Administrivia

- Programming Assignment 1 due on Monday.
- Check class newsgroup cornell.class.cs4120 for answers to frequently asked questions.

Bottom-up parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - construct right-most derivation of program
  - automatic parser generators (e.g., yacc, CUP, ocamlyacc)
  - detect errors as soon as possible
  - allows better error recovery

Top-down parsing

\[ S \rightarrow S + E \mid E \]
\[ E \rightarrow n \mid (S) \]

\[ S \rightarrow S + E \rightarrow E + E \rightarrow (S) + E \rightarrow (S) + E + E \rightarrow (S + E) + E \rightarrow (E + E) + E \rightarrow (1 + E) + E + E \rightarrow (1 + 2 + E) + E \rightarrow (1 + 2 + E) + E \]

- In left-most derivation, entire tree above a token (2) has to be expanded when encountered
- Must be able to predict productions!
Bottom-up parsing

• Right-most derivation -- backward
  – Start with the tokens
  – End with the start symbol

(1+2+(3+4))+5 ← (E+2+(3+4))+5 ← (S+2+(3+4))+5
← (S+E+(3+4))+5 ← (S+(S+4))+5 ← (S+(S+E))+5 ← (S+(S))+5 ← (S+E)+5 ← E+5 ← S+E ← S

Progress of bottom-up parsing

(1+2+(3+4))+5 ← (E+2+(3+4))+5 ← (S+2+(3+4))+5 ← (S+E+(3+4))+5 ← (S+(S+E))+5 ← (S+(S))+5 ← E+5 ← (S+E) ← S

Bottom-up parsing

• (1+2+(3+4))+5 ← (E+2+(3+4))+5
  ← (S+2+(3+4))+5 ← (S+E+(3+4))+5 ...

• Advantage of bottom-up parsing:
  select productions using more information

Top-down vs. Bottom-up

Bottom-up: Don’t need to figure out as much of the parse tree for a given amount of input
Shift-reduce parsing

- Parsing is a sequence of shift and reduce operations
- Parser state is a stack of terminals and non-terminals (grows to the right)
- Unconsumed input is a string of terminals
- Current derivation step is always stack + input

<table>
<thead>
<tr>
<th>Derivation step</th>
<th>stack</th>
<th>unconsumed input</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2+(3+4))+5</td>
<td>(1+2+(3+4))+5</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(E+2+(3+4))+5</td>
<td>(E)</td>
<td>+2+(3+4))+5</td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S)</td>
<td>+2+(3+4))+5</td>
</tr>
<tr>
<td>(S+E+(3+4))+5</td>
<td>(S+E)</td>
<td>+(3+4))+5</td>
</tr>
</tbody>
</table>

Shift-reduce parsing

- Parsing is a sequence of shifts and reduces

<table>
<thead>
<tr>
<th>Shift</th>
<th>move lookahead token to stack. No effect on derivation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>input</td>
</tr>
<tr>
<td>(</td>
<td>1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1</td>
<td>+2+(3+4))+5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reduce</th>
<th>Replace symbols γ in top of stack with non-terminal symbol X, corresponding to production X → γ (pop γ, push X). Reduces rightmost nonterminal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>input</td>
</tr>
<tr>
<td>(S+E)</td>
<td>+(3+4))+5</td>
</tr>
<tr>
<td>(S)</td>
<td>+(3+4))+5</td>
</tr>
</tbody>
</table>

Problem

- How do we know which action to take -- whether to shift or reduce, and which production?
- Sometimes can reduce but shouldn’t. – e.g., X → ε can always be reduced
- Sometimes can reduce in more than one way.
**Action Selection Problem**

- Given stack \( \sigma \) and look-ahead symbol \( b \), should parser:
  - **shift** \( b \) onto the stack (making it \( \sigma b \))
  - **reduce** some production \( X \rightarrow \gamma \) assuming that stack has the form \( \alpha \gamma \) (making it \( \alpha X \))
- If stack has form \( \alpha \gamma \), should apply reduction \( X \rightarrow \gamma \) (or shift) depending on stack prefix \( \alpha \)
  - \( \alpha \) is different for different possible reductions, since \( \gamma \)'s have different length.
  - How to keep track of possible reductions?

**Parser States**

- Goal: know what reductions are legal at any given point.
- Idea: summarize all possible stacks \( \sigma \) (and prefixes \( \alpha \)) as a finite parser **state**
  - Parser state is computed by a DFA that reads in the stack \( \sigma \)
  - Accept states of DFA: unique reduction!
- Summarizing discards information
  - affects what grammars parser handles
  - affects size of DFA (number of states)

**LR(0) parser**

- **Left-to-right scanning**, **Right-most derivation**, “zero” look-ahead characters
- Too weak to handle most language grammars (e.g., “sum” grammar)
- But will help us understand shift-reduce parsing...

**LR(0) states**

- A state is a set of **items** keeping track of progress on possible upcoming reductions
- An **LR(0) item** is a production from the language with a separator “.” somewhere in the RHS of the production

\[ E \rightarrow \text{num}. \]
\[ E \rightarrow (\text {. } S) \]

- Stuff before “.” is already on stack (beginnings of possible \( \gamma \)'s to be reduced)
- Stuff after “.” : what we might see next
- The prefixes \( \alpha \) represented by state itself
An LR(0) grammar: non-empty lists

\[
S \rightarrow (L) \mid id \\
L \rightarrow S \mid L,S \\
x \rightarrow (x,y) \quad (x, (y,z), w) \\
(((x)))) \rightarrow (x, (y, (z, w)))
\]

Start State & Closure

\[
S \rightarrow (L) \mid id \\
L \rightarrow S \mid L,S
\]

Constructing a DFA to read stack:
- First step: augment grammar with prod’n \( S' \rightarrow S \$
- Start state of DFA: empty stack = \( S' \rightarrow S \$
- Closure of a state adds items for all productions whose LHS occurs in an item in the state, just after ‘.’
  - set of possible productions to be reduced next
  - Added items have the ‘.’ located at the beginning: no symbols for these items on the stack yet

Applying terminal symbols

\[
S' \rightarrow S \$
S \rightarrow .(L) \\
L \rightarrow .S \\
S \rightarrow .(L) \\
L \rightarrow .L,S \\
S \rightarrow .(L) \\
S \rightarrow .id
\]

In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, and take closure.

