Lecture 5

CS 473: COMPILER DESIGN
Program 1 Questions

Top
Predictive Parsing

• Given an LL(1) grammar:
  – For a given nonterminal, the lookahead symbol uniquely determines the production to apply.
  – Top-down/predictive/recursive descent parsing
  – Driven by a parsing table:
    nonterminal and input token → production

$\quad T \rightarrow S\$
$\quad S \rightarrow E S'$
$\quad S' \rightarrow \varepsilon$
$\quad S' \rightarrow + S$
$\quad E \rightarrow \text{number} \mid (S)$

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>$+$</th>
<th>(</th>
<th>)</th>
<th>$$$(EOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$\rightarrow S$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>$\rightarrow E S'$</td>
<td></td>
<td></td>
<td>$\rightarrow E S'$</td>
<td></td>
</tr>
<tr>
<td>$S'$</td>
<td></td>
<td>$\rightarrow + S$</td>
<td></td>
<td>$\rightarrow \varepsilon$</td>
<td>$\rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$E$</td>
<td>$\rightarrow \text{num.}$</td>
<td></td>
<td></td>
<td>$\rightarrow (S)$</td>
<td></td>
</tr>
</tbody>
</table>

• Note: it is convenient to add a special end-of-file token $\$ and a start symbol $T$ (top-level) that requires $\$. 
How do we construct the parse table?

• Consider a given production: $A \rightarrow \gamma$
• Construct the set of all input tokens that may appear *first* in strings that can be derived from $\gamma$
  – Add the production $\rightarrow \gamma$ to the entry $(A,\text{token})$ for each such token.
• If $\gamma$ can derive $\varepsilon$ (the empty string), then we construct the set of all input tokens that may *follow* the nonterminal $A$ in the grammar.
  – Add the production $\rightarrow \gamma$ to the entry $(A, \text{token})$ for each such token.

• Note: if there are two different productions for a given entry, the grammar is not LL(1)
LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser
- Can extend to LL(k) (it just makes the table bigger)

Problems:
- Grammar must be LL(1)
- Can’t do left associativity

```
S → E + S | E
E → number | ( S )
S → ES'
S' → ε
S' → + S
E → number | ( S )
```
**LL(1) Summary**

- Top-down parsing that finds the leftmost derivation.
- Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser.
- Can extend to LL(k) (it just makes the table bigger).

**Problems:**
- Grammar must be LL(1)
- Can’t do left associativity

\[
\begin{align*}
S &\rightarrow S + E \mid E \\
E &\rightarrow \text{number} \mid ( S )
\end{align*}
\]

\[
\begin{align*}
S &\rightarrow ES' \\
S' &\rightarrow \varepsilon \\
S' &\rightarrow + S \\
E &\rightarrow \text{number} \mid ( S )
\end{align*}
\]
LL(1) Summary

• Top-down parsing that finds the leftmost derivation.
• Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser
• Can extend to LL(k) (it just makes the table bigger)

• Problems:
  – Grammar must be LL(1)
  – Can’t do left associativity

\[
\begin{align*}
S & \rightarrow S + E \mid E \\
E & \rightarrow \text{number} \mid (S)
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow S'E \\
S' & \rightarrow \varepsilon \\
S' & \rightarrow S + \\
E & \rightarrow \text{number} \mid (S)
\end{align*}
\]

• Is there a better way?
LR PARSING
Bottom-up Parsing (LR Parsers)

• LR(k) parser:
  – Left-to-right scanning
  – Rightmost derivation
  – k lookahead symbols

• LR grammars are more expressive than LL
  – Can handle left-recursive and right-recursive grammars, virtually all programming languages
  – Easier to express programming language syntax (no left-factoring)

• Technique: “Shift-Reduce” parsers
  – Work bottom-up instead of top-down
  – Construct rightmost derivation of a program in the grammar
  – Used by many parser generators (yacc, bison, CUP, ocamlyacc, merlin, etc.)
Top-down vs. Bottom-up

- Consider the left-recursive grammar:

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

- \((1 + 2 + (3 + 4)) + 5\)

- How much of the tree do we build after seeing \("(1 + 2)"\)?

- In top-down, must be able to guess which productions to use...

Note: ‘(‘ has been scanned but not consumed. Processing it is still pending.

[Diagram of top-down and bottom-up parsing trees]
### Progress of Bottom-up Parsing

<table>
<thead>
<tr>
<th>Reductions</th>
<th>Scanned</th>
<th>Input Remaining</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>(1 + 2 + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>((S + 2 + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>((S + E + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>((S + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4)) + 5 )</td>
</tr>
<tr>
<td>((S + (E + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
<tr>
<td>((S + (S + 4)) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
<tr>
<td>((S + (S + E)) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
<tr>
<td>((S + (S)) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
<tr>
<td>((S + E) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
<tr>
<td>(S + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
<tr>
<td>(S + E )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
<td>(1 + 2 + (3 + 4) + 4) + 5 )</td>
</tr>
</tbody>
</table>

**Rightmost derivation**

\[
S \rightarrow S + E \ | \ E \\
E \rightarrow \text{number} \ | \ (S)
\]
Shift/Reduce Parsing

