CSE P 501 – Compilers

LL and Recursive-Descent Parsing

Hal Perkins

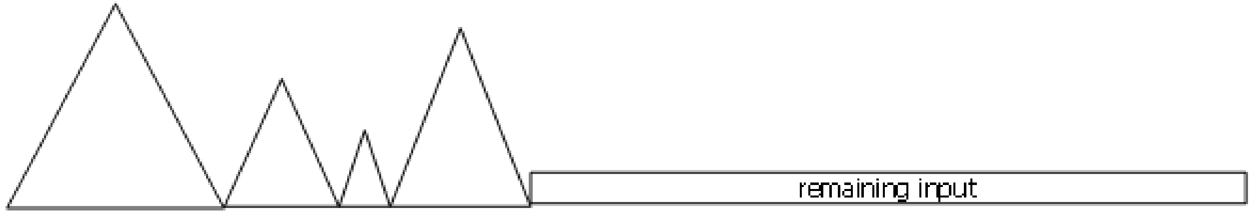
Winter 2016

Agenda

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

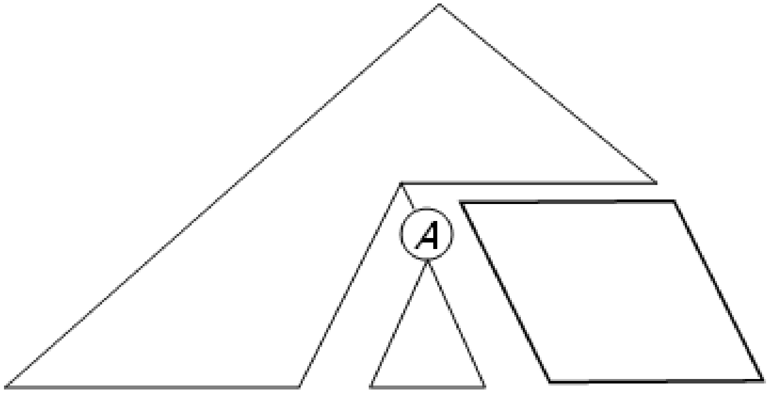
Basic Parsing Strategies (1)

- 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), …)



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

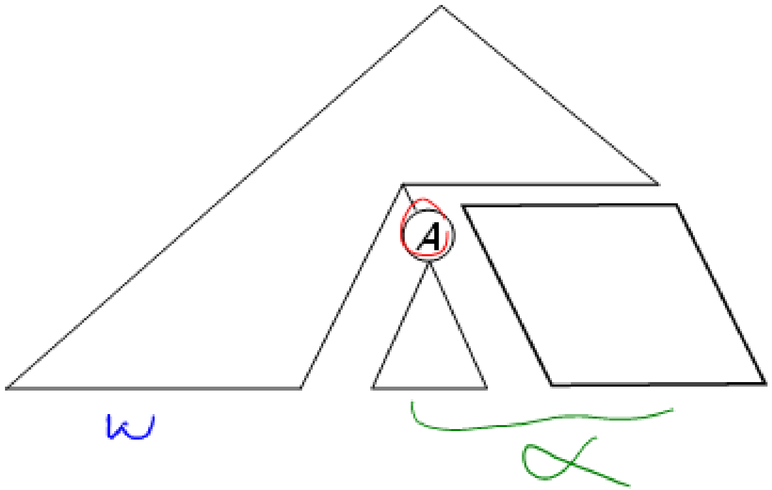
  $S \Rightarrow^* \underline{w}\underline{A}\alpha \Rightarrow^* wxy$

- 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 (i.e., no backtracking)



Predictive Parsing

- If we are located at some non-terminal *A*, and there are two or more possible productions

  $\underline{A} ::= \underline{\alpha}$

  $\underline{A} ::= \underline{\beta}$

  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

stmt

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

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

If the next part of the input begins with the tokens

IF LPAREN 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

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* ::= ε
- Table

term

nonterm row

	(	)	[	]	$
S	1	3	2	3	3

productions

LL vs LR (1)

- Tools can automatically generate parsers for both LL(1) and LR(1) grammars
- 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)

- ∴ LR(1) is more powerful than LL(1)
  - – Includes a larger set of languages
- ∴ (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 (documentation, IDE support, integrated AST generation, local culture/politics/economics etc.)

