CSE 401 – Compilers

LL and Recursive-Descent Parsing
Hal Perkins
Winter 2017
Agenda

• Top-Down Parsing
• Predictive Parsers
• LL(k) Grammars
• Recursive Descent
• Grammar Hacking
  – Left recursion removal
  – Left factoring
Basic Parsing Strategies (1)

• Bottom-up
  – Build up tree from leaves
    • Shift next input or reduce a handle
    • Accept when all input read and reduced to start symbol of the grammar
  – LR(k) and subsets (SLR(k), LALR(k), ...)

remaining input
Basic Parsing Strategies (2)

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

- Situation: have completed part of a left-most derivation
  \[ S \Rightarrow^* wA\alpha \Rightarrow^* wxy \]
- 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 **predictive parser** that can perform a top-down parse without backtracking
Example

• Programming language grammars are often suitable for predictive parsing
• Typical example

\[
stmt ::= \text{id} = \text{exp} \; | \; \text{return} \; \text{exp} \; | \; \text{if} \; ( \; \text{exp} \; ) \; stmt \; | \; \text{while} \; ( \; \text{exp} \; ) \; stmt
\]

If the next part of the input begins with the tokens

\[
\text{IF} \; \text{LPAREN} \; \text{ID(x)} \; ...
\]

we should expand \( stmt \) to an if-statement
LL(1) Property

• 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 it is true that
  \[ \text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset \]

• If a grammar has the LL(1) property, we can build a predictive parser for it that uses 1 symbol lookahead
**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 lookahead is enough for many practical programming language grammars
  - LL(k) for k>1 is rare in practice
    - and even if the grammar isn’t quite LL(1), it may be close enough that we can pretend it is LL(1) and cheat a little when it isn’t
Table-Driven LL(k) Parsers

• As with LR(k), a table-driven parser can be constructed from the grammar

• Example
  1. $S ::= ( S ) S$
  2. $S ::= [ S ] S$
  3. $S ::= \varepsilon$

• Table

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>[ ]</th>
<th>$ $</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
LL vs LR (1)

- 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 (i.e., contents of the stack) as well as the next input symbol
LL vs LR (2)

\[ \therefore \text{ LR}(1) \text{ is more powerful than LL}(1) \]

- Includes a larger set of languages

\[ \therefore \text{ (editorial opinion) If you’re going to use a tool-generated parser, might as well use LR} \]

- But there are some very good LL parser tools out there (ANTLR, JavaCC, ...) that might win for other reasons
Recursive-Descent Parsers

• One advantage of top-down parsing is that it is easy to implement by hand
  – And even if you use automatic tools, the code may be easier to follow and debug

• Key idea: write a function (method, procedure) 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

Grammar

\[ stmt ::= \text{id} = \text{exp} \; | \; \text{return} \; \text{exp} \; | \; \text{if} \; ( \; \text{exp} \; ) \; stmt \; | \; \text{while} \; ( \; \text{exp} \; ) \; stmt \]

Method for this grammar rule

// parse stmt ::= id=exp; | ...
void stmt( ) {
    switch(nextToken) {
        RETURN: returnStmt(); break;
        IF: ifStmt(); break;
        WHILE: whileStmt(); break;
        ID: assignStmt(); break;
    }
}

Example (more statements)

// parse while (exp) stmt
void whileStmt() {
  // skip "while" "("
  getNextToken();
  getNextToken();
  // parse condition
  exp();
  // skip ")"
  getNextToken();
  // parse stmt
  stmt();
}

// parse return exp ;
void returnStmt() {
  // skip "return"
  getNextToken();
  // parse expression
  exp();
  // skip ";"
  getNextToken();
}
Invariant for Parser 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 being parsed
  – 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., $E ::= E + T \mid \ldots$)
  – Common prefixes on the right side of productions
Left Recursion Problem

Grammar rule

\[ expr ::= expr + term \]
\[ \quad | \quad term \]

And the bug is?????

