CS 415: Lecture 7

- Predictive parsing, cont’d
- Bottom up parsing

Parsing LL(1) Grammars

Example:
1. \texttt{\textlt{goal}\textgt{} ::= \textlt{expr}\textgt{}}
2. \texttt{\textlt{expr}\textgt{} ::= \textlt{term}\textlt{expr’}\textgt{}}
3. \texttt{\textlt{expr’}\textgt{} ::= + \textlt{expr}\textgt{}}
4. | \texttt{\textlt{expr’}\textgt{} ::= \textlt{expr}\textgt{}}
5. | \texttt{\varepsilon}
6. \texttt{\textlt{term}\textgt{} ::= \textlt{factor}\textlt{term’}\textgt{}}
7. \texttt{\textlt{term’}\textgt{} ::= * \textlt{term}\textgt{}}
8. | \texttt{\textlt{term’}\textgt{} ::= \textlt{term’}\textgt{}}
9. | \texttt{\varepsilon}
10. \texttt{\textlt{factor}\textgt{} ::= \textlt{num}\textgt{}}
11. | \texttt{\textlt{id}\textgt{}}
Table-Driven LL(1) Parsing

push eof
push Start Symbol
X ← top-of-stack
repeat
    if X is a terminal then
    if X = token then
        pop X
        token ← next_token()
    else error()
    else /* X is a non-terminal */
    if M[X, token] = X → Y₁ Y₂ ⋯ Yₖ then
        pop X
        push Y₁, Y₂, ⋯, Yₖ
    else error()
X ← top-of-stack
until X = eof
if token ≠ eof then error()

Recursive Descent Parsing

goal:
token ← next_token();
if (expr() = ERROR | token ≠ EOF) then
    return ERROR;
else return OK;

expr:
if (term() = ERROR) then
    return ERROR;
else return expr_prime();

expr_prime:
if (token = PLUS) then
    token ← next_token();
    return expr();
else if (token = MINUS) then
    token ← next_token();
    return expr();
else return OK;

term:
if (factor() = ERROR) then
    return ERROR;
else return term_prime();

term_prime:
if (token = PLUS) then
    token ← next_token();
    return term();
else if (token = MINUS) then
    token ← next_token();
    return term();
else if (token = EOF) then
    return OK;
else return ERROR;

factor:
if (token = DIGIT) then
    token ← next_token();
    return OK;
else if (token = EOF) then
    return OK;
else if (token = ID) then
    token ← next_token();
    return OK;
else return ERROR;
Error Recovery

- Panic-mode error recovery
  - For each non-terminal, construct a set of terminals on which the parser can synchronize - for a non-terminal $A$, call this $\text{SYNCH}(A)$
  - When an error occurs with non-terminal $A$ on top of the stack, scan until an element of $\text{SYNCH}(A)$ is found, then pop $A$ and continue
- Heuristics for building $\text{SYNCH}$
  1. If $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNCH}(A)$
  2. Place keywords that start statements in $\text{SYNCH}(A)$
  3. If we can't match a terminal on the top of the stack
     1. Pop the terminal
     2. Print a message saying the proper terminal was inserted
     3. Continue the parse

Bottom-Up Parsing

- The parser repeatedly matches the right-hand side (rhs) of a production against a sub-string in the current right-sentential form
- At each match, it applies a reduction to build the parse tree
  1. Each reduction replaces the matched sub-string (handle) with the non-terminal on the left-hand side (lhs) of the production (handle pruning)
  2. Each reduction adds an internal node to the current parse tree
  3. The result is another right-sentential form
- The final result is a rightmost derivation, in reverse
Example

Rightmost derivation of
\( a+b+c \), handles in red

\[
\begin{align*}
S &\rightarrow E \quad (1) \\
E &\rightarrow E + T \quad (2) \\
E &\rightarrow T \quad (3) \\
T &\rightarrow id \quad (4)
\end{align*}
\]

Actions: shift, reduce, accept, error

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id1 + id2 + id3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ id1</td>
<td>+ id2 + id3 $</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ T</td>
<td>+ id2 + id3 $</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>$ E</td>
<td>+ id2 + id3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E +</td>
<td>id2 + id3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + id2</td>
<td>+ id3 $</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ E + T</td>
<td>+ id3 $</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ E</td>
<td>+ id3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E +</td>
<td>id3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + id3</td>
<td>$</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ E + T</td>
<td>$</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ E</td>
<td>$</td>
<td>reduce (1)</td>
</tr>
<tr>
<td>$ S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Trick appears to be scanning the input and finding valid right-sentential forms.
Handles

