CMSC 330: Organization of Programming Languages

Reminders
- Quiz #1 tomorrow!
  - Start at 9:30am
  - Programming languages, Ruby, RE/FA
- Midterm 1 next Wednesday (June 23)
  - Everything through today’s lecture
- Project 2 due next Friday (June 25) at midnight
- Practice problems posted for CFGs and parsing

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

Parsing

Many efficient techniques for parsing
  • i.e., turning strings into parse trees
  • Examples
    ➢ LL(k), SLR(k), LR(k), LALR(k),...
    ➢ Take CMSC 430 for more details

One simple technique: recursive descent parsing
  • This is a "top-down" parsing algorithm

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

Recursive 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

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

- Here \( n \) is an integer and \( \text{id} \) is an identifier

- 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 \( S \rightarrow A \mid B \rightarrow x \mid y \rightarrow z \)
    - 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 first in any sentential form derived from a nonterminal / production
First Sets

**Definition**
- **First(\(\gamma\))**, for any terminal or nonterminal \(\gamma\), is the set of initial terminals of all strings that \(\gamma\) may expand to.
- We’ll use this to decide what production to apply.

**Examples**
- **Given grammar** \(S \rightarrow x y z | a b c\)
  - First(xyz) = \{ x \}
  - First(abc) = \{ a \}
  - First(S) = First(xyz) U First(abc) = \{ x, a \}
- **Given grammar** \(S \rightarrow A | B \quad A \rightarrow x | y \quad B \rightarrow z\)
  - First(x) = \{ x \}
  - First(y) = \{ y \}
  - First(A) = \{ x, y \}
  - First(z) = \{ z \}
  - First(B) = \{ z \}
  - First(S) = \{ x, y, z \}

Calculating First(\(\gamma\))

- **For a terminal** \(a\)
  - First(a) = \{ a \}

- **For a nonterminal** \(N\)
  - If \(N \rightarrow \epsilon\), then add \(\epsilon\) to First(N).
  - If \(N \rightarrow \alpha_1 \alpha_2 ... \alpha_n\), then (note the \(\alpha_i\) are all the symbols on the right side of one single production):
    - Add First(\(\alpha_1 \alpha_2 ... \alpha_n\)) to First(N), where First(\(\alpha_1 \alpha_2 ... \alpha_n\)) is defined as:
      - First(\(\alpha_i\)) if \(\epsilon \notin \text{First}(\alpha_i)\)
      - Otherwise (First(\(\alpha_i\)) – \(\epsilon\)) \(\cup\) First(\(\alpha_2 ... \alpha_n\))
    - If \(\epsilon \in \text{First}(\alpha_i)\) for all \(1 \leq i \leq k\), then add \(\epsilon\) to First(N).

First( ) Examples

<table>
<thead>
<tr>
<th>Grammar</th>
<th></th>
<th>First(id)</th>
<th>First(&quot;=&quot;)</th>
<th>First(&quot;=&quot;)</th>
<th>First(&quot;=&quot;)</th>
<th>First(n)</th>
<th>First(n)</th>
<th>First(&quot;=&quot;)</th>
<th>First(&quot;=&quot;)</th>
<th>First(&quot;=&quot;)</th>
<th>First(&quot;=&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → id = n</td>
<td></td>
<td>{ id }</td>
<td>{ &quot;=&quot; }</td>
<td>{ &quot;=&quot; }</td>
<td>{ &quot;=&quot; }</td>
<td>{ n }</td>
<td>{ n }</td>
<td>{ &quot;=&quot; }</td>
<td>{ &quot;=&quot; }</td>
<td>{ &quot;=&quot; }</td>
<td>{ &quot;=&quot; }</td>
</tr>
<tr>
<td>L → E ; L</td>
<td>(\epsilon)</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
<td>{ L }</td>
</tr>
<tr>
<td>First(E)</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
</tr>
<tr>
<td>First(L)</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
<td>{ id, &quot;=&quot; }</td>
</tr>
</tbody>
</table>

Recursive Descent Parser Implementation

- **For terminals**, create function `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 `parse_a`, `parse_term(a)` to be aliases for `match(a)`

- **For each nonterminal** \(N\), create a function `parse_N`
  - Called when we’re trying to parse a part of the input which corresponds to (or can be derived from) \(N\)
  - `parse_S` for the start symbol \(S\) begins the parse
Parser Implementation (cont.)

The body of parse_N for a nonterminal N does the following:

- Let $N \rightarrow \beta_1 | ... | \beta_k$ be the productions of $N$
  - Here $\beta_i$ is the entire right side of a production - a sequence of terminals and nonterminals
- Pick the production $N \rightarrow \beta_i$ such that the lookahead is in First($\beta_i$)
  - It must be that First($\beta_i$) \cap First($\beta_j$) = \notin for $i \neq j$
  - If there is no such production, but $N \rightarrow \varepsilon$ then return
  - Otherwise fail with a parse error
- Suppose $\beta_i = \alpha_1 \alpha_2 ... \alpha_n$. Then call parse_{\alpha_1}(...; parse_{\alpha_n} to match the expected right-hand side, and return

Recursive Descent Parser Example 1

Given grammar $S \rightarrow xyz | abc$
- First(xyz) = \{ x \}, First(abc) = \{ a \}$

