CS 1622 Lecture 10
 Parsing (5)

---

**LL(1) Parsing Table Example**

- Left-factored grammar
  
  \[
  E \rightarrow T \overline{X} \\
  T \rightarrow ( E ) | \text{int} \ Y \\
  X \rightarrow + E | \varepsilon \\
  Y \rightarrow \ast T | \varepsilon
  \]

- The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T X</td>
<td></td>
<td>+ E</td>
<td>T X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>int Y</td>
<td>+ E</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>Y</td>
<td>* T</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

---

**LL(1) Parsing Table Example (Cont.)**

- Consider the [E, int] entry
  
  - "When current non-terminal is E and next input is int, use production \( E \rightarrow T \overline{X} \)"
  
  - This production can generate an int in the first place

- Consider the [Y, +] entry
  
  - "When current non-terminal is Y and current token is +, get rid of Y" 
  
  - Y can be followed by + only in a derivation in which \( Y \rightarrow \varepsilon \)
LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
- Consider the [E,*] entry
- "There is no way to derive a string starting with * from non-terminal E"

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at [S,a]
  - We use a stack to keep track of pending non-terminals - instead of procedure stack!
  - We reject when we encounter an error state
  - We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] = Y_1...Y_n
      then stack ← <Y_1...Y_n rest>;
      else error ();
    <t, rest> : if t == *next ++
      then stack ← <rest>;
      else error ();
  until stack == < >
```
### LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>T X</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

---

### Constructing Parsing Tables

- **LL(1) languages** are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
- We want to generate parsing tables from CFG

---

### Constructing Parsing Tables (Cont.)

- If $A \rightarrow \alpha$, where in the line of $A$ we place $\alpha$?
- In the column of $t$ where $t$ can start a string derived from $\alpha$:
  - $\alpha \rightarrow^* t \beta$
  - We say that $t \in \text{First}(\alpha)$
- In the column of $t$ if $\alpha \in \varepsilon$ and $t$ can follow an $A$:
  - $S \rightarrow^* \beta A t \delta$
  - We say $t \in \text{Follow}(A)$
Computing First Sets

**Definition**

\[ \text{First}(X) = \{ t | X \to^* t \alpha \} \cup \{ \varepsilon | X \to^* \varepsilon \} \]

**Algorithm:**

- \( \text{First}(t) = \{ t \} \)
- \( \varepsilon \in \text{First}(X) \) if \( X \to \varepsilon \) is a production
- \( \varepsilon \in \text{First}(X) \) if \( X \to A_1 \ldots A_n \)
  - and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
- \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \to A_1 \ldots A_n \alpha \)
  - and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

First Sets. Example

- Recall the grammar
  
  \[
  \begin{align*}
  E &\to T \ X \\
  T &\to ( \ E ) \mid \text{int} \\
  X &\to + \ E \mid \varepsilon \\
  Y &\to * \ T \mid \varepsilon
  \end{align*}
  \]

- First sets
  
  \[
  \begin{align*}
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \text{First}(\ ) &= \{ \} \\
  \end{align*}
  \]

Computing Follow Sets for Nonterminals

- Follow(S) needed for productions that generate the empty string \( \varepsilon \)

**Definition:**

\[ \text{Follow}(X) = \{ t | S \to^* \beta X t \delta \} \]

- Note that \( \varepsilon \) CAN NEVER BE IN FOLLOW(X)!!

**Intuition**

- If \( X \to A \ B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
- Also if \( B \to^* \varepsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
- If \( S \) is the start symbol then \( S \subseteq \text{Follow}(S) \)
Computing Follow Sets (Cont.)

Algorithm sketch:
- $\in \text{Follow}(S)$
- For each production $A \rightarrow \alpha X \beta$
  - $\text{First}(\beta) - \{\epsilon\} \subseteq \text{Follow}(X)$
- For each production $A \rightarrow \alpha X \beta$ where $\epsilon \in \text{First}(\beta)$
  - $\text{Follow}(A) \subseteq \text{Follow}(X)$

Follow Sets. Example
- Recall the grammar
  
  $$
  \begin{align*}
  E & \rightarrow T \, X \\
  T & \rightarrow ( \, E \, ) \mid \text{int} \, Y \\
  X & \rightarrow + \, E \mid \epsilon \\
  Y & \rightarrow * \, T \mid \epsilon
