Parsing

CSE 413, Autumn 2004
Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/

Common Orderings

- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)

- Bottom-up
  - Start at leaves and build up to the root
  - Effectively a rightmost derivation in reverse

OUTLINE

- Top-Down Parsers
  - Table-driven parsers
  - Recursive-descent parsers
  - Problems with recursive-descent parsers

- Bottom-up Parsers

- See Sections 4.3-4.5 of the textbook

Parse Tree Example

G

W → a = 1 ; if ( a + 1 ) b = 2 ;

Basic Parsing Strategies

- Top-Down
  - Begin at root with start symbol of grammar
  - Repeatedly pick a non-terminal and expand
  - Success when expanded tree matches input
  - LL(k)

LL(k) Parsers

- An LL(k) parser
  - Scans the input Left to right
  - Constructs a Leftmost derivation
  - Looking ahead at most k symbols

- 1-symbol look ahead is enough for many practical programming language grammars
Top-Down Parsing

- Situation: have completed part of a derivation
  \[ S \Rightarrow^* \alpha w \Rightarrow^* wxy \]
- Basic Step: Pick some production
  \[ A \rightarrow \beta_1 \beta_2 \ldots \beta_n \]
  that will properly expand \( A \) to match the input
  » Want this to be deterministic

Predictive Parsing

- If we are located at some non-terminal \( A \), and there are two or more possible productions
  \[ A \rightarrow \alpha \]
  \[ A \rightarrow \beta \]
  we want to make the correct choice by looking at just the next input symbol
- If we can do this, we can build a predictive parser that can perform a top-down parse without backtracking

Example

- Programming language grammars are often suitable for predictive parsing
- Common situation
  \[ stmt \rightarrow id = expr ; | return expr ; | if ( expr ) stmt | while ( expr ) stmt \]

Can we predict what production to use?

LL(1) Property

- FIRST(\( \alpha \))
  » the set of tokens that appear as the first symbols of one or more strings generated from \( \alpha \)
  » for example, from preceding slide:
  \[ stmt \rightarrow id = expr ; | return expr ; | if ( expr ) stmt | while ( expr ) stmt \]
  \[ \text{FIRST}(stmt) = \]

- A grammar has the LL(1) property if,
  » for all non-terminals \( A \), if productions \( A := \alpha \) and \( A := \beta \) both appear in the grammar, then \( \text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset \)
- If a grammar has the LL(1) property, we can build a predictive parser for it

Table-Driven LL(k) Parsers

- A table-driven parser can be constructed from the grammar (also true for LR(k))
- Example
  1. \( S \rightarrow ( S ) S \)
  2. \( S \rightarrow [ S ] S \)
  3. \( S \rightarrow \epsilon \)
- Table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>W: ( ( [ ] ) ) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursive-Descent Parsers

- An advantage of top-down parsing is that it is easy to implement by hand
- Key idea: write a function (procedure, method) corresponding to each non-terminal in the grammar
  » Each of these functions is responsible for matching its non-terminal with the next part of the input
Example: Statements

```c
void parseStmt() {
    switch(nextToken.getType()) {
    case Token.ID:
        parseAssignStmt(); break;
    case Token.KW_RETURN:
        parseReturnStmt(); break;
    case Token.KW_IF:
        parseIfStmt(); break;
    case Token.KW_WHILE:
        parseWhileStmt(); break;
    default:
        error(); break;
    }
}
```

// parse stmt → id=exp; | return expr; | if (expr) stmt

Example (cont)

```c
void parseWhileStmt() {
    matchToken(Token.KW_WHILE);
    matchToken(Token.LPAREN);
    parseExpr();
    matchToken(Token.RPAREN);
    parseStmt();
}
```

```c
void parseReturnStmt() {
    matchToken(Token.KW_RETURN);
    parseExpr();
    matchToken(Token.SEMICOLON);
}
```

Note: your code needs to handle the case when matchToken fails.

Invariant for Functions

- The parser functions need to agree on where they are in the input
- Useful invariant: When a parser function is called, the current token (next unprocessed piece of the input) is the token that begins the expanded non-terminal
  » Corollary: when a parser function is done, it must have completely consumed input correspond to that non-terminal

Possible Problems

- Two common problems for recursive-descent (and LL(1)) parsers
  1. Left recursion (e.g., \(E \rightarrow E + T \mid \cdots\))
  2. Common prefixes on the right hand side of productions

Problem #1: Left Recursion

- Grammar rule
  \(expr \rightarrow expr + term\)  
  \(term\)
- Code
  ```c
  void parseExpr() {
      if (current token is ADD) {
          matchToken(ADD);
          parseTerm();
      }
  }
  ```
- And the bug is?????

