E)) & = \{(\}
\end{array}
$$

The explicit construction is left as an exercise.

Eliminating Direct Left Recursion: Grammar Rewriting

— General Case —

- The grammar G_1:

$$\begin{array}{lcl} A & \to & \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \text{ // } \alpha_i \text{ does not begin with } A \\ A & \to & A\beta_1 \mid A\beta_2 \mid \cdots \mid A\beta_m \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{lcl} A & \to & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \\ A' & \to & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A' \mid \epsilon \end{array}$$

- G_1 and G_2 define the same language: $L(G_1) = L(G_2)$
- Example: in Slide 171, Grammar 2 is the transformed version of Grammar 1

Eliminating Direct Left Recursion Using Regular Operators

— General Case —

- The grammar G_1:

$$
\begin{array}{rcl}
A & \rightarrow & \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \\
A & \rightarrow & A\beta_1 \mid A\beta_2 \mid \cdots \mid A\beta_m
\end{array}
$$

- The transformed grammar G_2:

$$
A \rightarrow (\alpha_1 \mid \cdots \mid \alpha_n)(\beta_1 \mid \cdots \mid \beta_m)^*
$$

- G_1 and G_2 define the same language: $L(G_1) = L(G_2)$