  \end{align*}
  $$

- Follow sets
  
  $$
  \begin{align*}
  \text{Follow}(E) &= \{), \$\} \\
  \text{Follow}(X) &= \{\$, )\} \\
  \text{Follow}(T) &= \{+, ) , \$\} \\
  \text{Follow}(Y) &= \{+, ) , \$\}
  \end{align*}
  $$

Constructing LL(1) Parsing Tables
- Construct a parsing table $T$ for CFG $G$
  
  - For each production $A \rightarrow \alpha$ in $G$ do:
    
    - For each terminal $t \in \text{First}(\alpha)$ do
      
      - $T[A, t] = \alpha$
    
    - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
      
      - $T[A, t] = \alpha$
    
    - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
      
      - $T[A, \$] = \alpha$
Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1). Possible reasons:
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - Grammar is not LL(1)
- Most programming language grammars are not LL(1)
  - Some can be made LL(1) though but others can’t
- There are tools that build LL(1) tables
  - E.g. LLGen

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method in practice
- Concepts first, algorithms next time

An Introductory Example

- Bottom-up parsers don’t need left-factored grammars
- Hence we can revert to the “natural” grammar for our example:
  
  \[
  \begin{align*}
  E & \rightarrow T + E \mid T \\
  T & \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
  \end{align*}
  \]
- Consider the string: int \ast int + int
Bottom-Up Parsing

The Idea

Bottom-up parsing reduces a string to the start symbol by inverting productions:

- int * int + int  \( \rightarrow \) T \( \rightarrow \) int
- int * T + int  \( \rightarrow \) T \( \rightarrow \) int * T
- T + int  \( \rightarrow \) T
- T + T  \( \rightarrow \) E \( \rightarrow \) T
- T + E  \( \rightarrow \) E \( \rightarrow \) T + E
- E

Observation

- Read the productions in parse in reverse (i.e., from bottom to top)
- This is a rightmost derivation!
Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

T + E
E

A Bottom-up Parse in Detail

(1)

int * int + int
A Bottom-up Parse in Detail

(2)
\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times T + \text{int} \]

(3)
\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times T + \text{int} \]
\[ T + \text{int} \]

(4)
\[ \text{int} \times \text{int} + \text{int} \]
\[ \text{int} \times T + \text{int} \]
\[ T + \text{int} \]
\[ T + T \]
A Trivial Bottom-Up Parsing Algorithm

Let I = input string
repeat
    pick a non-empty substring β of I
    where X → β is a production
    if no such β, backtrack
    replace one β by X in I
until I = “S” (the start symbol) or all possibilities are exhausted
Questions

- Does this algorithm terminate?
- How fast is the algorithm?
- Does the algorithm handle all cases?
- How do we choose the substring to reduce at each step?

Where Do Reductions Happen

Important Fact #1 has an interesting consequence:
- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then $\omega$ is a string of terminals

Why? Because $\alpha X \omega \rightarrow \alpha \beta \omega$ is a step in a right-most derivation

Notation

Idea: Split string into two substrings
- Right substring is as yet unexamined by parsing (a string of terminals)
- Left substring has terminals and non-terminals

The dividing point is marked by a $|$ 
- The $|$ is not part of the string

Initially, all input is unexamined $|x_1x_2 \ldots x_n$
Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

- Shift: Move | one place to the right
  - Shifts a terminal to the left string

  \[ \text{ABC|xyz} \Rightarrow \text{ABC|x|yz} \]

Reduce

- Apply an inverse production at the right end of the left string
  - If \( A \rightarrow xy \) is a production, then

  \[ \text{Cbxy|ijk} \Rightarrow \text{CbA|ijk} \]
The Example with Reductions Only

\[
\begin{align*}
\text{int} \ast \text{int} \mid + \text{int} & \quad \text{reduce } T \rightarrow \text{int} \\
\text{int} \ast T \mid + \text{int} & \quad \text{reduce } T \rightarrow \text{int} \ast T \\
T \mid + \text{int} & \quad \text{reduce } T \rightarrow \text{int} \\
T + T & \quad \text{reduce } E \rightarrow \text{int} \ast T \\
T + E & \quad \text{reduce } E \rightarrow T + E \\
E & \\
\end{align*}
\]

The Example with Shift-Reduce Parsing

\[
\begin{align*}