- We are trying to find a sub-string \( \alpha \) of the current right-sentential form where
  - \( \alpha \) matches some production \( A ::= \alpha \)
  - reducing \( \alpha \) to \( A \) is one step in reverse of a rightmost derivation
- We will call such a string a handle
- Formally, a handle of a right-sentential form \( \gamma \) is a production \( A ::= \beta \) and a position in \( \gamma \) where \( \beta \) may be found. Convention: position specifies the right end of the handle
- If \( (A ::= \beta, k) \) is a handle, then replacing the \( \beta \) in \( \gamma \) at position \( k \) with \( A \) produces the previous right-sentential form in a rightmost derivation of \( \gamma \)

A Handle in the Parse Tree

```
Z
     
  A
     
α β
     
Action: reduce \( \beta \) to \( A \)
```

Rutgers University, DCS 9 CS 415: Compilers
### Handle-Pruning

The process we use to construct a bottom-up parse is called **handle-pruning**.

To construct a rightmost derivation:

\[ S' = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w \]

we set \( i \) to \( n \) and apply the following simple algorithm:

**Algorithm:**

1. Find the handle \((A_i := \beta_i, k_i)\) in \( \gamma_i \)
2. Replace \( \beta_i \) with \( A_i \) to generate \( \gamma_{i-1} \)

---

### Handles

**Provable fact:**

If \( G \) is unambiguous, then every right-sentential form has a unique handle.

**Proof:**

1. \( G \) is unambiguous \( \Rightarrow \) rightmost derivation is unique.
2. \( \Rightarrow \) a unique production \( A \vdash \beta \) applied to take \( \gamma_{i-1} \) to \( \gamma_i \)
3. \( \Rightarrow \) a unique position \( k \) at which \( A \vdash \beta \) is applied
4. \( \Rightarrow \) a handle \((A := \beta, k)\)
Handles

Provable fact:

The substring to the right of a handle contains only terminal symbols.

Proof: Follows from the fact that all $\gamma_2$ are right-sentential forms.

Corollary

The right end of a handle is to the right of the previously reduced variable.

Shift-Reduce Parsing

- Shift-reduce parsers use a stack and an input buffer
  1. Initialize the stack with $\$$
  2. Repeat until the top of the stack is the goal symbol and the input token is EOF
     a. Find the handle
        if we don't have a handle on top of the stack, shift an input symbol onto the stack
     b. Prune the handle
        if we have a handle ($A ::= \beta, k$) on top of the stack, reduce
           i. pop $|\beta|$ symbols off the stack
           ii. push $A$ onto the stack
Example

<table>
<thead>
<tr>
<th>Actions: shift, reduce, accept, error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
</tr>
<tr>
<td>$</td>
</tr>
<tr>
<td>$ id1</td>
</tr>
<tr>
<td>$ T</td>
</tr>
<tr>
<td>$ E</td>
</tr>
<tr>
<td>$ E +</td>
</tr>
<tr>
<td>$ E + id2</td>
</tr>
<tr>
<td>$ E + T</td>
</tr>
<tr>
<td>$ E</td>
</tr>
<tr>
<td>$ E + id3</td>
</tr>
<tr>
<td>$ E + T</td>
</tr>
<tr>
<td>$ E</td>
</tr>
<tr>
<td>$ S</td>
</tr>
</tbody>
</table>

Trick appears to be scanning the input and finding valid right-sentential forms.

Shift-Reduce Parsing

- **Actions**
  - **Shift** - push token onto stack
  - **Reduce** - remove handle from stack and push on corresponding non-terminal
  - **Accept** - recognize sentence when stack contains only the distinguished symbol and input is empty
  - **Error** - happens when none of the above is possible; means original input was not a sentence!
Possible Problems

- Can get into conflicts where one rule implies *shift* while another implies *reduce*
  
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \]
  
  On stack: if E then S
  
  Input: else
  
  Should *shift* trying for 2nd rule or *reduce* by first rule?