Code

// parse expr ::= ...
void expr() {
    expr();
    if (current token is PLUS) {
        getNextToken();
        term();
    }
}


UW CSE 401 Winter 2017
Left Recursion Problem

• If we code up a left-recursive rule as-is, we get an infinite recursion
• Non-solution: replace with a right-recursive rule

\[
expr ::= \text{term} + expr \mid \text{term}
\]

– Why isn’t this the right thing to do?
Formal Left Recursion Solution

• Rewrite using right recursion and a new non-terminal
• Original: $expr ::= expr + term \mid term$
• New:

  $expr ::= term \ extrptail$

  $\extrptail ::= + term \ extrptail \mid \varepsilon$

• Properties
  – No infinite recursion if coded up directly
  – Maintains required left associatively (if you handle things correctly in the semantic actions)
Another Way to Look at This

• Observe that

\[ expr ::= expr + term \mid term \]

generates the sequence

\[ \ldots (((term + term) + term) + \ldots) + term \]

• We can sugar the original rule to reflect this

\[ expr ::= term \{ + term \}^* \]

• This leads directly to recursive-descent parser code

  -- Just be sure to do the correct thing to handle associativity as the terms are parsed
Code for Expressions (1)

```c
// parse
// expr ::= term { + term }*
void expr() {
    term();
    while (next symbol is PLUS) {
        getNextToken();
        term();
    }
}

// parse
// term ::= factor { * factor }*
void term() {
    factor();
    while (next symbol is TIMES) {
        getNextToken();
        factor();
    }
}
```
Code for Expressions (2)

// parse
// factor ::= int | id | ( expr )
void factor() {

    switch(nextToken) {

        case INT:
            process int constant;
            getNextToken();
            break;

        case ID:
            process identifier;
            getNextToken();
            break;

        case LPAREN:
            getNextToken();
            expr();
            getNextToken();
            break;

        ...

    }
}
What About Indirect Left Recursion?

• A grammar might have a derivation that leads to a left recursion

\[ A \Rightarrow \beta_1 \Rightarrow^* \beta_n \Rightarrow A \gamma \]

• Solution: transform the grammar to one where all productions are either

\[ A ::= a\alpha \quad \text{– i.e., starts with a terminal symbol, or} \]
\[ A ::= A\alpha \quad \text{– i.e., direct left recursion} \]

then use formal left-recursion removal to eliminate all direct left recursions
Eliminating Indirect Left Recursion

• Basic idea: Rewrite all productions \( A ::= B \ldots \) where \( A \) and \( B \) are different non-terminals by using all \( B ::= \ldots \) productions to replace the initial rhs \( B \)

• Example: Suppose we have \( A ::= B\delta, B ::= \alpha, \) and \( B ::= \beta. \) Replace \( A ::= B\delta \) with \( A ::= \alpha\delta \) and \( A ::= \beta\delta. \)

• Need to pick an order to process the non-terminals to avoid re-introducing indirect left recursions. Not complicated, just be systematic.
  – Details in any compiler or formal-language textbook
Second Problem: 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

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

• Factored grammar

\[
\text{ifStmt ::= if ( expr ) stmt ifTail} \\
\text{ifTail ::= else stmt | } \varepsilon
\]
Parsing if Statements

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

• (If you squint properly this is really just left factoring with the two productions handled by a single routine)

```c
// parse
void ifStmt()
{
    getNextToken();
    getNextToken();
    expr();
    getNextToken();
    stmt();
    if (next symbol is ELSE) {
        getNextToken();
        stmt();
    }
}
```
Another Lookahead Problem

• In languages like FORTRAN, parentheses are used for array subscripts

• A FORTRAN grammar includes something like

  \[ \text{factor} ::= \text{id} ( \text{subscripts} ) \mid \text{id} ( \text{arguments} ) \mid \ldots \]

• When the parser sees “\text{id (}”, how can it decide whether this begins an array element reference or a function call?
Two Ways to Handle $id$ (?)

• Use the type of $id$ to decide
  — Requires declare-before-use restriction if we want to parse in 1 pass; also means parser needs semantic information, not just grammar

• Use a covering grammar
  
  $$\text{factor ::= id ( commaSeparatedList ) | ...}$$

  and fix/check later when more information is available (e.g., types)
Top-Down Parsing Concluded

• Works with a smaller set of grammars than bottom-up, but can be done for most sensible programming language constructs
  – Possibly with some grammar refactoring

• If you need to write a quick-n-dirty parser, recursive descent is often the method of choice
  – And some sophisticated hand-written parsers for real languages (e.g., C++) are “based on” LL parsing, but with lots of customizations
Parsing Concluded

• That’s it!
• On to the rest of the compiler
• Coming attractions
  – Intermediate representations (ASTs etc.)
  – Semantic analysis (including type checking)
  – Symbol tables
  – & more...