CMSC 330: Organization of Programming Languages

Parsing
Recall: Steps of Compilation

- Lexing
- Parsing
- Intermediate Code Generation
- Optimization

Source program → Compiler → Target program
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

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

(Assume: \text{id} is variable name, \text{n} is integer)

Show parse tree for
\[
\{ \text{x} = 3 ; \{ \text{y} = 4 ; \} ; \} \]

{ x = 3 ; { y = 4 ; } ; }
Bottom-up Parsing

E → id = n | { L }
L → E ; L | ε

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
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

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

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

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

One input might be

- \( \{ \; \text{x} = 3 \; \}; \; \{ \; \text{y} = 4 \; \}; \; \} \)

- This would get turned into a list of tokens
  \[ \{ \; \text{x} = 3 \; \}; \; \{ \; \text{y} = 4 \; \}; \; \} \]

- And we want to turn it into a parse tree
Parsing Example (cont.)

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

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

Lookahead
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 \textit{x}
• Given grammar \( S \rightarrow xyz \mid abc \)
  \( \rightarrow \) Select \( S \rightarrow xyz \) since 1st terminal in RHS matches \( x \)
• Given grammar \( S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z \)
  \( \rightarrow \) 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 \textit{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 $a$
  - $\text{First}(a) = \{ a \}$

- 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
      - $\text{First}(\alpha_1)$ if $\varepsilon \notin \text{First}(\alpha_1)$
      - Otherwise $(\text{First}(\alpha_1) – \varepsilon) \cup \text{First}(\alpha_2 \ldots \alpha_n)$
    - If $\varepsilon \in \text{First}(\alpha_i)$ for all $i$, $1 \leq i \leq k$, then add $\varepsilon$ to $\text{First}(N)$
First( ) Examples

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

First(id) = \{ id \}
First(\"=\") = \{ \"=\" \}
First(n) = \{ n \}
First(\"\}\) = \{ \"\}\}\
First(\";\") = \{ \";\" \}
First(E) = \{ id, \"\}\}\
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 \( N \rightarrow \beta_1 \mid \ldots \mid \beta_k \) 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 \( \text{First}(\beta_i) \)
    - It must be that \( \text{First}(\beta_i) \cap \text{First}(\beta_j) = \emptyset \) for \( i \neq j \)
    - If there is no such production, but \( N \rightarrow \varepsilon \) then return
    - Otherwise fail with a parse error
  - Suppose \( \beta_i = \alpha_1 \alpha_2 \ldots \alpha_n \). Then call `parse_\alpha_1() ; \ldots ; parse_\alpha_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 $S \rightarrow xyz \mid abc$
  - First($xyz$) = { x }, First($abc$) = { a }
- Parser

```cpp
void 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
  ```java
  void parse_S() {
    if ((lookahead == "x") || (lookahead == "y")) {
      parse_A(); // S → A
    } else if (lookahead == "z") {
      parse_B(); // S → B
    } else error();
  }

  void parse_A() {
    if (lookahead == "x") {
      match("x"); // A → x
    } else if (lookahead == "y") {
      match("y"); // A → y
    } else error();
  }

