Recitation #9

Administrative
- Assignment 4 is due next Monday (03/28 @ 11:59 p.m.)

Bottom-Up Parser
- **Bottom-up Parser**, i.e. we start with the terminals in the input string and subsequently compute recognized parse trees by going from already recognized rhs of productions to the non-terminal on the lefthand side
  - In a way it’s a flipped parse tree: beginning at the leaves (terminals) and working up towards the root
  - Most prevalent type is based on concept of LR(k) parsers; “L” stands for left-to-right scanning, “R” constructs rightmost derivation in reverse, “k” is the number of input symbols of lookahead → usually deal with LR(1)
- **Reductions**
  - Reduction is the reverse of a production
  - Then bottom-up parsing can be thought of as “reducing” the input string to the start variable
  - Example expression “id * id”:
    - id*id, F*id, T*id, T*F, T, E
  - It builds a right derivation in reverse:
    - $E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow id * id$
  - How do we decide which reduction to apply? → **“Handle Pruning”**
    - A handle is a substring that matches the body of a production AND whose reduction represents one step along the rightmost derivation reverse
    - The substring to the right of the handle only contains non-terminals!
    - Not all leftmost substrings that match the body of some production are handles; only when they are part of the rightmost derivation
    - By keep replacing current handles and executing reductions, we “prune” our way to the start symbol
- **Shift-Reduce Parsing**
  - $\$$ marks the bottom of the stack and the right side of the input
  - Top of the stack is to the right of $\$$; **handles always appear on top of stack**
  - Four possible actions:
    - *Shift*. Shift the next input symbol onto the top of the stack
    - *Reduce*. Reduce a on the top of the stack to its non-terminal
    - *Accept*. Successful completion of parsing if $\$S$ on stack
    - *Error*. Discover syntax error and call error recovery.
- Example: String \( \text{id}_1 + \text{id}_2 \ast \text{id}_3 \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $</td>
<td>id_1 + id_2 \ast id_3$</td>
<td>None</td>
<td>Shift</td>
</tr>
<tr>
<td>2 $\text{id}_1$</td>
<td>+id_2 \ast id_3$</td>
<td>id_1</td>
<td>Reduce by ( F \rightarrow \text{id} )</td>
</tr>
<tr>
<td>3 $F$</td>
<td>+id_2 \ast id_3$</td>
<td>( F )</td>
<td>Reduce by ( T \rightarrow F )</td>
</tr>
<tr>
<td>4 $T$</td>
<td>+id_2 \ast id_3$</td>
<td>( T )</td>
<td>Reduce by ( E \rightarrow T )</td>
</tr>
<tr>
<td>5 $E$</td>
<td>+id_2 \ast id_3$</td>
<td>None</td>
<td>Shift</td>
</tr>
<tr>
<td>6 $E +$</td>
<td>id_2 \ast id_3$</td>
<td>None</td>
<td>Shift</td>
</tr>
<tr>
<td>7 $E + \text{id}_2$</td>
<td>$\ast \text{id}_3$</td>
<td>( \text{id}_2 )</td>
<td>Reduce by ( F \rightarrow \text{id} )</td>
</tr>
<tr>
<td>8 $E + F$</td>
<td>$\ast \text{id}_3$</td>
<td>( \text{id}_3 )</td>
<td>Reduce by ( T \rightarrow F )</td>
</tr>
<tr>
<td>9 $E + T$</td>
<td>$\ast \text{id}_3$</td>
<td>None</td>
<td>Shift</td>
</tr>
<tr>
<td>10 $E + T \ast$</td>
<td>( \text{id}_3 )</td>
<td>None</td>
<td>Shift</td>
</tr>
<tr>
<td>11 $E + T \ast \text{id}_3$</td>
<td>$\text{id}_3$</td>
<td>( \text{id}_3 )</td>
<td>Reduce by ( F \rightarrow \text{id} )</td>
</tr>
<tr>
<td>12 $E + T \ast F$</td>
<td>$\text{id}_3 \ast F$</td>
<td>Reduce by ( T \rightarrow T \ast F )</td>
<td></td>
</tr>
<tr>
<td>13 $E + T$</td>
<td>$\text{id}_3 \ast F$</td>
<td>Reduce by ( E \rightarrow E + T )</td>
<td></td>
</tr>
<tr>
<td>14 $E$</td>
<td>$\text{id}_3 \ast F$</td>
<td>None</td>
<td>DONE</td>
</tr>
</tbody>
</table>

