Announcements

- Quiz 1
- Hold on to paper, bring over at the end
- HW1 due today
- HW2 will be posted tonight
  - Due Tue, Sep 18 at 2pm in Submitty!
  - Team assignment. Form teams in Submitty!
  - Top-down and bottom-up grammars and parsing

Last Class

- Top-down parsing vs. bottom-up parsing
- Top-down parsing
  - Introduction
  - A top-down depth-first parser (with backtracking)
  - Recursive descent predictive parser
  - Table-driven top-down predictive 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
  - Intro with examples
  - Handles
  - LR items
  - More

Programming Language Syntax
Parsing

Read: finish Chapter 2.3.2 and start Chapter 2.3.3

LL(1) Parsing Tables

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

```
<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>expr $</td>
<td>$</td>
<td>expr</td>
<td>term term_tail</td>
</tr>
<tr>
<td>expr</td>
<td>term term_tail</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term</td>
<td>id factor_tail</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>factor_tail</td>
<td>-</td>
<td>*</td>
<td>id factor_tail</td>
<td>$$</td>
</tr>
</tbody>
</table>
```

E.g., entry “nonterminal A on terminal a” contains production $A \rightarrow \alpha$

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 $A \rightarrow \alpha$
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. Why?

FIRST and FOLLOW sets

- Let α be any sequence of nonterminals and terminals
  - FIRST(α) is the set of terminals a that begin the strings derived from α
  - If there is a derivation α ⇒* ε, then ε is in FIRST(α)
- Let A be a nonterminal
  - 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 ε can be added to any FIRST(α) set
  1. If α starts with a terminal a, then FIRST(α) = \{a\}
  2. If α is a nonterminal X, where X \rightarrow ε, then add ε to FIRST(α)
  3. If α is a nonterminal X \rightarrow Y_1 Y_2 \ldots Y_n then place α in FIRST(X) if for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1) and ε is in all of FIRST(Y_i). ...
  - If \epsilon is in all of FIRST(Y_i), add ε to FIRST(X).
  - Everything in FIRST(Y_i) is surely in FIRST(X)
  - If Y_i does not derive ε, then we add nothing more.
  - Otherwise, we add FIRST(Y_i), and so on
  - Similarly, if α is Y_1 Y_2 \ldots Y_n, we'll repeat the above

Warm-up Exercise

\[
\begin{align*}
\text{FIRST}(\text{term}) &= \{ \text{id} \} \\
\text{FIRST}(\text{expr}) &= \text{FIRST}(\text{start}) = \\
\text{FIRST}(\text{term.tail}) &= \\
\text{FIRST}(\text{term.tail} + \text{term.tail}) &= \\
\text{FIRST}(\text{factor.tail}) &= \\
\text{FIRST}(\text{term.tail} + \text{term.tail}) &= \\
\end{align*}
\]
Exercise

\[
\begin{align*}
\text{start} & \rightarrow S \\
S & \rightarrow x S | A y \\
A & \rightarrow BCD | \varepsilon
\end{align*}
\]

Exercise

Compute FIRST sets:

\[
\begin{align*}
\text{FIRST}(x S) & = \text{FIRST}(S) \\
\text{FIRST}(A y) & = \text{FIRST}(A) \\
\text{FIRST}(BCD) & = \text{FIRST}(B) \\
\text{FIRST}(z S) & = \text{FIRST}(C) \\
\text{FIRST}(v S) & = \text{FIRST}(D) \\
\text{FIRST}(w S) & = 1
\end{align*}
\]

Exercise

\[
\begin{align*}
\text{start} & \rightarrow \text{expr} \\
\text{expr} & \rightarrow \text{term} \text{ term_tail} \\
\text{term} & \rightarrow \text{id} \text{ factor_tail} \\
\text{factor_tail} & \rightarrow * \text{id} \text{ factor_tail} | \varepsilon
\end{align*}
\]

Exercise

Compute FOLLOW sets:

\[
\begin{align*}
\text{FOLLOW}(A) & = \text{FOLLOW}(B) = \text{FOLLOW}(C) = \text{FOLLOW}(D) = \text{FOLLOW}(S) = \\
\text{FOLLOW}(expr) & = \{ \text{$$} \} \\
\text{FOLLOW(term)} & = \text{FOLLOW(term_tail)} = \text{FOLLOW(factor_tail)} =
\end{align*}
\]