Parser

```c
parse_S() {
    if (lookahead == "x") {
        match("x"); match("y"); match("z"); // S \rightarrow xyz
    } else if (lookahead == "a") {
        match("a"); match("b"); match("c"); // S \rightarrow abc
    } else error();
}
```

Recursive Descent Parser Example 2

Grammar $S \rightarrow A | B$ $A \rightarrow x | y$ $B \rightarrow z$
- First(S) = \{ x, y, z \}, First(A) = \{ x, y \}, First(B) = \{ z \}$

Parser

```c
parse_S() {
    if (lookahead == "x") || (lookahead == "y")
        parse_A(); // S \rightarrow A
    else if (lookahead == "z")
        parse_B(); // S \rightarrow B
    else error();
}
```

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

```c
parse_B() {
    if (lookahead == "z")
        match("z"); // B \rightarrow z
    else error();
}
```
Recursive Descent Parser Example 3

**Grammar**

\[ E \rightarrow \text{id} = n \mid \{ L \} \quad L \rightarrow E ; L \mid \epsilon \]

- **First(E)** = \{ id, "{" \}, First(L) = \{ id, "{" \}, \epsilon \}

**Parser**

```
parse_E() {
    if (lookahead == "id") {
        match("id");
        match("=");
        // E → id = n
        match("n");
    } else if (lookahead == ")") {
        match(";");
        // L → E ; L
        parse_L();
    } else error();
}
```

**Things to Notice**

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

**Examples**

- **Grammar**
  
  \[ S \rightarrow \text{xyz} \]
  
  \[ S \rightarrow \text{abc} \]
  
  **String** "xyz"

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

- **Grammar**
  
  \[ S \rightarrow \text{A} \mid \text{B} \]
  
  \[ A \rightarrow \text{x} \mid \text{y} \]
  
  \[ B \rightarrow \text{z} \]
  
  **String** "x"

```java
parse_A() {
    match("x");
    match("z");
}
```

**Left Factoring**

- Consider parsing the grammar
  
  \[ E \rightarrow \text{ab} \mid \text{ac} \]

- **First(ab)** = a
- **First(ac)** = a
- **Parser cannot choose between RHS based on lookahead!**

- **Parser fails whenever** \[ A \rightarrow \alpha_1 \mid \alpha_2 \] and
  
  - **First(\alpha_1) \cap First(\alpha_2) !\subseteq \epsilon or \emptyset**

- **Solution**
  
  - **Rewrite grammar using left factoring**
Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 | x\alpha_2 | ... | x\alpha_n | \beta$

- Rewrite grammar as
  - $A \rightarrow xL | \beta$
  - $L \rightarrow \alpha_1 | \alpha_2 | ... | \alpha_n$

- Repeat as necessary

- Examples
  - $S \rightarrow ab | ac \Rightarrow S \rightarrow aL \quad L \rightarrow b | c$
  - $S \rightarrow abcA | abB | a \Rightarrow S \rightarrow aL \quad L \rightarrow bcA | bb | \epsilon$
  - $L \rightarrow bcA | bB | \epsilon \Rightarrow L \rightarrow bL' | \epsilon \quad L' \rightarrow cA | B$

Left Recursion

- Consider grammar $S \rightarrow Sa | \epsilon$
  - $\text{First}(Sa) = a$, so we're ok as far as which production
  - Try writing parser
    ```
    parse_S() {
        if (lookahead == "a") {
            parse_S();
            match("a"); if $S \rightarrow Sa$
        } else {
            }
    }
    ```

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

Right Recursion

- Consider grammar $S \rightarrow aS | \epsilon$
  - Again, $\text{First}(aS) = a$
  - Try writing parser
    ```
    parse_S() {
        if (lookahead == "a") {
            match("a"); parse_S(); if $S \rightarrow aS$
        } else {
            }
    }
    ```

  - Will parse_S() infinite loop?
    - Invoking match() will advance lookahead, eventually stop
  - Top-down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  - $A \rightarrow x\alpha_1 | x\alpha_2 | ... | x\alpha_n | \beta$
    - Why must $\beta$ exist?

- Rewrite grammar as
  - $A \rightarrow \beta L$
  - $L \rightarrow \alpha_1 L | \alpha_2 L | ... | \alpha_n L | \epsilon$

- Replaces left recursion with right recursion
- Repeat as necessary
Eliminating Left Recursion (cont.)

- Examples
  - $S \rightarrow Sa | \varepsilon \quad 
    \Rightarrow S \rightarrow LA \quad L \rightarrow aL | \varepsilon$
  - $S \rightarrow Sa | Sb | c \quad 
    \Rightarrow S \rightarrow cL \quad L \rightarrow aL | bL | \varepsilon$

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

Expr Grammar for Top-Down Parsing

- $E \rightarrow T E'$
- $E' \rightarrow \varepsilon | + E$
- $T \rightarrow P \ T'$
- $T' \rightarrow \varepsilon | * T$
- $P \rightarrow n | (E)$

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

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

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

Example

```
E → a | b | c | E+E | E-E | E*E | (E)
```

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 → aA

Node parse_S( ) {
  Node n1, n2;
  if (lookahead == “a”) {
    n1 = match(“a”);
    n2 = parse_A();
    return new Node(n1,n2);
  }
}
```
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)