Recursive-Descent Parsers

- A main 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 (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

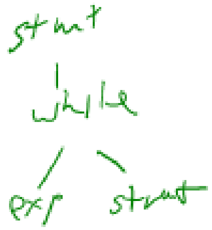
Grammar

stmt ::= *id* = *exp* ;
  | return *exp* ;
  | if (*exp*) *stmt*
  | while (*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" "("
    skipToken(WHILE);
    skipToken(LPAREN);

    // parse condition
    exp();

    // skip ")"
    skipToken(RPAREN);

    // parse stmt
    stmt();
}
```

```
// parse return exp ;
void returnStmt() {
    // skip "return"
    skipToken(RETURN);

    // parse expression
    exp();

    // skip ";"
    skipToken(SCOLON);
}
```

```
// aux method: advance past expected token
void skipToken(Token expected) {
    if (nextToken == expected)
        getNextToken();
    else error("token" + expected + "expected");
}
```

Recursive-Descent Recognizer

- Easy!
- Pattern of method calls traces leftmost derivation in parse tree
- Examples only handle valid programs and choke on errors. Real parsers need:
  - Better error recovery (don't get stuck on bad token)
  - Semantic checks (declarations, type checking, …)
  - Some sort of processing after recognizing (build AST, 1-pass code generation, …)

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

- Need to fix to avoid backtracking

Left Recursion Problem

Grammar rule

expr ::= *expr* + *term*
| *term*

And the bug is????

Code

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

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 | term

  – Why isn’t this the right thing to do?

One Left Recursion Solution

- Rewrite using right recursion and a new non-terminal
- Original: *expr* ::= *expr* + *term* | *term*
- New

  expr ::= *term exprtail*

  exprtail ::= + *term exprtail* | ε
- Properties
  - No infinite recursion if coded up directly
  - Maintains required left associatively (*if* you interpret the parse tree the right way in the semantic actions)

Another Way to Look at This

- Observe that

  $expr ::= expr + term \mid term$

  generates the sequence

  $(\ldots(\underline{\underline{(term + term)} + term}) + \ldots) + term$

- We can sugar the original rule to reflect this

  $expr ::= term\ \underline{\{ + term \}}^*$

- This leads directly to parser code
  - Just be sure to do the correct thing to handle associativity as the terms are parsed

Code for Expressions (1)

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

```
// parse
//     term ::= factor { * factor }*
void term() {
      factor();
      while (next symbol is  TIMES) {
       skipToken(TIMES);
       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:
       skipToken(LPAREN);
       expr();
       skipToken(RPAREN);
     }
}
```

What About Indirect Left Recursion?

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

  → $A \texttt{=>} \beta_1 \texttt{=>}^* \beta_n \texttt{=>} A\ \gamma$

- There are systematic ways to factor such grammars – see the book for details
  - Basic idea: if $\underline{A_i ::= A_j\gamma}$ and $A_j ::= \delta_1 \mid \ldots \mid \delta_k$, replace $A_i ::= A_j\gamma$ with $A_i ::= \delta_1\gamma \mid \ldots \mid \delta_k\gamma$. Repeat for all nonterminals. The result may have direct left recursions which can can be eliminated with the previous transformations.

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

  ifStmt ::= if (*expr*) *stmt*

  | if (*expr*) *stmt* else *stmt*

- Factored grammar

  ifStmt ::= if (*expr*) *stmt* *ifTail*

  ifTail ::= else *stmt* | ε

Parsing if Statements

- But it's easiest to just directly code up "else matches closest if" rule
- (If you squint properly this is really just left factoring with the two productions combined in a single routine)

```
// parse
//     if (expr) stmt [ else stmt ]
void ifStmt() {
 ✓skipToken(IF);
 ✓skipToken(LPAREN);
 →expr();
 ✓skipToken(RPAREN);
 →stmt();
  if (next symbol is ELSE) {
   ✓skipToken(ELSE);
   →stmt();
  }
}
```

Another Lookahead Problem

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

  factor ::= *id* (*subscripts*) | *id* (*arguments*) | …
- When the parser sees "ID LPAREN", how can it decide if this starts an array access or a function call?

a(1,3)

Two Ways to Handle *id*(x, x, x)

- Use the type of *id* to decide
  - Requires declare-before-use restriction if we want to parse in 1 pass
  - (Modularity issue: requires more semantic processing in the parser)
- Use a covering grammar

  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
  - With some possible grammar refactoring
    - And maybe a little cheating (occasional extra lookahead, …)
- 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…