Announcements

- HW1 due on Monday February 8th
  - Name and date your submission
  - Submit electronically in Homework Server AND on paper in the beginning of class
    - Make sure you have an account in HW Server!
  - You may submit late on Thursday February 11th for 50% credit
  - No other submissions accepted

- Email questions to csci4430@cs.lists.rpi.edu

Last Class

- Top-down parsing vs. bottom-up parsing
  - Top-down parsing
    - Introduction
    - A backtracking parser
    - Recursive descent predictive parser
    - Table-driven top-down parser
    - LL(1) parsing tables, FIRST and FOLLOW sets

Today’s Lecture Outline

- Top-down (also called LL) parsing
  - LL(1) parsing tables, FIRST, FOLLOW and PREDICT sets
  - Writing an LL(1) grammar

- Bottom-up (also called LR) parsing
  - Model of a bottom-up (LR) parser

Programming Language Syntax

- Parsing

Read: finish Chapter 2.3.2 and start Chapter 2.3.3

LL(1) Parsing Tables

- One dimension is nonterminal to expand
- Other dimension is lookahead token

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>A</th>
<th>α</th>
<th>ε</th>
</tr>
</thead>
</table>

- E.g., entry “nonterminal A on terminal a” contains production $A \rightarrow a$
  - This means, when the parser is at nonterminal A and the lookahead token in the stream is a, the parser must expand A by production $A \rightarrow α$
**LL(1) Parsing Tables**

- We can construct an LL(1) parsing table for any context-free grammar.
  - In general, the table will contain multiply-defined entries. That is, for some nonterminal and lookahead token, more than one productions apply.

- A grammar whose LL(1) parsing table has no multiply-defined entries is said to be LL(1) grammar.
  - LL(1) grammars are a very special subclass of context-free grammars.

**FIRST and FOLLOW sets**

- Let $\alpha$ be any sequence of nonterminals and terminals.
  - $\text{FIRST}(\alpha)$ is the set of terminals $a$ that begin the strings derived from $\alpha$.
  - If there is a derivation $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon$ is in $\text{FIRST}(\alpha)$.

- Let $A$ be a nonterminal.
  - $\text{FOLLOW}(A)$ is the set of terminals $b$ (including special end-of-input marker $$) that can appear immediately to the right of $A$ in some sentential form: $\text{start} \Rightarrow^* \ldots A \ldots \Rightarrow^* \ldots$

**Computing FIRST**

- Apply these rules until no more terminals or $\varepsilon$ can be added to any $\text{FIRST}(\alpha)$ set.
  1. If $\alpha$ starts with a terminal $a$, then $\text{FIRST}(\alpha) = \{a\}$.
  2. If $\alpha$ is a nonterminal $X$, where $X \rightarrow \varepsilon$, then $\varepsilon \in \text{FIRST}(\alpha)$.
  3. If $\alpha$ is a nonterminal $X \rightarrow Y_1 Y_2 \ldots Y_n$, then place $\alpha$ in $\text{FIRST}(X)$ if for some $i$, $Y_i$ is in $\text{FIRST}(Y_i)$ and $\varepsilon$ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1}), \text{FIRST}(Y_i), \ldots, \text{FIRST}(Y_n)$.

**Exercise**

| Start $\Rightarrow S \ \$$ | $B \Rightarrow z \ S \ | \varepsilon$
|--------------------------|---------------------|
| $S \Rightarrow x \ S \ | A \ y$ | $C \Rightarrow v \ S \ | \varepsilon$
| $A \Rightarrow BCD \ | \varepsilon$ | $D \Rightarrow w \ S$

**Warm-up Exercise**

- $\text{start} \Rightarrow \text{expr} \ \$$
- $\text{expr} \Rightarrow \text{term} \ \text{term_tail}$
- $\text{term_tail} \Rightarrow \text{term} \ + \ \text{term_tail} \ | \varepsilon$
- $\text{term} \Rightarrow \text{id} \ \text{factor_tail}$
- $\text{factor_tail} \Rightarrow \text{factor} \ * \ \text{factor_tail} \ | \varepsilon$

