Compiler Development (CMPSC 401)

Syntax Analysis
Bottom-Up Parsing

Janyl Jumadinova

March 5, 2019
**Idea**: Apply productions in reverse to convert the user’s program to the start symbol.
Bottom Up Parsing

\[
\begin{align*}
E & \rightarrow T \\
E & \rightarrow E + T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Bottom Up Parsing

\[
\begin{align*}
E \rightarrow & \ T & \text{int } + (\text{int } + \text{int } + \text{int}) \\
E \rightarrow & \ E + T & \Rightarrow T + (\text{int } + \text{int } + \text{int}) \\
T \rightarrow & \ \text{int} & \Rightarrow E + (\text{int } + \text{int } + \text{int}) \\
T \rightarrow & \ (E) & \Rightarrow E + (T + \text{int } + \text{int}) \\
& & \Rightarrow E + (E + \text{int } + \text{int}) \\
& & \Rightarrow E + (E + T + \text{int}) \\
& & \Rightarrow E + (E + \text{int}) \\
& & \Rightarrow E + (E + T) \\
& & \Rightarrow E + (E) \\
& & \Rightarrow E + T \\
& & \Rightarrow E
\end{align*}
\]
If $S \rightarrow *\alpha$, the string $\alpha$ is called a sentential form of the grammar.

In the derivation $S \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow \ldots \rightarrow \beta_n$, each of the $\beta_i$ are sentential forms.

A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential).
Bottom-Up Parsing

- The **handle** of the right-sentential form is a substring corresponding to the right-hand side of the production that produced it from the previous step in the rightmost derivation.

- Handle can also be represented as the production and the position of the last symbol being replaced.

- A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle.
Finding Handles

- Where do we look for handles?
- How do we search for possible handles?
  - Once we know where to search, how do we identify candidate handles?
- How do we recognize handles?
  - Once we have found a candidate handle, how do we check that it really is the handle?
Shift/Reduce Algorithm

The bottom-up parsers we will consider are called shift/reduce parsers.

Idea: Split the input into two parts:
- Left substring is our work area; all handles must be here.
- Right substring is input we have not yet processed; consists purely of terminals.

At each point, decide whether to:
- Move a terminal across the split (shift)
- Reduce a handle (reduce)
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \ast T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]

\[
\text{int} + \text{int} \ast \text{int} + \text{int}
\]
Shift/Reduce Algorithm

\[ \begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \times T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*} \]
Shift/Reduce Algorithm

\[
\begin{align*}
E &\rightarrow F \\
E &\rightarrow E + F \\
F &\rightarrow F * T \\
F &\rightarrow T \\
T &\rightarrow \text{int} \\
T &\rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \times T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \times T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]

Diagram:

```
  E  +  int  *  int  +  int
    |      |
    F  int |
    |      |
    T     
    |      |
  int   
  |      |
  E
```
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F * T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E &\rightarrow F \\
E &\rightarrow E + F \\
F &\rightarrow F \ast T \\
F &\rightarrow T \\
T &\rightarrow \text{int} \\
T &\rightarrow (E)
\end{align*}
\]
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \text{int}$
$T \rightarrow (E)$
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F * T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \to F \\
E & \to E + F \\
F & \to F * T \\
F & \to T \\
T & \to \text{int} \\
T & \to (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
& E \rightarrow F \\
& E \rightarrow E + F \\
& F \rightarrow F \ast T \\
& F \rightarrow T \\
& T \rightarrow \text{int} \\
& T \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \ast T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
**Shift/Reduce Algorithm**

- $E \rightarrow F$
- $E \rightarrow E + F$
- $F \rightarrow F * T$
- $F \rightarrow T$
- $T \rightarrow \text{int}$
- $T \rightarrow (E)$

![Diagram of shift/reduce algorithm]
Shift/Reduce Algorithm

\[ \begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \ast T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*} \]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F * T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F * T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F * T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \cdot T \\
F & \rightarrow T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Shift/Reduce Algorithm

\[
\begin{align*}
E & \rightarrow F \\
E & \rightarrow E + F \\
F & \rightarrow F \ast T \\
T & \rightarrow \text{int} \\
T & \rightarrow (E)
\end{align*}
\]
Observations

- Since reductions are always at the right side of the left area, we never need to shift from the left to the right.
- No need to “uncover” something to do a reduction.
- Consequently, shift/reduce parsing means
  - **Shift**: Move a terminal from the right to the left area.
  - **Reduce**: Replace some number of symbols at the right side of the left area.
Observations

- All activity in a shift/reduce parser is at the far right end of the left area.
- Idea: Represent the left area as a stack.
Observations

- All activity in a shift/reduce parser is at the far right end of the left area.
- Idea: Represent the left area as a stack.
- **Shift**: Push the next terminal onto the stack.
- **Reduce**: Pop some number of symbols from the stack, then push the appropriate nonterminal.
Finding Handles

- Where do we look for handles?
Finding Handles

- Where do we look for handles?
  - At the top of the stack.

