Parsing

CSE 413, Autumn 2002
Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

Common Orderings

- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  - LL(k)
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost derivation in reverse
  - LR(k) and subsets (LALR(k), SLR(k), etc.)

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 frontier
LR(1) Parsing

- Left to right scan
- Rightmost derivation
- 1 symbol lookahead
- Most practical programming languages have an LR(1) grammar
- LALR(1), SLR(1), etc. – subsets of LR(1)

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 the grammar
  » LR(k) and subsets (SLR(k), LALR(k), …)

Example

- Grammar
  \[ S ::= aABe \]
  \[ A ::= A bc | b \]
  \[ B ::= d \]

- Bottom-up Parse

  a b b c d e

Details

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation
  \[ S =\beta_1 =\beta_2 = \ldots =\beta_{n-2} =\beta_{n-1} = \beta_n = w \]
  parser will discover \( \beta_{n-1} = \beta_n \), then \( \beta_{n-2} = \beta_{n-1} \), etc.
- Parsing terminates when
  » \( \beta_1 \) 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

Implementing Shift-Reduce Parsers

- Key Data structures
  - A stack holding the frontier of the tree
  - A string with the remaining input

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

Shift-Reduce Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>abbcde$</td>
<td>shift</td>
</tr>
</tbody>
</table>
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

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
  \[ S =>* wA\alpha =>* wxy \]
- Basic Step: Pick some production
  \[ A ::= \beta_1 \beta_2 \ldots \beta_n \]
  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
  $A ::= \alpha$
  $A ::= \beta$
we want to make the correct choice by looking at just the next input symbol
- If we can do this, we can build a prediction parser that can perform a top-down parse without backtracking

Example

- Programming language grammars are often suitable for predictive parsing
- Common situation
  $stmt ::= id = expr \mid return expr \mid if ( expr ) stmt \mid while ( expr ) stmt$
If the first part of the unparsed input begins with the tokens
  IF LPAREN ID(x) …
we know we can expand $stmt$ to an if-statement

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: FIRST($stmt$) = \{Token.ID, Token.KW_RETURN, Token.KW_IF, Token.KW WHILE\}
- 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. $S ::= ( S ) S$
  2. $S ::= [ S ] S$
  3. $S ::= \varepsilon$
- Table

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
</table>

$w: ( [ ] )$
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
- \(\therefore\) LR(1) is more powerful than LL(1)
  - Includes a larger set of grammars
  - but LL(1) is sufficient for many languages

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

```
// parse stmt ::= id=exp; | ...

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;
    }
}
```

Example (cont)

```
// parse while (exp) stmt
void parseWhileStmt() {
    matchToken(Token.KW_WHILE);
    matchToken(Token.LPAREN);
    parseExpr();
    matchToken(Token.RPAREN);
    parseStmt();
}

// parse return exp ;
void parseReturnStmt() {
    matchToken(Token.KW_RETURN);
    matchToken(Token.LPAREN);
    parseExpr();
    matchToken(Token.RPAREN);
    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
  - Left recursion (e.g., \( E ::= E + T | \ldots \) )
  - Common prefixes on the right hand side of productions

Left Recursion Problem

- Grammar rule
  \[
  expr ::= expr + term \\
  | \quad \text{term}
  \]

- Code
  ```
  // parse expr ::= ...
  void parseExpr() {
    parseExpr();
    if (current token is ADD) {
      matchToken(ADD);
      parseTerm();
    }
  }
  ```

- And the bug is?????
Left Recursion Solution

- Rewrite using right recursion and a new non-terminal
- Original: \( expr ::= expr + term \mid term \)
- New
  \[
  expr ::= term exprTail \\
  exprTail ::= + term exprTail \mid \epsilon
  \]
- Properties
  » No infinite recursion if coded up directly
  » Maintains left associativity (required)

Another Way to Look at This

- Observe that
  \[
  expr ::= expr + term \mid term
  \]
  generates the sequence
  \[term + term + term + \ldots + term\]
- We can sugar the original rule to show this
  » \( expr ::= term ( + term )* \)
  » or \( expr ::= term \{ + term \} \)
- 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();
    }
}

// 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
  \[
  A \Rightarrow B_1 \Rightarrow^* B_n \Rightarrow A\gamma
  \]
- 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
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
- Solution: Factor the common prefix into a separate production

Left Factoring Example

- Original grammar
  
  \[
  \text{ifStmt} ::= \text{if} (\text{expr}) \text{stmt} \\
  \quad| \text{if} (\text{expr}) \text{stmt else stmt}
  \]

- Factored grammar
  
  \[
  \text{ifStmt} ::= \text{if} (\text{expr}) \text{stmt ifTail} \\
  \text{ifTail} ::= \text{else stmt} | \epsilon
  \]

Parsing if Statements

- But it’s easiest to just code up the “else matches closest if” rule directly

```
// 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
  
  \[
  \text{factor} ::= \text{id} (\text{subscripts}) | \text{id} (\text{arguments}) | \ldots
  \]
- When the parser sees “\text{id} (”) how can it decide between an array element reference and a function call?
Handling \textit{id} ( ? )

- Use the type of \textit{id} to decide
  - Requires declare-before-use restriction if we want to parse in 1 pass
- Use a covering grammar
  \[
  \text{factor} ::= \text{id} \left( \text{commaSeparatedList} \right) | \ldots
  \]
  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