**FIRST(term) = \{id\}**
- $\text{FIRST(expr)} = \text{FIRST(start)}$
- $\text{FIRST(term_tail)} = \text{FIRST}(+ \ \text{term_tail})$
- $\text{FIRST(factor_tail)} = \text{FIRST}(\varepsilon)$

**Computing FOLLOW**

- Apply these rules until nothing can be added to any FOLLOW($A$) set.
  1. If there is a production $A \rightarrow aB\varepsilon$, then everything in $\text{FIRST}(B)$ except $\varepsilon$ should be added to $\text{FOLLOW}(B)$.
  2. If there is a production $A \rightarrow aB$, or a production $A \rightarrow aB\beta$, where $\text{FIRST}(B)$ contains $\varepsilon$, then everything in $\text{FOLLOW}(A)$ should be added to $\text{FOLLOW}(B)$. 

**Exercise**

- Compute $\text{FIRST}$ sets:
  - $\text{FIRST}(x \ S) = \text{FIRST}(S)$
  - $\text{FIRST}(A \ y) = \text{FIRST}(A)$
  - $\text{FIRST}(BCD) = \text{FIRST}(B)$
  - $\text{FIRST}(z \ S) = \text{FIRST}(C)$
  - $\text{FIRST}(v \ S) = \text{FIRST}(D)$
  - $\text{FIRST}(w \ S) = \text{FIRST}($
Warm-up

```
FOLLOW(expr) = { $$ }
FOLLOW(term_tail) =
FOLLOW(term) =
FOLLOW(factor_tail) =
```

Exercise

```
FOLLOW(A) =
FOLLOW(B) =
FOLLOW(C) =
FOLLOW(D) =
FOLLOW(S) =
```

PREDICT Sets

```
PREDICT(A → α) =
  FIRST(α) if α does not derive ε
  (FIRST(α) \{ε\}) \cup FOLLOW(A) if α derives ε
```

Constructing LL(1) Parsing Table

- Algorithm uses PREDICT sets
  - foreach production \( A → α \) in grammar \( G \)
  - foreach symbol \( c \) in PREDICT(\( A → α \))
    - add \( A → α \) to entry parse_table[\( A, c \)]
  - If all entries in parse_table contain at most one production, then \( G \) is said to be LL(1)

Exercise

```
Compute PREDICT sets:
PREDICT(S → x S) =
PREDICT(S → A y) =
PREDICT(A → BCD) =
PREDICT(A → ε) =
... etc...
```

Writing an LL(1) Grammar

- Most context-free grammars are not LL(1) grammars
- Obstacles to LL(1)-ness
  - Left recursion is an obstacle. Why?
  - Common prefixes are an obstacle. Why?

```
expr → expr + term | term
term → term * id | id
```

```
stmt → if b then stmt else stmt | if b then stmt | a
```
Removal of Left Recursion

- Left recursion can be removed from a grammar mechanically
- Started from this left-recursive expression grammar:
  \[ expr \rightarrow expr + term | term \]
  \[ term \rightarrow term \ast id | id \]
- After removal of left recursion we obtain this equivalent grammar, which is LL(1):
  \[ expr \rightarrow term \ast id | id \]
  \[ term \rightarrow term \ast id | id \]

Removal of Common Prefixes

- Common prefixes can be removed mechanically as well, by using left-factoring
- Original if-then-else grammar:
  \[ stmt \rightarrow if b then stmt else stmt | \]
  \[ a | \]
- After left-factoring:
  \[ stmt \rightarrow if b then stmt else_part | a \]
  \[ else_part \rightarrow else stmt | \]

Exercise

- Compute FIRSTs:
  \[ FIRST(stmt $$), FIRST(if b then stmt else_part), FIRST(a), FIRST(else stmt) \]
- Compute FOLLOW:
  \[ FOLLOW(else_part) \]
- Compute PREDICT sets for all 5 productions
- Construct the LL(1) parsing table. Is this grammar an LL(1) grammar?