- Some notes on this table:
  - In line 9, \( T \) is on the right side of the stack, but is not a handle, even though there is a rule \( E \rightarrow T \). It’s not a handle because if the reduction would be executed, \( E + E \) wouldn’t get us anywhere. Also, \( E + T \) is not a handle because replacing it with \( E \) brings us into trouble later.
  - Look at lines 9 and 13; the stack looks the same, but in one you shift, in the other you reduce \( \rightarrow \) Make that decision based on looking at the next input symbol

- The key here is finding the handle
  - We know that handle is on stack and that there are finite number of handles \( \rightarrow \) build DFA
  - LR(1) parsers build a DFA that runs over the stack and finds them, based on one symbol lookahead on input

- LR parsers are table-driven (much like LL(1) parsing table)
  - Can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written
  - No backtracking required
  - Detects a syntactic error as soon as it is possible to do so on a left-to-right scan of input
  - All LL(1) grammars are proper subset of LR(1) grammars
**SLR Parser**

- **Terminology**
  - *LR(0) item*: of grammar $G$ is a production of $G$ with a dot at some position of the body, e.g. $A \to \varepsilon$ yields $A \rightarrow \varepsilon$ and $A \to aBb$ yields:
    
    \[
    A \rightarrow aBb \quad A \to aBb \quad A \to aBb \quad A \to aBb
    \]

    (intuitively, $A \to aBb$ indicates that we have just seen on the input string $a$ and hope next to see a string derivable from $Bb$; $A \to aBb\varepsilon$ indicates we have seen $aBb$ and it may be time to reduce)

  - *Canonical LR(0) collection*: Sets of LR(0) items that provide basis for building DFA and make parsing decision \$ \to$ LR(0) automaton has one state for each item set in canonical LR(0) collection.

  - *Augmented Grammar*: If $G$ is grammar with start symbol $S$, then the augmented grammar $G'$ with a new start symbol $S'$ and production $S' \rightarrow S$

- **Closure of Item Sets**
  - $I$ is set of items

  - CLOSURE($I$) can be computed as follows:
    - Add every item in $I$ to CLOSURE($I$)
    - If $A \to \alpha\beta$ in CLOSURE($I$) and $B \to \gamma$ is a production, then add $B \to \gamma$ to CLOSURE($I$).
    - Apply (2) until no more new items can be added to CLOSUE($I$)

  - Intuition: $A \to \alpha\beta$ indicates that we might next see a substring derivable from $\beta$. Prefix will derive from $B$, that's why we add $B \to \gamma$.

  - In particular we are interested in CLOSUE($\{S' \rightarrow S\}$)

  - Closure lists all handles that might trigger reduction operation
Kernel Items: initial item \( S' \rightarrow S \) and all items whose dots are not at the left end

Nonkernel Items: All items with their dots at the left end, except for \( S' \rightarrow S \).

Nonkernel items don’t need to be saved explicitly as they can be reconstructed by closure operation on kernel items

- **GOTO function**
  - \( \text{GOTO}(I, X) \), where \( I \) is set of items and \( X \) is grammar symbol
  - \( \text{GOTO}(I, X) \) is the closure of the set of all items \( A \rightarrow \alpha X \beta \) such that \( A \rightarrow \alpha X \beta \) is in \( I \)
  - Defines transitions in the LR(0) automaton; states of automaton are sets of items and \( \text{GOTO}(I, X) \) specifies transition from state \( I \) under input \( X \)

- **Canonical LR(0) Collection Construction**
  - Construct set from augmented grammar \( G' \)
  - This constructs states of LR(0) automaton
  - How do these states help us parse?
    - If currently in state \( j \) and we read input \( a \)
      - Shift, if there is a transition in the automaton from state \( j \) on input \( a \)
      - Items in state \( j \) tell us how to reduce otherwise
  - Start state is \( \text{CLOSURE}(\{S' \rightarrow S\}) \)
  - All states are accepting states in DFA
  - Algorithm:
    ```c
    void items(G') {
        C = CLOSURE(\{S' \rightarrow S\});
        repeat
            for ( each set of items I in C )
                for ( each grammar symbol X )
                    if ( GOTO(I, X) is not empty and not in C )
                        add GOTO(I, X) to C;
            until no new sets of items are added to C on a round;
    }
    ```

- **Building SLR-Parsing table**
  - First, build the FIRST and FOLLOW sets for the grammar
  - Then follow algorithm in book pages 325 – 326

- **Worksheet to build SLR parser for simple grammar**