- **Parser state:**
  - Stack holds term so far (terminals and nonterminals)
  - Remaining input is a string of terminals
  - Look at stack and input to decide next step

- **Parsing is a sequence of shift and reduce operations:**

- **Shift:** move look-ahead token to the stack

- **Reduce:** Replace symbols \( \gamma \) at top of stack with nonterminal \( X \) such that \( X \rightarrow \gamma \) is a production. (pop \( \gamma \), push \( X \))

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((1 + 2 + (3 + 4)) + 5)</td>
<td>shift (</td>
</tr>
<tr>
<td>(</td>
<td>1 + 2 + (3 + 4)) + 5</td>
<td>shift 1</td>
</tr>
<tr>
<td>(1</td>
<td>+ 2 + (3 + 4)) + 5</td>
<td>reduce: ( E \rightarrow \text{number} )</td>
</tr>
<tr>
<td>(E</td>
<td>+ 2 + (3 + 4)) + 5</td>
<td>reduce: ( S \rightarrow E )</td>
</tr>
<tr>
<td>(S</td>
<td>+ 2 + (3 + 4)) + 5</td>
<td>shift +</td>
</tr>
<tr>
<td>(S +</td>
<td>2 + (3 + 4)) + 5</td>
<td>shift 2</td>
</tr>
<tr>
<td>(S + 2</td>
<td>+ (3 + 4)) + 5</td>
<td>reduce: ( E \rightarrow \text{number} )</td>
</tr>
</tbody>
</table>
Questions

Top
simple LR parsing with no lookahead

LR(0) PARSING
LR Parser States

• Given the stack and input, how do we know what action to take?
• Idea: Build a DFA whose states are based on the input seen so far.

• First try: LR(0) parsing
  – Left-to-right scanning, Right-most derivation, zero look-ahead tokens
  – Simplest example of shift-reduce parsing (a bit too simple for the real world)
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

  \[
  S \rightarrow ( L ) \mid \text{id} \\
  L \rightarrow S \mid L, S
  \]

- Example strings:
  - \( x \)
  - \((x, y, z)\)
  - \(((x)))\)
  - \((x, (y, z), w)\)
  - \((x, (y, (z, w)))\)

Parse tree for \((x, (y, z), w)\)
Shift/Reduce Parsing

• Parser state:
  – Stack holds term so far (terminals and nonterminals)
  – Remaining input is a string of terminals
  – Look at stack + input to decide next step

• Parsing is a sequence of *shift* and *reduce* operations:

• **Shift**: move look-ahead token to the stack: e.g.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>(x, (y, z), w)</td>
<td>shift (</td>
</tr>
<tr>
<td>x, (y, z), w</td>
<td></td>
<td>shift x</td>
</tr>
</tbody>
</table>

• **Reduce**: Replace symbols γ at top of stack with nonterminal X such that X ⟹ γ is a production. (pop γ, push X): e.g.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x</td>
<td>, (y, z), w)</td>
<td>reduce S ⟹ id</td>
</tr>
<tr>
<td>(S</td>
<td>, (y, z), w)</td>
<td>reduce L ⟹ S</td>
</tr>
</tbody>
</table>

\[
S \rightarrow (L) \mid \text{id} \\
L \rightarrow S \mid L, S
\]
Action Selection Problem

• Given a stack $\sigma$ and a lookahead symbol $b$, should the parser:
  – Shift $b$ onto the stack (new stack is $\sigma b$)?
  – Reduce by a production $X \rightarrow \gamma$, replacing $\gamma$ with $X$ on the stack?

• Sometimes the parser can reduce but shouldn’t
  – For example, $X \rightarrow \varepsilon$ can always be reduced

• Sometimes the stack can be reduced in multiple different ways

• Main idea: construct a DFA that tracks progress through all the productions we might be in the middle of
LR(0) States

- An LR(0) **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 an extra separator “.” somewhere in the right-hand-side.

```
S \rightarrow ( L ) \mid id
L \rightarrow S \mid L, S
```

- Example items: $S \rightarrow .( L )$ or $S \rightarrow (. L)$ or $L \rightarrow S$.

- Stuff before the ‘.’ is already on the stack (beginnings of possible right-hand sides of productions)
- Stuff after the ‘.’ is what might be seen next

```
S \rightarrow .( L )
S \rightarrow .id
L \rightarrow L, . S
S \rightarrow .( L )
S \rightarrow .id
```
Constructing the DFA: start state & closure

- First step: Add a new production \( T \rightarrow S\$ \) to the grammar
- Start state:

\[
\begin{align*}
T & \rightarrow S\$ \\
S & \rightarrow ( L ) \mid \text{id} \\
L & \rightarrow S \mid L, S
\end{align*}
\]
Constructing the DFA: start state & closure

- First step: Add a new production $T \rightarrow S\$ to the grammar
- Start state:

  $T \rightarrow .S\$
  $S \rightarrow .( L )$
  $S \rightarrow .id$

$T \rightarrow S\$
$S \rightarrow ( L ) \mid id$
$L \rightarrow S \mid L , S$
Constructing the DFA: start state & closure