Lecture Outline

- Top-down (also called LL) Parsing (continue)
  - LL(1) parsing table, FIRST, FOLLOW and PREDICT sets
  - Writing an LL(1) grammar
- Bottom-up (also called LR) Parsing
  - Model of the bottom-up (LR) parser

Bottom-up Parsing

- Terminals are seen in the order of appearance in the token stream
  \[ id, id, id ; \]
- Parse tree is constructed
  - From the leaves to the top
  - A right-most derivation in reverse

Stack | Input | Action
--- | --- | ---
| id, id, id | shift
| id, id, id | shift
| id, id, id | shift
| id, id, id | shift
| id, id, id | shift
| id, id, id | reduce by list_tail→;
## Bottom-up Parsing

**Stack**
- `id, id, id list_tail`

**Input**
- `id + id * id`

**Action**
- Reduce by
  - `list_tail -> . id list_tail`

### Stack
- `id, id list_tail`

### Input
- `id + id * id`

### Action
- Reduce by
  - `list_tail -> . id list_tail`

### Stack
- `id list_tail`

### Input
- `id + id * id`

### Action
- Reduce by
  - `list -> . id list_tail`

### Stack
- `list`

### Input
- `id + id * id`

### Action
- ACCEPT

---

## Bottom-up Parsing

- Also called LR parsing
- LR parsing is better than LL parsing
  - Accepts larger class of languages
  - Just as efficient!
- LR parsers work with LR(k) grammars
  - L stands for “left-to-right” scan of input
  - R stands for “rightmost” derivation
  - k stands for “need k tokens of lookahead”
- We are interested in LR(0) and LR(1) and variants in between

---

## LR Parsing

- The parsing method used in practice
  - LR parsers recognize virtually all PL constructs
  - LR parsers recognize a much larger set of grammars than predictive parsers
  - LR parsing is efficient
- LR parsing variants
  - SLR (or Simple LR)
  - LALR (or Lookahead LR) – yacc/bison generate LALR parsers
  - LR (Canonical LR)
  - SLR < LALR < LR

---

## Main Idea

- **Stack** ← **Input**
  - **Stack**: holds the part of the input seen so far
    - A string of both terminals and nonterminals
  - **Input**: holds the remaining part of the input
    - A string of terminals
  - **Parser** performs two actions
    - Reduce: parser pops a “suitable” production right-hand-side off the stack, and pushes the production left-hand-side on the stack
    - Shift: parser pushes next terminal from the input on top of the stack

---

## Example

- Recall the grammar
  - `expr -> expr + term | term`
  - `term -> term * id | id`

  - This is not LL(1) because it is left recursive
  - LR parsers can handle left recursion!

- Consider string
  - `id + id * id`
id + id*id

**Stack**  
expr+term *id shift *
expr+term* id shift id
expr+term*id reduce by term→term *id
expr+term reduce by expr→expr*term
expr ACCEPT, SUCCESS

**Input**  
expr
+ term
* id

**Action**  
Reduce by term
Reduce by expr
Reduce by id
Shift

---

**Model of an LR parser**

- Stack is \( (s_0, X_1, s_1, \ldots, X_m, s_m) \), input pointer at \( a_i \)
- \( \text{action}[s_m, a_i] \) is *shift* \( s \)
  - Push \( a_i \) and state \( s \) on stack: \( (s_0, X_1, s_1, \ldots, X_m, s_m, a_i, s) \)
- \( \text{action}[s_m, a_i] \) is *reduce* by \( A \rightarrow \beta \)
  - Pop \( \beta \) (i.e., pop 2*|\( \beta \)| things off the stack - all symbols in \( \beta \) plus all their corresponding states): \( (s_0, X_1, s_1, \ldots, X_m, s_m, |\beta|, s) \)
  - Push \( A \) and goto[|\( \beta \)|, A] on top of the stack: \( (s_0, X_1, s_1, \ldots, X_m, s_m, |\beta|, A, s) \)

---

**LR Parsing Table**

<table>
<thead>
<tr>
<th>State</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$$</th>
<th>expr</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td>$$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s5</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>s5</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Top-down (also called LL) Parsing (continue)
  - LL(1) parsing table and predict sets
  - Writing an LL(1) grammar
- Bottom-up (also called LR) Parsing
  - Model of the bottom-up (LR) parser
  - LR parsing table

Next Class

- We will continue with Bottom-up Parsing.
  Keep reading Chapter 2.3.3