Last Lecture

- Context free grammars
  - Derivations
  - Parse trees
  - Ambiguity
  - Associativity & precedence
  - Designing grammars

This Lecture

- Parsing
  - Recursive descent
  - FIRST sets
- Rewriting grammars
  - Left factoring
  - Eliminating left recursion
- Abstract syntax trees (ASTs)

Steps of Compilation

Recursive Descent Parsing

- Goal
  - Determine if we can produce the string to be parsed from the grammar’s start symbol
- Approach
  - Recursively replace nonterminal with RHS of production
- At each step, we’ll keep track of two facts
  - What tree node are we trying to match?
  - What is the lookahead (next token of the input string)?
    - Helps guide selection of production used to replace nonterminal
Recursion descent parsing (cont.)

- At each step, 3 possible cases
  - If we're trying to match a terminal
    - If the lookahead is that token, then succeed, advance the lookahead, and continue
  - If we're trying to match a nonterminal
    - Pick which production to apply based on the lookahead
  - Otherwise fail with a parsing error

Parsing example

\[
E \rightarrow id = n | \{ L \} \\
L \rightarrow E ; L | \epsilon
\]

- One input might be
  \[
  \{ x = 3 ; \{ y = 4 ; \} ; \}
  \]
  - This would get turned into a list of tokens
    \[
    \{ x = 3 ; \{ y = 4 ; \} ; \}
    \]
  - And we want to turn it into a parse tree

Recursive descent parsing (cont.)

- Key step
  - Choosing which production should be selected
- Two approaches
  - Backtracking
    - Choose some production
    - If fails, try different production
    - Parse fails if all choices fail
  - Predictive parsing
    - Analyze grammar to find FIRST sets for productions
    - Compare with lookahead to decide which production to select
    - Parse fails if lookahead does not match FIRST

First sets

- Motivating example
  - The lookahead is \( x \)
  - Given grammar \( S \rightarrow xyz \mid abc \)
    - Select \( S \rightarrow xyz \) since 1st terminal in RHS matches \( x \)
  - Given grammar \( S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z \)
    - Select \( S \rightarrow A \), since \( A \) can derive string beginning with \( x \)
- In general
  - Choose a production that can derive a sentential form beginning with the lookahead
  - Need to know what terminal may be \( \text{first} \) in any sentential form derived from a nonterminal / production

First sets - definition

- \( \text{First}(y) \), for any terminal or nonterminal \( y \), is the set of initial terminals of all strings that \( y \) may expand to
- We'll use this to decide what production to apply

Examples

- Given grammar \( S \rightarrow xyz \mid abc \)
  - \( \text{First}(xyz) = \{ x \} \), \( \text{First}(abc) = \{ a \} \)
  - \( \text{First}(S) = \text{First}(xyz) \cup \text{First}(abc) = \{ x, a \} \)
- Given grammar \( S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z \)
  - \( \text{First}(x) = \{ x \} \), \( \text{First}(y) = \{ y \} \), \( \text{First}(A) = \{ x, y \} \)
  - \( \text{First}(z) = \{ z \} \), \( \text{First}(B) = \{ z \} \)
  - \( \text{First}(S) = \{ x, y, z \} \)
Calculating First(\(\gamma\))

- For a terminal \(a\)
  - \(\text{First}(a) = \{a\}\)
- For a nonterminal \(N\)
  - If \(N \rightarrow \epsilon\), then add \(\epsilon\) to \(\text{First}(N)\)
  - If \(N \rightarrow a_1 a_2 \ldots a_n\), then (note the \(a_i\) are all the symbols on the right side of one single production):
    - Add \(\text{First}(a_1 a_2 \ldots a_k)\) to \(\text{First}(N)\), where \(\text{First}(a_1 a_2 \ldots a_k)\) is defined as
      - \(\text{First}(a_i)\) if \(\epsilon \notin \text{First}(a_i)\)
      - Otherwise \((\text{First}(a_1) - \epsilon) \cup \text{First}(a_2 \ldots a_k)\)
    - If \(\epsilon \in \text{First}(a_i)\) for all \(i\), \(1 \leq i \leq k\), then add \(\epsilon\) to \(\text{First}(N)\)

Parser Implementation (cont.)

