CMSC 330: Organization of Programming Languages

Parsing

Recall: Steps of Compilation

Implementing Parsers

- 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
    - Other algorithms are bottom-up

Top-Down Parsing

E → id = n | { L }
L → E ; L | ε

(Assume: id is variable name, n is integer)

Show parse tree for
{ x = 3 ; { y = 4 ; } ; }
Bottom-up Parsing

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

Show parse tree for \( \{ x = 3 ; \{ y = 4 ; \} ; \} \)

Note that final trees constructed are same as for top-down; only order in which nodes are added to tree is different

Example: Shift-Reduce Parsing

- 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
- Complicated to use; requires tool support
  - Bison, yacc produce shift-reduce parsers from CFGs

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

Tradeoffs

- Recursive descent parsers
  - 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
  - Fast
    - Can be implemented with a simple table
- Shift-reduce parsers handle more grammars
  - Error messages may be confusing
- Most languages use hacked parsers (!)
  - Strange combination of the two
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

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

- 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

Parsing Example (cont.)

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

\[ \{ x = 3 ; \{ y = 4 ; \} ; \} \]

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 first in any sentential form derived from a nonterminal / production

First Sets

Definition
- $\text{First}(\gamma)$, 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 $S \rightarrow xyz \mid abc$
  - $\text{First}(xyz) = \{ x \}$, $\text{First}(abc) = \{ a \}$
  - $\text{First}(S) = \text{First}(xyz) \cup \text{First}(abc) = \{ x, a \}$
- 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 $\text{First}(\gamma)$

For a terminal $a$
- $\text{First}(a) = \{ a \}$

For a nonterminal $N$
- If $N \rightarrow \varepsilon$, then add $\varepsilon$ to $\text{First}(N)$
- If $N \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$ then (note the $\alpha_i$ are all the symbols on the right side of one single production):
  - Add $\text{First}(\alpha_1 \alpha_2 \ldots \alpha_n)$ to $\text{First}(N)$, where $\text{First}(\alpha_1 \alpha_2 \ldots \alpha_n)$ is defined as
    - $\text{First}(\alpha)_{i}$ if $\varepsilon \not\in \text{First}(\alpha_{i})$
    - Otherwise $(\text{First}(\alpha_{i}) - \varepsilon) \cup \text{First}(\alpha_{2} \ldots \alpha_{n})$
  - If $\varepsilon \in \text{First}(\alpha)$ for all $1 \leq i \leq k$, then add $\varepsilon$ to $\text{First}(N)$

First( ) Examples

$$E \rightarrow id = n \mid \{ L \} \mid \varepsilon$$
$$L \rightarrow E ; L \mid \varepsilon$$

$\text{First(id)} = \{ id \}$
$\text{First(\"\")} = \{ \"\" \}$
$\text{First(n)} = \{ n \}$
$\text{First(\"\")} = \{ \"\" \}$
$\text{First(\"\")} = \{ \"\" \}$
$\text{First(\";\")} = \{ \";\" \}$
$\text{First(E)} = \{ id, \"\" \}$
$\text{First(L)} = \{ id, \"\", \varepsilon \}$
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 → β_1 | ... | β_k` be the productions of `N`
  - Here `β_i` is the entire right side of a production- a sequence of terminals and nonterminals
  - Pick the production `N → β_i` such that the lookahead is in `First(β_i)`
    - It must be that `First(β_i) ∩ First(β_j) = ∅` for `i ≠ j`
    - If there is no such production, but `N → ε` then return
    - Otherwise fail with a parse error
- Suppose `β_i = α_1 α_2 ... α_n`. Then call `parse_α_1(); ... ; parse_α_n();` to match the expected right-hand side, and return

Parser Implementation (cont.)

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

Recursive Descent Parser

Given grammar `S → xyz | abc`
- `First(xyz) = { x }, First(abc) = { a }

Parser
```cpp
parse_S() { 
  if (lookahead == "x") {
    match("x"); match("y"); match("z"); // S → xyz
  }
  else if (lookahead == "a") {
    match("a"); match("b"); match("c"); // S → abc
  }
  else error();
}
```
Recursive Descent Parser

Given grammar $S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z$

- $\text{First}(A) = \{ x, y \}$, $\text{First}(B) = \{ z \}$

Parser

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

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

```c
void parse_B() {
    if (lookahead == "z") {
        match("z"); // B \rightarrow z
    } else {
        error();
    }
}
```

Example

$$E \rightarrow \text{id} = n \mid \{ L \}$$

- $\text{First}(E) = \{ \text{id}, \{ \} \}$

```c
void parse_E() {
    if (lookahead == "id") {
        match("id");
        match("=");
        parse_A(); // E \rightarrow \text{id} = A
        match("n");
    } else if (lookahead == ")") {
        match(";");
        parse_L(); // E \rightarrow \text{id} = B
        match(";");
    } else {
        error();
    }
}
```

```c
void parse_L() {
    if (lookahead == "id" || lookahead == ")") {
        parse_A(); // L \rightarrow A
        match(";");
        parse_L();
    } else {
        // L \rightarrow ε
    }
}
```

Things to Notice

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

Examples

- Grammar
  - $S \rightarrow xyz$
  - $A \rightarrow x \mid y$
  - $B \rightarrow z$
- String "xyz"
  ```
  parse_S()
  match("x")
  match("y")
  match("z")
  ```

- Grammar
  - $S \rightarrow A \mid B$
  - $A \rightarrow x \mid y$
  - $B \rightarrow z$
- String "x"
  ```
  parse_S()
  match("x")
  ```

Things to Notice (cont.)

- This is a **predictive parser**
  - Because the lookahead determines exactly which production to use

- This parsing strategy may fail on some grammars
  - Production First sets overlap
  - Production First sets contain $\epsilon$
  - Possible infinite recursion

- Does not mean grammar is not usable
  - Just means this parsing method not powerful enough
  - May be able to change grammar
Conflicting FIRST Sets

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

Left Factoring Algorithm

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

Alternative Approach

- Change structure of parser
  - First match common prefix of productions
  - Then use lookahead to chose between productions
- Example
  - Consider parsing the grammar \( E \rightarrow a+b \mid a\ast b \mid a \)
    ```
    parse_E() {
    match("a"); // common prefix
    if (lookahead == "+") {
        // E → a+b
        match("+"); match("b");}
    else if (lookahead == "\ast") { // E → a\ast b
        match("\ast"); match("b");
        else { // E → a
    }
    }
    ```

Left Recursion

- Consider grammar \( S \rightarrow Sa \mid \varepsilon \)
  - Try writing parser
    ```
    parse_S() {
    if (lookahead == "a") {
        parse_S();
        match("a"); // S → 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 $S \to aS \mid \epsilon$
  - Again, $\text{First}(aS) = a$
  - Try writing parser

```
parse_S() {
    if (lookahead == "a") {
        match("a");
        parse_S(); // $S \to aS$
    }
    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
  - $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
    - $\beta$ must exist or derivation will not yield string
- Rewrite grammar as (repeat as needed)
  - $A \to \beta L$
  - $L \to \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \epsilon$
- Replaces left recursion with right recursion

Examples
- $S \to Sa \mid \epsilon$  ⇐ $S \to aL \mid \epsilon$
- $S \to Sa \mid Sb \mid c$  ⇐ $S \to cL \mid S \to bL \mid \epsilon$

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
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)
  - $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$

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

- $S \rightarrow aA$

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

The Compilation Process

- source program → Compiler → target program
- Lexing → Parsing → Intermediate Code Generation → Optimization
- regexps DFAs → CFGs PDAs
- (may not actually be constructed)