Exercise

\[
\begin{align*}
\text{start} & \rightarrow S \\
S & \rightarrow x S | A y \\
A & \rightarrow BCD | \varepsilon
\end{align*}
\]

Constructing LL(1) Parsing Table

- Algorithm uses PREDICT sets
  - foreach production \( A \rightarrow \alpha \) in grammar \( G \)
  - foreach symbol \( c \) in PREDICT(\( A \rightarrow \alpha \))
    - add \( A \rightarrow \alpha \) to entry parse_table[\( A, c \)]

- If each entry in parse_table contains at most one production, then \( G \) is said to be LL(1)
Exercise

\[
\begin{align*}
\text{start} & \rightarrow \text{stmt} $$ & \text{stmt} & \rightarrow \text{if b then stmt else_part} | \text{a else_part} | \text{else stmt} | \epsilon \\
\text{stmt} & \rightarrow \text{if b then stmt else_part} \\
\text{else_part} & \rightarrow \text{else stmt} | \epsilon
\end{align*}
\]

Compute PREDICT sets:

\[
PREDICT(S \rightarrow x S) = PREDICT(S \rightarrow Ay) = PREDICT(A \rightarrow BCD) = PREDICT(A \rightarrow \epsilon) = \ldots \text{ etc } \ldots
\]

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?

Removal of Left Recursion

- Left recursion can be removed from a grammar mechanically
- Started from this left-recursive expression grammar:

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} | \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{id} | \text{id}
\end{align*}
\]

- After removal of left recursion we obtain this equivalent grammar, which is LL(1):

\[
\begin{align*}
\text{expr} & \rightarrow \text{term_tail} \\
\text{term_tail} & \rightarrow + \text{term} \text{term_tail} | \epsilon \\
\text{factor_tail} & \rightarrow \ast \text{id} \text{factor_tail}
\end{align*}
\]

Removal of Common Prefixes

- Common prefixes can be removed mechanically as well, by using left-factoring
- Original if-then-else grammar:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if b then stmt else stmt} | \text{if b then stmt} | \text{a} \\
\text{else_part} & \rightarrow \text{else stmt} | \epsilon
\end{align*}
\]

- After left-factoring:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if b then stmt else_part} | \text{a} \\
\text{else_part} & \rightarrow \text{else stmt} | \epsilon
\end{align*}
\]

Exercise

- Compute FIRSTs:
  - FIRST($\text{stmt} $$), \text{FIRST(} \text{stmt} \text{), FIRST(} \text{if b then stmt else_part} \text{), FIRST(} \text{a} \text{), FIRST(} \text{else stmt} \text{)}$

- 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?

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
  - Intro with examples
  - Handles
  - LR items
  - More
### 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

### LR Parsing

- **Main Idea**
  - **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 top of stack, and pushes production’s left-hand-side on the stack
    - **Shift**: parser pushes next terminal from the input on top of the stack

### Bottom-up Parsing

- **Stack**: holds the part of the input seen so far
- **Input**: holds the remaining part of the input
- **Action**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id, id, id list_tail</code></td>
<td>reduce by <code>list_tail -&gt; , id list_tail</code></td>
<td></td>
</tr>
<tr>
<td><code>id, id list_tail</code></td>
<td>reduce by <code>list_tail -&gt; , id list_tail</code></td>
<td></td>
</tr>
<tr>
<td><code>id list_tail</code></td>
<td>reduce by <code>list -&gt; id list_tail</code></td>
<td></td>
</tr>
<tr>
<td><code>list</code></td>
<td>ACCEPT</td>
<td></td>
</tr>
</tbody>
</table>

---

### Bottom-up Parsing

- **Stack**: holds the part of the input seen so far
- **Input**: holds the remaining part of the input
- **Action**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id, id, id;</code></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td><code>id, id, id;</code></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td><code>id, id;</code></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td><code>id, id;</code></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td><code>id, id, id;</code></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td><code>id, id, id;</code></td>
<td>reduce by <code>list_tail -&gt; ;</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

- Recall the grammar
  
  \[
  \begin{align*}
  &expr \rightarrow expr + term | term \\
  &term \rightarrow term \cdot id | id
  \end{align*}
  \]