Possible Problems

- Can have two grammar rules with same right hand side which leads to *reduce-reduce* conflicts
  
  A \rightarrow \alpha \text{ and } B \rightarrow \alpha \text{ both in grammar}

  When \alpha\ on top of the stack, how know which production choose? That is, whether to *reduce* to A or B?

- In both kinds of conflicts, problem is with the grammar - grammars are ambiguous in both cases.
  
  You should already suspect that bottom-up parsers cannot handle ambiguity either (unless there’s backtracking).
Operator Precedence Parsing

- A simplified bottom up parsing technique used for expression grammars
- Requires
  - No right hand side of rule is empty
  - No right hand side has 2 adjacent non-terminals
- Drawbacks
  - Small class of grammars qualify
  - Overloaded operators are hard (unary minus)
  - Parser correctness hard to prove

Operator Precedence

- Define three precedence relations
  - a < b, a yields in precedence to b
  - a > b, a takes precedence over b
  - a = b, a has same precedence as b
- Find handle as a <====> pattern at top of stack;
- Check relation between top of stack and next input symbol
- Basically, ignore non-terminals
Example

\[ Z \rightarrow E \\
E \rightarrow E \cdot E \mid E + E \mid id \]

Define precedence relations between + and *.

\[ + < *, * > +, + > +, * > * \] (last 2 ensure left associativity)

Form table of precedences.

Now parse using the table.

<table>
<thead>
<tr>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>+</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>

Example

Compare top of stack token to next input token.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Compares</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>&lt;</td>
<td>id1 + id2 * id3 $</td>
</tr>
<tr>
<td>$&lt; id1</td>
<td>&gt;</td>
<td>+ id2 * id3 $</td>
</tr>
<tr>
<td>$E</td>
<td>&lt;</td>
<td>+ id2 * id3 $</td>
</tr>
<tr>
<td>$E +</td>
<td>&lt;</td>
<td>id2 * id3 $</td>
</tr>
<tr>
<td>$E + id2</td>
<td>&gt;</td>
<td>* id3 $</td>
</tr>
<tr>
<td>$E + E</td>
<td>&lt;</td>
<td>* id3 $</td>
</tr>
<tr>
<td>$E + E *</td>
<td>&lt;</td>
<td>id3 $</td>
</tr>
<tr>
<td>$E + E * &lt; id3</td>
<td>&gt;</td>
<td>$</td>
</tr>
<tr>
<td>$E + E * E</td>
<td>&gt;</td>
<td>$</td>
</tr>
<tr>
<td>$E + E</td>
<td>&gt;</td>
<td>$</td>
</tr>
<tr>
<td>$E</td>
<td>&gt;</td>
<td>$</td>
</tr>
<tr>
<td>accept</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Making OP parsing practical

- How to store these precedences compactly?
- Precedence functions
  - Find functions $f(), g()$ such that
    - $f(\text{token}_1) > g(\text{token}_2)$ means $\text{token}_1 > \text{token}_2$
    - $f(\text{token}_1) = g(\text{token}_2)$ means $\text{token}_1 = \text{token}_2$
    - $f(\text{token}_1) < g(\text{token}_2)$ means $\text{token}_1 < \text{token}_2$
  - Graph partitioning algorithm to find $f(), g()$ if possible.

Precedence Functions

- Form graph from table of precedences
  - Nodes formed by $f(\text{token}_1), f(\text{token}_2), \ldots, g(\text{token}_1)$ etc.
    - Form equivalence classes of nodes based on the $=$ relation (equal precedence, e.g., $*/$)
  - Edges show required relations between function values
    - If $\text{token}_1 > \text{token}_2$, then $f(\text{token}_1) \rightarrow g(\text{token}_2)$
    - If $\text{token}_1 < \text{token}_2$, then $f(\text{token}_1) \leftarrow g(\text{token}_2)$
  - If the graph is acyclic, then can find integer value assignments for the range values of $f, g$
    - Let value of $f(\text{token}_1)$ be the length of the longest path from the node representing $f(\text{token}_1)$
Example

```
id + * $
id > > >
+ < > < >
* < > > >
$ < < <
```

Acyclic graph yields
```
id + * $
```
```
f(id) g(id)
g(+) g(*) g($) f(*) f(+)
f($)```

For Next Class

- Read ASU 4.6 & 4.7
- Remember that written assignment 1 is due at beginning of class