- First step: Add a new production
  \[ T \rightarrow S$ \] to the grammar

- Start state of the DFA = empty stack, contains the item:
  \[ T \rightarrow .S$ \]

- Closure of a state:
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the ‘.’
  - The added items have the ‘.’ located at the beginning (no symbols for those items have been added to the stack yet)
  - Newly added items might start with nonterminals, so keep iterating until a fixed point is reached

- Example: \( \text{CLOSURE} \{T \rightarrow .S$\} \) = \{T \rightarrow .S$, S \rightarrow (. L ), S \rightarrow .id\}

- Resulting “closed state” contains all possible productions that might be reduced next.
Example: Constructing the DFA

- First, we construct a state with the initial item $T \rightarrow .S$

\[
\begin{align*}
T & \rightarrow S$ \\
S & \rightarrow (L) \ | \ id \\
L & \rightarrow S \ | \ L , S
\end{align*}
\]
Example: Constructing the DFA

- Next, we take the closure of that state:
  \[ \text{CLOSURE}\{\{T \rightarrow .S$\}\} = \{T \rightarrow .S$, S \rightarrow .( L ), S \rightarrow .id\} \]

- In the set of items, the nonterminal S appears after the ‘.’
- So we add items for each S production in the grammar

\[
\begin{align*}
T & \rightarrow S$
S & \rightarrow ( L ) \mid \text{id} \\
L & \rightarrow S \mid L\ , S
\end{align*}
\]
Example: Constructing the DFA

Next we add the transitions:
- Each terminal and nonterminal that can appear after the ‘.’ in the source state gets an outgoing edge.
- The target state includes all items from the source state that have the edge-label symbol after the ‘.’, with the ‘.’ moved past it (modeling shifting the item onto the stack)

T \mapsto S$
S \mapsto (L) \mid \text{id}
L \mapsto S \mid L, S
Finally, for each new state, we take the closure.

\[
\begin{align*}
T & \rightarrow .S$ \\
S & \rightarrow .( L ) \\
S & \rightarrow .id \\
T & \rightarrow S.$ \\
S & \rightarrow ( . L ) \\
S & \rightarrow ( L ) \mid id \\
L & \rightarrow S \mid L , S
\end{align*}
\]
Example: Constructing the DFA

T ⟷ S$
S ⟷ (. L )
S ⟷ .id

T ⟷ S$
S ⟷ ( L ) | id
L ⟷ S | L , S

• Finally, for each new state, we take the closure.
Example: Constructing the DFA

- Finally, for each new state, we take the closure.
- Note that we have to perform two iterations to compute CLOSURE({S ⟷ ( . L )})
  - First iteration adds L ⟷ .S and L ⟷ .L, S
  - Second iteration adds S ⟷ .(L) and S ⟷ .id
Full DFA for the Example

1. T ⟷ S$.  
2. S ⟷ id.  
3. S ⟷ (. L )  
4. S ⟷ .id  
5. S ⟷ ( L . )  
7. L ⟷ S.  
8. L ⟷ L, . S  
9. L ⟷ L, S.

• If a reduce state is reached, reduce
• Otherwise, shift, and then follow the transition for the token shifted
• If no such transition, parse error

Reduce state: ‘.’ at the end of the production

Done!
**Full DFA for the Example**

1. \( T \rightarrow .S$ \)
2. \( S \rightarrow .id \)
3. \( S \rightarrow (. \ L ) \)
4. \( S \rightarrow .id \)
5. \( S \rightarrow .id \)
6. \( S \rightarrow (L) \)
7. \( L \rightarrow S. \)$
8. \( L \rightarrow L, . S \)
9. \( L \rightarrow L, S \)

**Stack: ( id. (state 2)**

**Stack after reducing:** ( S.

1. If a reduce state is reached, reduce
2. Otherwise, shift, and then follow the transition for the token shifted
3. If no such transition, parse error

**Done!**
Full DFA for the Example

- If a reduce state is reached, reduce
- Otherwise, shift, and then follow the transition for the token shifted
- If no such transition, parse error
- After reducing, re-run the DFA on the stack to find the new state

Stack: (id. (state 2)
Stack after reducing: (S. (state 7)
Done!

Stack: ( id. (state 2)
Stack after reducing: ( S. (state 7)
Stack after reducing again: ( L. (state 5)
Using the DFA

• Run the parser stack through the DFA
• The resulting state tells us which productions might be reduced next.
  – If not in a reduce state, then shift the next symbol and transition according to DFA
  – If in a reduce state $X \rightarrow \gamma$ with stack $\alpha\gamma$, pop $\gamma$ and push $X$
  – Then re-run the DFA again

• Optimization: No need to re-run the DFA from beginning every step
  – Store the state with each symbol on the stack: e.g. $13(3L_5)_6$
  – On a reduction $X \rightarrow \gamma$, pop stack to reveal the state too: e.g. From stack $13(3L_5)_6$ reduce $S \rightarrow ( L )$ to reach stack $13$
  – Next, push the reduction symbol: $13S$
  – Then take the corresponding step in the DFA to find next state: $13S_7$
Questions