Applying non-terminals

\[
S' \rightarrow S \$
S \rightarrow .(L) \\
L \rightarrow .S \\
S \rightarrow .(L) \\
L \rightarrow .L,S \\
S \rightarrow .(L) \\
S \rightarrow .id
\]

- Non-terminals on stack treated just like terminals (but added by reductions)
Applying reduce actions

- Pop RHS off stack, replace with LHS X (X→γ), rerun DFA (e.g. (x))

Parsing example: ((x),y)

<table>
<thead>
<tr>
<th>derivation</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x),y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>((x),y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>((x),y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 2</td>
</tr>
<tr>
<td>((x),y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>((S),y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>((L),y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>(S,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce S→(L)</td>
</tr>
<tr>
<td>(L,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>(L,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 8</td>
</tr>
<tr>
<td>(L,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 9</td>
</tr>
<tr>
<td>(L,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>(S,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce L→L,S</td>
</tr>
<tr>
<td>(L,y)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>(L)</td>
<td>1, 0</td>
<td>(x),y</td>
<td>reduce S→(L)</td>
</tr>
<tr>
<td>S</td>
<td>1, 0</td>
<td>(x),y</td>
<td>done</td>
</tr>
</tbody>
</table>

Optimization

- Don't need to rerun DFA from beginning on every reduction
- On reducing X→γ with stack αγ:
  - pop γ off stack, revealing prefix α and state
  - take single step in DFA from top state
  - push X onto stack with new DFA state

- final state

( (L) , y )  state = 6
( S , y )  state = ?
Implementation: LR parsing table

- **Action table**: Used at every step to decide whether to shift or reduce.
- **Goto table**: Used only when reducing, to determine next state.

### Action Table
- **Example**: $a \rightarrow X \rightarrow \gamma.$

### Goto Table
- $X$

---

Shift-reduce parsing table

1. **Shift and goto state** $n$
2. **Reduce using** $X \rightarrow \gamma$
   - pop symbols $\gamma$ off stack
   - using state label of top (end) of stack, look up $X$ in goto table and go to that state

- **DFA + stack = push-down automaton (PDA)**

---

List grammar parsing table

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>id</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>$</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>S→id</td>
<td>S→id</td>
<td>S→id</td>
<td>S→id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S→(L)</td>
<td>S→(L)</td>
<td>S→(L)</td>
<td>S→(L)</td>
<td>S→(L)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L→S</td>
<td>L→S</td>
<td>L→S</td>
<td>L→S</td>
<td>L→S</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td>L→L,S</td>
<td></td>
</tr>
</tbody>
</table>

- **Shift-reduce parsing**
  - Grammars can be parsed bottom-up using a DFA + stack
    - DFA processes stack $\sigma$ to decide what reductions might be possible given
    - shift-reduce parser or push-down automaton (PDA)
    - Compactly represented as LR parsing table
  - State construction converts grammar into states that decide action to take
Checkpoints

- Limitations of LR(0) grammars
- SLR, LR(1), LALR parsers
- Automatic parser generators

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action -- in those states, always reduce ignoring lookahead
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Choose based on lookahead.

$$
\begin{align*}
L & \rightarrow L, S. \\
S & \rightarrow S, L.
\end{align*}
$$

List grammar parsing table

<table>
<thead>
<tr>
<th></th>
<th>(   )</th>
<th>id</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S \rightarrow id</td>
<td>S \rightarrow id</td>
<td>S \rightarrow id</td>
<td>S \rightarrow id</td>
<td>s1</td>
<td>g7</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S \rightarrow (L)</td>
<td>S \rightarrow (L)</td>
<td>s5</td>
<td>s6</td>
<td>s7</td>
<td>s8</td>
</tr>
<tr>
<td>6</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L \rightarrow L, S</td>
<td>L \rightarrow L, S</td>
<td>L \rightarrow L, S</td>
<td>L \rightarrow L, S</td>
<td>L \rightarrow L, S</td>
<td>L \rightarrow L, S</td>
</tr>
<tr>
<td>9</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
<td>L \rightarrow S</td>
</tr>
</tbody>
</table>

An LR(0) grammar?

$$
\begin{align*}
S & \rightarrow S + E | E \\
E & \rightarrow \text{num} | ( S )
\end{align*}
$$

- Left-associative: LR(0)
- Right-associative version: not LR(0)

$$
\begin{align*}
S & \rightarrow E + S | E \\
E & \rightarrow \text{num} | ( S )
\end{align*}
$$
LR(0) construction

\[
S \rightarrow E + S \mid E \\
E \rightarrow \text{num} \mid (S)
\]

SLR grammars

- Idea: Only add reduce action to table if lookahead symbol is in the FOLLOW set of the non-terminal being reduced
- Eliminates some conflicts.
- \(FOLLOW(S) = \{\$,\)\}
- Many language grammars are SLR.