- How do we search for possible handles?
  - Once we know where to search, how do we identify candidate handles?

- How do we recognize handles?
  - Once we have found a candidate handle, how do we check that it really is the handle?
Searching for Handles

- When using a shift/reduce parser, we must decide whether to shift or reduce at each point.
- We only want to reduce when we know we have a handle.
Finding Handles

- Where do we look for handles?
  - At the top of the stack.
- How do we search for possible handles?
Finding Handles

- Where do we look for handles?
  - **At the top of the stack.**

- How do we search for possible handles?
  - **Build a handle-finding automaton.**

- How do we recognize handles?
  - Once we have found a candidate handle, how do we check that it really is the handle?
Our automaton will tell us all places where a handle might be. However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.
Our automaton will tell us all places where a handle might be.

However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.

We will thus use **predictive bottom-up parsing**: *Have a deterministic procedure for guessing where handles are.*

There are many predictive algorithms, each of which recognize different grammars.
LR(1)

- Bottom-up predictive parsing with:
  - **L**: Left-to-right scan
  - **R**: Rightmost derivation
  - (1): One token lookahead

- Tries to intelligently find handles by using a lookahead token at each step.
A Deterministic Automaton

S → E
E → T;
E → T + E
T → int
T → (E)

E → T + E
E → · T;
E → · T + E
T → · int
T → · (E)

S → E ·
E → · T;
E → · T + E
T → · int
T → · (E)

T → (E) ·
T → (E)

T → (E)
E → · T;
E → · T + E
T → · int
T → · (E)

E → T ·
E → T · + E

;
Guess which series of productions we are reversing.
Use this information to maintain information about what lookahead to expect.
When deciding whether to shift or reduce, use lookahead to disambiguate.
How do we know what lookahead to expect at each state?

Observation:
- There are only finitely many productions we can be in at any point.
- There are only finitely many positions we can be in each production.
- There are only finitely many lookahead sets at each point.
How do we know what lookahead to expect at each state?

Observation:
- There are only finitely many productions we can be in at any point.
- There are only finitely many positions we can be in each production.
- There are only finitely many lookahead sets at each point.
- Construct an automaton to track lookaheads!
A Deterministic LR(1) Automata
LR(1) parsers are usually represented via two tables: an action table and a goto table.

The action table maps each state to an action:
- shift, which shifts the next terminal, and
- reduce $A \rightarrow \omega$, which performs reduction
Representing LR(1) Automata

- LR(1) parsers are usually represented via two tables: an action table and a goto table.
- The action table maps each state to an action:
  - shift, which shifts the next terminal, and
  - reduce $A \rightarrow \omega$, which performs reduction
- Any state of the form $A \rightarrow \omega$ does that reduction; everything else shifts.
- The goto table maps state/symbol pairs to a next state.
- This is just the transition table for the automaton.
Representing LR(1) Automata

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>(   )</th>
<th>+</th>
<th>$</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
<td>s2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>ACCEPT</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>r2</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r5</td>
<td></td>
<td></td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td></td>
<td>s7</td>
<td></td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s10</td>
<td>s14</td>
<td></td>
<td>s10</td>
<td>s8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>s9</td>
<td>s12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>r2</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>r4</td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>s11</td>
<td></td>
<td></td>
<td></td>
<td>s13</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>s11</td>
<td>s14</td>
<td></td>
<td>s10</td>
<td>s15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>s16</td>
<td>s12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Production Rules:**

- \( S \rightarrow E \)  
- \( E \rightarrow T \)  
- \( E \rightarrow E + T \)  
- \( T \rightarrow \text{int} \)  
- \( T \rightarrow (E) \)  

(1)  
(2)  
(3)  
(4)  
(5)
LR(1) Algorithm

- Begin with an empty stack and the input set to \( \omega \), where \( \omega \) is the string to parse. Set \textbf{state} to the initial state.
LR(1) Algorithm

- Begin with an empty stack and the input set to ω$, where ω is the string to parse. Set state to the initial state.
- Repeat the following:
  - Let the next symbol of input be t.
  - If action[state,t] is shift, then shift the input and set state=goto[state,t].
  - If action[state,t] is reduce $A \rightarrow ω$:
    - Pop $|ω|$ symbols off the stack; replace them with $A$.
    - Let the state atop the stack be top-state.
    - Set state=goto[top-state,A]
  - If action[state,t] is accept, then the parse is done.
  - If action[state,t] is error, report an error.