Left Recursion Problem

- Non-solution: replace with a right-recursive rule
  \(expr \rightarrow term + expr \mid term\)
  » Why isn’t this the right thing to do?
Left Recursion Solution

- Rewrite using right recursion and a new non-terminal
- Original:
  \[ expr \rightarrow expr + term \mid term \]
- New
  \[ expr \rightarrow term exprTail \]
  \[ exprTail \rightarrow + term exprTail \mid \epsilon \]
- Properties

Another Way to Look at This

- Observe that
  \[ expr \rightarrow expr + term \mid term \]
  generates the sequence
  \[ term + term + term + \ldots + term \]
- We can sugar the original rule to show this
  
  » \[ expr \rightarrow term ( + term )* \]
  
  » or \[ expr \rightarrow term \{ + term \} \]
- This can simplify the parser code

Code for Expressions

```c
// parse // expr ::= term { + term }
void parseExpr() {
    parseTerm();
    while (next symbol is ADD) {
        matchToken(ADD);
        parseTerm();
    }
}

// parse // term ::= factor { * factor }
void term() {
    parseFactor();
    while (next symbol is MUL) {
        matchToken(MUL);
        parseFactor();
    }
}
```

What About Indirect Left Recursion?

- A grammar might have a derivation that leads to a left recursion
  \[ A \Rightarrow \beta_1 \Rightarrow \beta_n \Rightarrow A \gamma \]
- There are systematic ways to factor such grammars
  » But we won’t need them in our grammar
  » refer to a compiler text for more info

Problem #2: Left Factoring

- If two rules for a non-terminal have right hand sides that begin with the same symbol, we can’t predict which one to use
  \[ ifStmt \rightarrow if ( expr ) stmt \]
  \[ ifStmt \rightarrow if ( expr ) stmt \ else stmt \]
- Solution: Factor the common prefix
  \[ ifStmt \rightarrow if ( expr ) stmt ifTail \]
  \[ ifTail \rightarrow else stmt \mid \epsilon \]

Parsing if Statements

- But it may be easiest to just code up the "else matches closest if" rule directly:

```c
// parse // if ( expr ) stmt [ else stmt ]
void parseIfStmt() {
    matchToken(IF);
    matchToken(LPAREN);
    parseExpr();
    matchToken(RPAREN);
    parseStmt();
    if (next symbol is ELSE) {
        matchToken(ELSE);
        parseStmt();
    }
}
```
Another Lookahead Problem

- In languages like FORTRAN, parentheses are used for array subscripts
- A FORTRAN grammar includes something like
  \[ \text{factor} \rightarrow \text{id} \left( \text{subscripts} \right) \mid \text{id} \left( \text{arguments} \right) \mid \ldots \]
- When the parser sees “id (”, how can it decide between an array element reference and a function call?

Handling \( \text{id} \ (\ ?) \)

- Use the type of \( \text{id} \) to decide
- Use a covering grammar
  \[ \text{factor} \rightarrow \text{id} \left( \text{commaSeparatedList} \right) \mid \ldots \]
  and fix later when more information is available
- Semantic analysis after parsing can resolve details that are difficult to express directly in the grammar

Top-Down Parsing Concluded

- Works with a smaller set of grammars than bottom-up, but can be done for most sensible programming language constructs
- If you need to write a quick-n-dirty parser, recursive descent is often the method of choice

Basic Parsing Strategies

- Bottom-up
  - Build up tree from leaves
  - Shift next input or reduce using a production
  - Accept when all input read and reduced to start symbol of remaining input

Bottom-Up Parsing

- Idea: Read the input left to right
- Whenever we’ve matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- The upper edge of this partial parse tree is known as the \textit{frontier}

LR(1) Parsing

- Left to right scan
- Rightmost derivation
- 1 symbol lookahead
- Most practical programming languages have an LR(1) grammar
Details

• The bottom-up parser reconstructs a reverse rightmost derivation
• Given the rightmost derivation
  \[ S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w \]
  parser will discover \( \beta_{n-1} \Rightarrow \beta_n \), then \( \beta_{n-2} \Rightarrow \beta_{n-1} \), etc.
• Parsing terminates when
  » \( \beta_i \) reduced to \( S \) (success), or
  » No match can be found (syntax error)

How Do We Automate This?

• Key: given what we’ve already seen and the next input symbol, decide what to do.
• Choices:
  » Perform a reduction (ie, reduce)
  » Look ahead further (ie, shift)
• Can reduce \( A \Rightarrow \beta \) if both of these hold:
  » \( A \Rightarrow \beta \) is a valid production
  » \( A \Rightarrow \beta \) is a step in this rightmost derivation
• This is known as a shift-reduce parser

Shift-Reduce Parser Operations

• Shift – push the next input symbol onto the stack
• Reduce – if the top of the stack is the right side of a handle \( A \ ::= \beta \), pop the right side \( \beta \) and push the left side \( A \).
• Accept – announce success
• Error – syntax error discovered

How Do We Automate This?

• Definition
  » Viable prefix – a prefix of a form that can appear on the stack of the shift-reduce parser
• Construct a DFA to recognize viable prefixes given the stack and remaining input
  » Perform reductions when we recognize them
• Most compiler building tools are based on this design and implement LR parsing using a DFA constructed from a set of grammar productions

LL vs LR

• Table-driven parsers for both LL and LR can be automatically generated by tools
• LL(1) has to make a decision based on a single non-terminal and the next input symbol
• LR(1) can base the decision on the entire left context as well as the next input symbol
• \( \because \) LR(1) is more powerful than LL(1)
  » Includes a larger set of grammars
  » but LL(1) is sufficient for many languages