\text{int} \ast \text{int} \mid + \text{int} & \quad \text{shift} \\
\text{int} \mid + \text{int} & \quad \text{shift} \\
\text{int} \ast \mid + \text{int} & \quad \text{shift} \\
\text{int} \ast \text{int} \mid + \text{int} & \quad \text{reduce } T \rightarrow \text{int} \\
\text{int} \ast T \mid + \text{int} & \quad \text{reduce } T \rightarrow \text{int} \ast T \\
T \mid + \text{int} & \quad \text{shift} \\
T + T & \quad \text{reduce } E \rightarrow \text{int} \ast T \\
T + E & \quad \text{reduce } E \rightarrow T + E \\
E & \\
\end{align*}
\]

A Shift-Reduce Parse in Detail (1)

\[
\begin{align*}
\text{int} \ast \text{int} \mid + \text{int} \\
\text{int} \ast \text{int} \mid + \text{int} \\
\end{align*}
\]
A Shift-Reduce Parse in Detail (2)

\[ \text{int} \times \text{int} + \text{int} \]

\[ \text{int} \times \text{int} + \text{int} \]

A Shift-Reduce Parse in Detail (3)

\[ \text{int} \times \text{int} + \text{int} \]

\[ \text{int} \times \text{int} + \text{int} \]

A Shift-Reduce Parse in Detail (4)

\[ \text{int} \times \text{int} + \text{int} \]

\[ \text{int} \times \text{int} + \text{int} \]
A Shift-Reduce Parse in Detail (5)

T
int * int + int
int | * int + int
int * int | + int
int * T | + int

A Shift-Reduce Parse in Detail (6)

T
int * int + int
int | * int + int
int * int | + int
int * T | + int
T | + int

A Shift-Reduce Parse in Detail (7)

T
int * int + int
int | * int + int
int * int | + int
int * T | + int
T | + int
T + | int
A Shift-Reduce Parse in Detail (8)

A Shift-Reduce Parse in Detail (9)

A Shift-Reduce Parse in Detail (10)
A Shift-Reduce Parse in Detail (11)

<table>
<thead>
<tr>
<th>E</th>
<th>T + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>T + int</td>
<td>T + E</td>
</tr>
<tr>
<td>int * int + int</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>* int + int</td>
</tr>
<tr>
<td>int + int</td>
<td>int * int + int</td>
</tr>
<tr>
<td>int + int</td>
<td>int * T</td>
</tr>
<tr>
<td>T</td>
<td>+ int</td>
</tr>
<tr>
<td>T +</td>
<td>int</td>
</tr>
<tr>
<td>T + int</td>
<td>T + E</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Key Issue (will be resolved by algorithms)

- How do we decide when to shift or reduce?
  - Consider step int | * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake: No way to reduce to the start symbol E
Conflicts

- Generic shift-reduce strategy:
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift

- But what if there is a choice?
  - If it is legal to shift or reduce, there is a shift-reduce conflict
  - If it is legal to reduce by two different productions, there is a reduce-reduce conflict

Source of Conflicts

- Ambiguous grammars always cause conflicts

- But beware, so do many non-ambiguous grammars

Conflict Example

Consider our favorite ambiguous grammar:

\[
\begin{align*}
E & \rightarrow E + E \\
& | \quad E \cdot E \\
& | \quad (E) \\
& | \quad \text{int}
\end{align*}
\]
One Shift-Reduce Parse

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>**E</td>
<td>**E + int</td>
</tr>
<tr>
<td>**E * E</td>
<td>+ int</td>
</tr>
<tr>
<td>**E + E</td>
<td></td>
</tr>
<tr>
<td>**E</td>
<td></td>
</tr>
</tbody>
</table>

... reduce **E → **E * **E

shift

Another Shift-Reduce Parse

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>**E</td>
<td>**E + int</td>
</tr>
<tr>
<td>**E * E</td>
<td>+ int</td>
</tr>
<tr>
<td>**E + E</td>
<td></td>
</tr>
<tr>
<td>**E</td>
<td></td>
</tr>
</tbody>
</table>

... shift

Example Notes

- In the second step **E * E | + int we can either shift or reduce by **E → **E * **E

- Choice determines associativity of + and *

- As noted previously, grammar can be rewritten to enforce precedence

- Precedence declarations are an alternative
Precedence Declarations Revisited

- Precedence declarations cause shift-reduce parsers to resolve conflicts in certain ways.
- Declaring "*" has greater precedence than "+" causes parser to reduce at \( E \ast E \mid + \) int.
- More precisely, precedence declaration is used to resolve conflict between reducing a "*" and shifting a "+".

Precedence Declarations Revisited (Cont.)

- The term "precedence declaration" is misleading.
- These declarations do not define precedence; they define conflict resolutions.
  - Not quite the same thing!