

CSE 582 – Compilers

LL and Recursive-Descent Parsing

Hal Perkins
Autumn 2002

10/16/2002 © 2002 Hal Perkins & UW CSE F-1

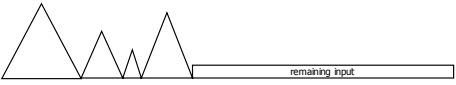
Agenda

- Top-Down Parsing
- Predictive Parsers
- LL(k) Grammars
- Recursive Descent
- Grammar Hacking
 - Left recursion removal
 - Factoring

10/16/2002 © 2002 Hal Perkins & UW CSE F-2

Basic Parsing Strategies (1)

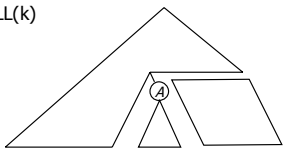
- Bottom-up
 - Build up tree from leaves
 - Shift next input or reduce a handle
 - Accept when all input read and reduced to start symbol of the grammar
 - LR(k) and subsets (SLR(k), LALR(k), ...)



10/16/2002 © 2002 Hal Perkins & UW CSE F-3

Basic Parsing Strategies (2)

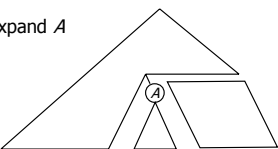
- Top-Down
 - Begin at root with start symbol of grammar
 - Repeatedly pick a non-terminal and expand
 - Success when expanded tree matches input
 - LL(k)



10/16/2002 © 2002 Hal Perkins & UW CSE F-4

Top-Down Parsing

- Situation: have completed part of a derivation
 $S \Rightarrow^* wA\alpha \Rightarrow^* wxy$
- Basic Step: Pick some production
 $A ::= \beta_1 \beta_2 \dots \beta_n$
that will properly expand A to match the input
 - Want this to be deterministic



10/16/2002 © 2002 Hal Perkins & UW CSE F-5

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

10/16/2002 © 2002 Hal Perkins & UW CSE F-6

Example

- Programming language grammars are often suitable for predictive parsing
- Common situation


```
stmt ::= id = exp ; | return exp ;
      | if ( exp ) stmt | while ( exp ) stmt
```

If the first part of the unparsed input begins with the tokens

```
IF LPAREN ID(x) ...
```

we can expand *stmt* to an if-statement

10/16/2002 © 2002 Hal Perkins & UW CSE F-7

LL(k) 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 the case that $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- If a grammar has the LL(1) property, we can build a predictive parser for it

10/16/2002 © 2002 Hal Perkins & UW CSE F-8

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

10/16/2002 © 2002 Hal Perkins & UW CSE F-9

Table-Driven LL(k) Parsers

- As with LR(k), a table-driven parser can be constructed from the grammar
- Example
 - $S ::= (S) S$
 - $S ::= [S] S$
 - $S ::= \epsilon$
- Table

	()	[]	\$
S	1	3	2	3	3

10/16/2002 © 2002 Hal Perkins & UW CSE F-10

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 as well as the next input symbol

10/16/2002 © 2002 Hal Perkins & UW CSE F-11

LL vs LR (2)

- \therefore LR(1) is more powerful than LL(1)
 - Includes a larger set of grammars
- \therefore (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, ...)

10/16/2002 © 2002 Hal Perkins & UW CSE F-12

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

10/16/2002

© 2002 Hal Perkins & UW CSE

F-13

Example: Statements

```
Grammar
stmt ::= id = exp ;
      | return exp ;
      | if ( exp ) stmt
      | while ( exp ) stmt

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

10/16/2002

© 2002 Hal Perkins & UW CSE

F-14

Example (cont)

```
// 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();
}
```

10/16/2002

© 2002 Hal Perkins & UW CSE

F-15

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

10/16/2002

© 2002 Hal Perkins & UW CSE

F-16

Possible Problems

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

10/16/2002

© 2002 Hal Perkins & UW CSE

F-17

Left Recursion Problem

```
Grammar rule
expr ::= expr + term
      | term

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

And the bug is????
```

10/16/2002

© 2002 Hal Perkins & UW CSE

F-18

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

$$expr ::= term + expr \mid term$$
 - Why isn't this the right thing to do?

10/16/2002 © 2002 Hal Perkins & UW CSE F-19

Left Recursion Solution

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

$$expr ::= term \textit{exptail}$$

$$\textit{exptail} ::= + term \textit{exptail} \mid \epsilon$$
- Properties
 - No infinite recursion if coded up directly
 - Maintains left associativity (required)

10/16/2002 © 2002 Hal Perkins & UW CSE F-20

Another Way to Look at This

- Observe that

$$expr ::= expr + term \mid term$$
 generates the sequence

$$term + term + term + \dots + term$$
- We can sugar the original rule to show this

$$expr ::= term \{ + term \}$$
- This leads directly to parser code

10/16/2002 © 2002 Hal Perkins & UW CSE F-21

Code for Expressions (1)

```

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

```

10/16/2002 © 2002 Hal Perkins & UW CSE F-22

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();
  }
}

```

10/16/2002 © 2002 Hal Perkins & UW CSE F-23

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
 - See the book

10/16/2002 © 2002 Hal Perkins & UW CSE F-24

Left Factoring

- n 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
- n Solution: Factor the common prefix into a separate production

10/16/2002 © 2002 Hal Perkins & UW CSE F-25

Left Factoring Example

- n Original grammar

$$ifStmt ::= if (expr) stmt \mid if (expr) stmt \text{ else } stmt$$
- n Factored grammar

$$ifStmt ::= if (expr) stmt \text{ ifTail}$$

$$ifTail ::= \text{ else } stmt \mid \epsilon$$

10/16/2002 © 2002 Hal Perkins & UW CSE F-26

Parsing if Statements

- n But it's easiest to just code up the "else matches closest if" rule directly

```
// parse
// if (expr) stmt [ else stmt ]
void ifStmt() {
    getNextToken();
    getNextToken();
    expr();
    getNextToken();
    stmt();
    if (next symbol is ELSE) {
        getNextToken();
        stmt();
    }
}
```

10/16/2002 © 2002 Hal Perkins & UW CSE F-27

Another Lookahead Problem

- n In languages like FORTRAN, parentheses are used for array subscripts
- n A FORTRAN grammar includes something like

$$factor ::= id (subscripts) \mid id (arguments) \mid \dots$$
- n When the parser sees "id(", how can it decide between an array element reference and a function call?

10/16/2002 © 2002 Hal Perkins & UW CSE F-28

Handling *id*(?)

- n Use the type of *id* to decide
 - n Requires declare-before-use restriction if we want to parse in 1 pass
- n Use a covering grammar

$$factor ::= id (commaSeparatedList) \mid \dots$$
 and fix later when more information is available

10/16/2002 © 2002 Hal Perkins & UW CSE F-29

Top-Down Parsing Concluded

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

10/16/2002 © 2002 Hal Perkins & UW CSE F-30



Parsing Concluded

- n That's it!
- n On to the rest of the compiler
- n Coming attractions
 - n Intermediate representations (ASTs &c)
 - n Semantic analysis (including type checking)
 - n Symbol tables
 - n & more...

10/16/2002

© 2002 Hal Perkins & UW CSE

F-31