- For terminals, create function \(\text{match}(a)\)
  - If lookahead is \(a\) it consumes the lookahead by advancing the lookahead to the next token, and returns
  - Otherwise fails with a parse error if lookahead is not \(a\)
- In algorithm descriptions, consider \(\text{parse}_{-a}\), \(\text{parse}_{-\text{term}}(a)\) to be aliases for \(\text{match}(a)\)
- For each nonterminal \(N\), create a function \(\text{parse}_{-N}\)
  - Called when we're trying to parse a part of the input which corresponds to (or can be derived from) \(N\)
  - \(\text{parse}_{-S}\) for the start symbol \(S\) begins the parse

Parser Implementation (cont.)

- Parse is built on procedure calls
- Procedures may be (mutually) recursive
Recursive Descent Parser

- Given grammar $S \rightarrow A \mid B$, $A \rightarrow x \mid y$, $B \rightarrow z$
  - First($A$) = { $x$, $y$ }, First($B$) = { $z$ }

Parser

```cpp
parse_S() {
    if (lookahead == "x") {
        match("x"); // A \rightarrow x
        parse_A();
    } else if (lookahead == "y") {
        match("y"); // A \rightarrow y
        parse_B();
    } else error();
}
```

Example

- $E \rightarrow \text{id} = n \mid \{ L \}$
- First($E$) = { id, "{" }
- Parse $E()$

<table>
<thead>
<tr>
<th>L \rightarrow E ; L</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_E() {</td>
<td></td>
</tr>
</tbody>
</table>
|   if (lookahead == "id") {
|     match("id");
|     // E \rightarrow \text{id} = n
|   } else if (lookahead == "{") {
|     match("{");
|     // E \rightarrow \{ L \}
|   } else error();
| } |

Things to Notice

- If you draw the execution trace of the parser
  - You get the parse tree

Examples

- Grammar
  - $S \rightarrow A \mid B$
  - $A \rightarrow x \mid y$
  - $B \rightarrow z$
- String "xyz"

```cpp
parse_S() {
    match("x");
    match("y");
    match("z");
}
```

- Grammar
  - $S \rightarrow A \mid B$
  - $A \rightarrow x \mid y$
  - $B \rightarrow z$
- String "x" y z

```cpp
parse_A() {
    match("x");
    parse_S();
}
```

…

Left Factoring

- Consider parsing the grammar $E \rightarrow ab \mid ac$
  - First($ab$) = $a$
  - First($ac$) = $a$
  - Parser cannot choose between RHS based on lookahead!
  - Parser fails whenever $A \rightarrow α_1 \mid α_2$ and
    - First($α_1$) \cap First($α_2$) = \emptyset or ∅
- Solution
  - Rewrite grammar using left factoring

Left Factoring Algorithm

- Given grammar
  - $A \rightarrow xα_1 \mid xα_2 \ldots \mid xα_n \mid β$
- Rewrite grammar as
  - $A \rightarrow xL \mid β$
  - $L \rightarrow α_1 \mid α_2 \ldots \mid α_n$
- Repeat as necessary

Examples

- $S \rightarrow \text{ab} \mid \text{ac}$
- $S \rightarrow \text{abcA} \mid \text{abB} \mid \text{a}$
- $L \rightarrow \text{bcA} \mid \text{bB} \mid \varepsilon$
- $L \rightarrow \text{bcA} \mid \text{bB} \mid \varepsilon$
- $L \rightarrow \{ \text{bL'} \mid \varepsilon \}$, $L' \rightarrow \text{cA} \mid B$

Example (cont.)

- This is a predictive parser
  - Because the lookahead determines exactly which production to use
- This parsing strategy may fail on some grammars
  - Possible infinite recursion
  - Production First sets overlap
  - Production First sets contain ε
- Does not mean grammar is not usable
  - Just means this parsing method not powerful enough
  - May be able to change grammar

<table>
<thead>
<tr>
<th>parse_S() {</th>
<th></th>
</tr>
</thead>
</table>
| if (lookahead == "x") {
|   match("x");
|   parse_A();
| } |

<table>
<thead>
<tr>
<th>parse_A() {</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>match(&quot;x&quot;);</td>
<td></td>
</tr>
<tr>
<td>parse_S();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parse_B() {</th>
<th></th>
</tr>
</thead>
</table>
| if (lookahead == "y") {
|   match("y");
|   parse_B();
| } |

<table>
<thead>
<tr>
<th>parse_B() {</th>
<th></th>
</tr>
</thead>
</table>
| if (lookahead == "z") {
|   match("z");
|   parse_B();
| } |

