CMSC 330: Organization of Programming Languages

Parsing

Last Lecture
- Context free grammars
  - Leftmost & rightmost derivations
  - Ambiguity
  - Designing grammars
  - Associativity & precedence

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

Steps of Compilation
- 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

Lexing → Parsing → Intermediate Code Generation → Optimization
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 → id = n | { L }
L → E ; L | ε

- Here n is an integer and 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 → xyz | abc
    - Select S → xyz since 1st terminal in RHS matches x
  - Given grammar S → A | B A → x | y B → z
    - Select S → 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(γ), for any terminal or nonterminal γ, is the set of initial terminals of all strings that γ may expand to
  - We’ll use this to decide what production to apply

- Examples
  - Given grammar S → xyz | abc
    - First(xyz) = { x }, First(abc) = { a }
    - First(S) = First(xyz) U First(abc) = { x, a }
  - Given grammar S → A | B A → x | y B → z
    - First(x) = { x }, First(y) = { y }, First(A) = { x, y }
    - First(z) = { z }, First(B) = { z }
    - First(S) = { x, y, z }
Calculating First(γ)

- For a terminal a
  - First(a) = { a }
- For a nonterminal N
  - If N → ε, then add ε to First(N)
  - If N → α₁ α₂ ... αₙ, then (note the αᵢ are all the symbols on the right side of one single production):
    - First(αᵢ) if ε ∉ First(αᵢ)
    - Otherwise (First(αᵢ) – ε) ∪ First(αₙ ... αₙ)
  - If ε ∈ First(αᵢ) for all i, 1 ≤ i ≤ k, then add ε to First(N)

First( ) Examples

E → id = n | { L }  
L → E ; L | ε
First(id) = { id }  
First("=") = { "=" }  
First(n) = { n }  
First("{") = { "{" }  
First("}"") = { "}" }  
First(""); = { "," }  
First(L) = { id, "," }  
First(L) = { id, "{", "," }

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 → β₁ | ... | βₖ be the productions of N
    - Here βᵢ is the entire right side of a production- a sequence of terminals and nonterminals
  - Pick the production N → βᵢ such that the lookahead is in First(βᵢ)
    - It must be that First(βᵢ) ∩ First(βⱼ) = ∅ for i ≠ j
    - If there is no such production, but N → ε then return
    - Otherwise fail with a parse error
  - Suppose βᵢ = α₁ α₂ ... αₙ. Then call parse_α₁(); ... ; parse_αₙ() to match the expected right-hand side, and return

Recursive Descent Parser

- Given grammar S → xyx | abc
  - First(abc) = { a }
- Parser
  
```java
  parse_S() {
    if (lookahead == "x") {
      match("x"); match("y"); match("z"); // S → xyx
    } else if (lookahead == "a") {
      match("a"); match("b"); match("c"); // S → abc
    } else error();
  }
```
Recursive Descent Parser

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

Parser

- \( \text{parse}_S( ) \) {
  if (lookahead == "x") {
    \( \text{parse}_A( ) \); // \( S \rightarrow A \)
  } else if (lookahead == "z") {
    \( \text{parse}_B( ) \); // \( S \rightarrow B \)
  } else error();
}

Example

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

- \( L \rightarrow E ; L \ | \ \varepsilon \)
  \( \text{parse}_E( ) \) {
    if (lookahead == "id") {
      match("id"); // \( E \rightarrow \text{id} \)
      match("="); // \( \text{id} = n \)
      match("n");
    } else if (lookahead == ")") {
      match("\}"); // \( E \rightarrow \\}
    } 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"
  - \( \text{parse}_S( ) \) {
    match("x"); // \( S \rightarrow x \)
    match("y"); // \( S \rightarrow y \)
    match("z"); // \( S \rightarrow z \)
  }

- Grammar
  - \( S \rightarrow A \ | B \)
  - \( A \rightarrow x \ | y \)
  - \( B \rightarrow z \)
- String "x"
  - \( \text{parse}_A( ) \) {
    match("x"); // \( A \rightarrow x \)
  }

Left Factoring

- Consider parsing the grammar \( E \rightarrow ab \ | \ ac \)
  - First(ab) = a
  - First(ac) = a
  - Parser cannot choose between RHS based on lookahead!
- Parser fails whenever \( A \rightarrow \alpha_1 \ | \ \alpha_2 \) and
  - First(\( \alpha_1 \)) \cap First(\( \alpha_2 \)) = \{ \varepsilon \} or \( \emptyset \)
- Solution
  - Rewrite grammar using left factoring

Left Factoring Algorithm

- Given grammar
  - \( A \rightarrow x \alpha_1 \ | \ x \alpha_2 \ | \ldots \ | x \alpha_n \ | \beta \)
  - Rewrite grammar as
    - \( A \rightarrow xL \ | \ \beta \)
    - \( L \rightarrow \alpha_1 \ | \alpha_2 \ | \ldots \ | \alpha_n \)
- Repeat as necessary
- Examples
  - \( S \rightarrow ab \ | \ ac \quad \varepsilon \rightarrow S \rightarrow aL \quad L \rightarrow b \ | \ c \)
  - \( S \rightarrow abcA \ | \ abB \ | \ a \quad \varepsilon \rightarrow S \rightarrow aL \quad L \rightarrow bcA \ | \ bb \ | \ \varepsilon \)
  - \( L \rightarrow bcA \ | \ bb \ | \ \varepsilon \quad \varepsilon \rightarrow L \rightarrow bL' \ | \ \varepsilon \quad L' \rightarrow cA \ | \ B \)
Left Recursion

Consider grammar \( S \rightarrow S a | \varepsilon \)
- \( \text{First}(S a) = a \) so we're ok as far as which production
- Try writing parser

```java
parse_S() {
    if (lookahead == "a") {
        parse_S();
        match("a"); // S \rightarrow Sa
    } else {
    }
}
```
- 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 \( S \rightarrow aS | \varepsilon \)
- Again, \( \text{First}(aS) = a \)
- Try writing parser

```java
parse_S() {
    if (lookahead == "a") {
        match("a");
        parse_S(); // S \rightarrow aS
    } else {
    }
}
```
- 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
  - \( A \rightarrow A \alpha_1 | A \alpha_2 | ... | A \alpha_n | \beta \)
  - Why must \( \beta \) exist?
- Rewrite grammar as
  - \( A \rightarrow \beta L \)
  - \( L \rightarrow \alpha_1 L | \alpha_2 L | ... | \alpha_n L | \varepsilon \)
- Replaces left recursion with right recursion
- Repeat as necessary

Eliminating Left Recursion (cont.)

- Examples
  - \( S \rightarrow Sa | \varepsilon \)  \( \iff S \rightarrow L \ L \rightarrow aL | \varepsilon \)
  - \( S \rightarrow Sa | Sb | c \)  \( \iff S \rightarrow cL \ L \rightarrow aL | bL | \varepsilon \)
- May need more powerful algorithms to eliminate mutual recursion leading to left recursion
  - \( S \rightarrow Aa | b \)
  - \( A \rightarrow Sb \)

Expr Grammar for Top-Down Parsing

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow \varepsilon | + E \\
T & \rightarrow P T' \\
T' & \rightarrow \varepsilon | * T \\
P & \rightarrow n | (E)
\end{align*}
\]
- 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
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
- Parse tree
- AST

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
  - $S \rightarrow aA$
    - Node parse_S() {
      Node n1, n2;
      if (lookahead == "a") {
        n1 = match("a");
        n2 = parse_A();
        return new Node(n1, n2);
      }
    }
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)