  void parse_B() {
    if (lookahead == "z") {
      match("z"); // B → z
    } else error();
  }
  ```
Example

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

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

$L \rightarrow E ; L \mid \varepsilon$

\begin{align*}
\text{parse}_E( ) & \{ \\
& \text{if (lookahead == \"id\") } \{ \\
& \text{match("id");} \\
& \text{match("="); } \text{ // E \rightarrow id = n} \\
& \text{match("n");} \\
& \} \\
& \text{else if (lookahead == \"\{"\}) } \{ \\
& \text{match("\{"\});} \\
& \text{parse}_L( ); \text{ // E \rightarrow \{ L \}} \\
& \text{match("\}"}); \\
& \} \\
& \} \\
& \text{else error() ;} \\
\}
\end{align*}

\begin{align*}
\text{parse}_L( ) & \{ \\
& \text{if ((lookahead == \"id\") ||} \\
& \text{(lookahead == \"\{"\})) } \{ \\
& \text{parse}_E( );} \\
& \text{match(";")); \text{ // L \rightarrow E ; L} \\
& \text{parse}_L( );} \\
& \} \\
& \text{else ; } \text{ // L \rightarrow \varepsilon} \\
\}
\end{align*}
Things to Notice

If you draw the execution trace of the parser

• You get the parse tree

Examples

• Grammar
  \[ S \rightarrow xyz \]
  \[ S \rightarrow abc \]

• String “xyz”
  \[
  \text{parse}_S() \\
  \text{match(“x”) / \text{\textbackslash} \text{\textbackslash}} \\
  \text{match(“y”) x y z} \\
  \text{match(“z”)}
  \]

• Grammar
  \[ S \rightarrow A | B \]
  \[ A \rightarrow x | y \]
  \[ B \rightarrow z \]

• String “x”
  \[
  \text{parse}_S() \\
  \text{\textbackslash} \\
  \text{parse}_A() \\
  \text{match(“x”)}
  \]
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 $E \rightarrow ab \mid ac$

- $\text{First}(ab) = a$  Parse cannot choose between
- $\text{First}(ac) = a$  RHS based on lookahead!

Parser fails whenever $A \rightarrow \alpha_1 \mid \alpha_2$ and

- $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \neq \varepsilon \text{ or } \emptyset$

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
- \( S \rightarrow ab | ac \quad \Rightarrow S \rightarrow aL \quad L \rightarrow b | c \)
- \( S \rightarrow abcA | abB | a \quad \Rightarrow S \rightarrow aL \quad L \rightarrow bcA | bB | \epsilon \)
- \( L \rightarrow bcA | bB | \epsilon \quad \Rightarrow L \rightarrow bL' | \epsilon \quad L' \rightarrow cA | B \)
Alternative Approach

- **Change structure of parser**
  - First match common prefix of productions
  - Then use lookahead to choose between productions

- **Example**
  - Consider parsing the grammar $E \rightarrow a+b | a*b | a$

```c
parse_E( ) {
    match("a"); // common prefix
    if (lookahead == "+") { // E → a+b
        match("+"); match("b");
    } else {
    // E → a
    }
    if (lookahead == "+") { // E → a+b
        match("+"); match("b");
    } else {
    // E → a
    }
    if (lookahead == "*") { // E → a*b
        match("*"); 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 `parse_S( )` has an infinite loop
  - if (lookahead = "a") then parse_S( )
- Infinite loop occurs in grammar with left recursion
Right Recursion

Consider grammar $S \rightarrow aS \mid \varepsilon$

- Again, $\text{First}(aS) = a$
- Try writing parser

```cpp
parse_S() {
    if (lookahead == "a") {
        match("a");
        parse_S();  // $S \rightarrow aS$
    }
    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

• \(A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta\)
  - \(\beta\) must exist or derivation will not yield string

Rewrite grammar as (repeat as needed)

• \(A \rightarrow \beta L\)
• \(L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \epsilon\)

Replaces left recursion with right recursion

Examples

• \(S \rightarrow Sa \mid \epsilon\)  \(\Rightarrow S \rightarrow L\)  \(L \rightarrow aL \mid \epsilon\)
• \(S \rightarrow Sa \mid Sb \mid c\)  \(\Rightarrow S \rightarrow cL\)  \(L \rightarrow aL \mid bL \mid \epsilon\)
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)
  - $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$

```
    *  
   / 
  +  
 / 
 b   d
```
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

```java
Node parse_S() {
    Node n1, n2;
    if (lookahead == "a") {
        n1 = match("a");
        n2 = parse_A();
        return new Node(n1, n2);
    }
    return null;
}
```
The Compilation Process

source program → Compiler → target program

Lexing → Parsing → AST → Intermediate Code Generation → Optimization

regexps DFAs → CFGs PDAs

(may not actually be constructed)