<table>
<thead>
<tr>
<th>parse_A() {</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>match(&quot;x&quot;);</td>
<td></td>
</tr>
<tr>
<td>parse_S();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Left Recursion

- Consider grammar \( S \rightarrow Sa \mid \varepsilon \)
- First(\( Sa \)) = \( a \), so we're ok as far as which production
- Try writing parser

\[
\begin{align*}
\text{parse}_S() & \\
\text{if (lookahead} \rightarrow \text{ "a")} & \\
\text{parse}_S(); & \text{// } S \rightarrow Sa \\
\text{else} & \\
\end{align*}
\]

- Body of \( \text{parse}_S() \) has an infinite loop
- If (lookahead = "a") then \( \text{parse}_S() \)
- Infinite loop occurs in grammar with left recursion

Right Recursion

- Consider grammar \( S \rightarrow aS \mid \varepsilon \)
- Again, First(\( aS \)) = \( a \)
- Try writing parser

\[
\begin{align*}
\text{parse}_S() & \\
\text{if (lookahead} \rightarrow \text{ "a")} & \\
\text{match("a")}; & // S \rightarrow aS \\
\text{else} & \\
\end{align*}
\]

- Will \( \text{parse}_S() \) infinite loop?
  - Invoking \( \text{match()} \) will advance lookahead, eventually stop
- Top down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  - \( A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \beta \)
    - Why must \( \beta \) exist?
- Rewrite grammar as
  - \( A \rightarrow \beta L \)
  - \( L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \varepsilon \)
- Replaces left recursion with right recursion
- Repeat as necessary

Expr Grammar for Top-Down Parsing

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow \varepsilon \mid + E \\
T & \rightarrow P T' \\
T' & \rightarrow \varepsilon \mid * T \\
P & \rightarrow n \mid (E) \\
\end{align*}
\]

- Notice we can always decide what production to choose with only one symbol of lookahead

Eliminating Left Recursion (cont.)

- Examples
  - \( S \rightarrow Sa \mid \varepsilon \Rightarrow S \rightarrow L \rightarrow aL \mid \varepsilon \)
  - \( S \rightarrow Sa \mid Sb \mid c \Rightarrow S \rightarrow cL \rightarrow aL \mid bl \mid \varepsilon \)

- May need more powerful algorithms to eliminate mutual recursion leading to left recursion
  - \( S \rightarrow Aa \mid b \)
  - \( A \rightarrow Sb \)

Tradeoffs with Other Approaches

- Recursive descent parsers are easy to write
  - The formal definition is a little clunky, but if you follow the code then it's almost what you might have done if you weren't told about grammars formally
  - They're unable to handle certain kinds of grammars
- Recursive descent is good for a simple parser
  - Though tools can be fast if you're familiar with them
- Can implement top-down predictive parsing as a table-driven parser
  - By maintaining an explicit stack to track progress
Tradeoffs with Other Approaches

- More powerful techniques need tool support
  - Can take time to learn tools
- Main alternative is bottom-up, shift-reduce parser
  - Replaces RHS of production with LHS (nonterminal)
  - Example grammar
    - \( S \rightarrow aA, A \rightarrow Bc, B \rightarrow b \)
  - Example parse
    - \( abc \rightarrow aBc \rightarrow aA \rightarrow S \)
  - Derivation happens in reverse
  - Something to look forward to in CMSC 430

What's Wrong With Parse Trees?

- Parse trees contain too much information
  - Example
    - Parentheses
    - Extra nonterminals for precedence
  - This extra stuff is needed for parsing
- But when we want to reason about languages
  - Extra information gets in the way (too much detail)

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts

Producing an AST

- To produce an AST, we can modify the `parse()` functions to construct the AST along the way
  - `match(a)` returns an AST node (leaf) for a
  - `Parse_A` returns an AST node for A
    - AST nodes for RHS of production become children of LHS node
- Example
  - \( S \rightarrow aA \)

Abstract Syntax Trees (cont.)

- Intuitively, ASTs correspond to the data structure you’d use to represent strings in the language
  - Note that grammars describe trees
  - So do OCaml datatypes (which we’ll see later)
  - \( E \rightarrow a \mid b \mid c \mid E + E \mid E - E \mid E^* E \mid (E) \)

The Compilation Process
Summary

- Learned a little about parsing
  - Recursive descent parser
  - Predictive parsing using FIRST sets
- Rewriting grammars for predicative parsing
  - Left factoring
  - Eliminating left recursion
- Abstract syntax trees (ASTs)