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

- Consider string
  
  \[id \cdot id \cdot id\]

id + id\cdot id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr+term</td>
<td>*id</td>
<td>shift *</td>
</tr>
<tr>
<td>expr+term*</td>
<td>id</td>
<td>shift id</td>
</tr>
<tr>
<td>expr+term*id</td>
<td>reduce by</td>
<td>term \rightarrow \term*id</td>
</tr>
<tr>
<td>expr+term</td>
<td>reduce by</td>
<td>expr\rightarrow expr*term</td>
</tr>
<tr>
<td>expr</td>
<td>ACCEPT,</td>
<td>SUCCESS</td>
</tr>
</tbody>
</table>

id + id\cdot id

Sequence of reductions performed by parser

- A rightmost derivation in reverse
- The stack (e.g., expr) concatenated with remaining input (e.g., +id*id) gives a sentential form (expr+id*id) in the rightmost derivation

id + id\cdot id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id+id*id</td>
<td>shift id</td>
<td></td>
</tr>
<tr>
<td>id<em>id</em>id</td>
<td>reduce by</td>
<td>term \rightarrow id</td>
</tr>
<tr>
<td>term<em>id</em>id</td>
<td>reduce by</td>
<td>expr\rightarrow term</td>
</tr>
<tr>
<td>expr<em>id</em>id</td>
<td>shift +</td>
<td></td>
</tr>
<tr>
<td>expr+</td>
<td>id*id</td>
<td>shift id</td>
</tr>
<tr>
<td>expr+term*id</td>
<td>reduce by</td>
<td>term \rightarrow id</td>
</tr>
<tr>
<td>expr+term</td>
<td>reduce by</td>
<td>expr\rightarrow expr*term</td>
</tr>
<tr>
<td>expr</td>
<td>ACCEPT,</td>
<td>SUCCESS</td>
</tr>
</tbody>
</table>

Handle

- A handle
  
  - Consider a rightmost derivation
    
    \[S \Rightarrow \ldots \Rightarrow \alpha A \Rightarrow \alpha \beta w.\] We say that
    
    \[A \rightarrow \beta\] at position \(\alpha\) is a handle of \(\alpha \beta w\)

- Recall our example id+id\cdot id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>expr+term</td>
<td>*id</td>
<td></td>
</tr>
<tr>
<td>expr+term*id</td>
<td>reduce by</td>
<td>term \rightarrow \term*id</td>
</tr>
<tr>
<td>expr+term</td>
<td>reduce by</td>
<td>expr\rightarrow expr*term</td>
</tr>
<tr>
<td>expr</td>
<td>ACCEPT,</td>
<td>SUCCESS</td>
</tr>
</tbody>
</table>

Question

- Consider id*id*id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>expr+term</td>
<td>*id</td>
<td></td>
</tr>
<tr>
<td>expr+term*id</td>
<td>reduce by</td>
<td>term \rightarrow id</td>
</tr>
</tbody>
</table>

  - Answer: No! It brings sentential form \(term*id*id\) into \(expr*id*id\) which is NOT derivable from \(expr\)!
Question

- How about Stack Input
- `term*id *id` Is `term → term*id` at position ε a handle of `term*id*id`?

Answer: Yes! It brings sentential form `term*id*id` into `term*id` which is clearly derivable: `expr → term → term*id`

Model of an LR parser

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
  - Intro with examples
  - Handles
  - LR items
  - More

Closure of an LR Item

- The closure of an LR item `A → αβ` is the set of LR items formed as follows:
  - `A → αβ` is in the closure of `A → αβ`
  - If the dot is in front of a nonterminal `B` for some item in the closure, then all of `B → γ1`, `B → γ2`, ... `B → γn` are in the closure (`B → γ1`, `B → γ2`, ... `B → γn` are all productions for `B`)

LR Items

- An LR item is a production with a dot at some position on the right-hand side
  - E.g., `A → αβ`
  - We are trying to find an `A`  
  - We have already seen `α` (it is on top of the stack)
  - We are looking for `β`

- Groups related LR items into sets.
- Sets correspond to parsing states, state 0, 1, etc.
### Example

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code> → <code>expr</code></td>
</tr>
<tr>
<td><code>expr</code> → <code>expr</code> + <code>term</code></td>
</tr>
</tbody>
</table>

#### Question

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code> → <code>expr</code></td>
</tr>
<tr>
<td><code>expr</code> → <code>expr</code> + <code>term</code></td>
</tr>
</tbody>
</table>

#### Summary

- **Top-down (also called LL) parsing**
  - `LL(1)` parsing table and predict sets
  - Writing an `LL(1)` grammar
- **Bottom-up (also called LR) parsing**
  - Intro with examples
  - Handles
  - LR items

### Next Class

- We will conclude discussion of bottom-up parsing. Keep reading Chapter 2.3.3