

CSE P 501 – Compilers

LL and Recursive-Descent Parsing

Hal Perkins

Autumn 2009

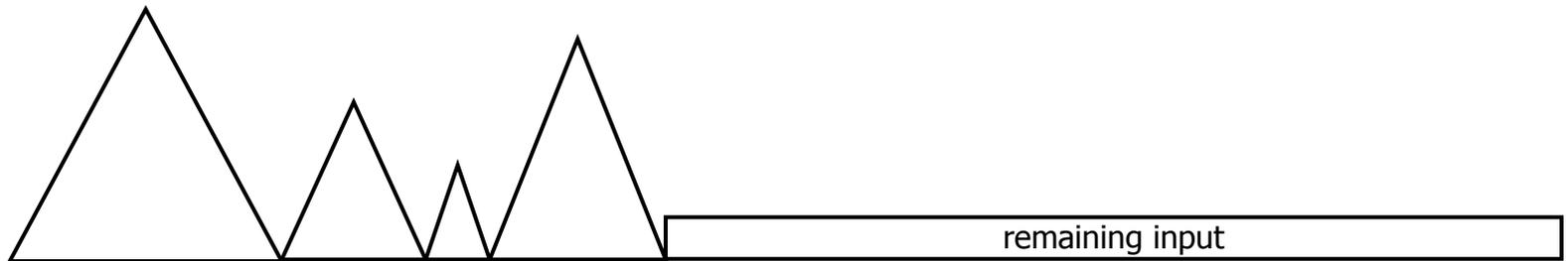


Agenda

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

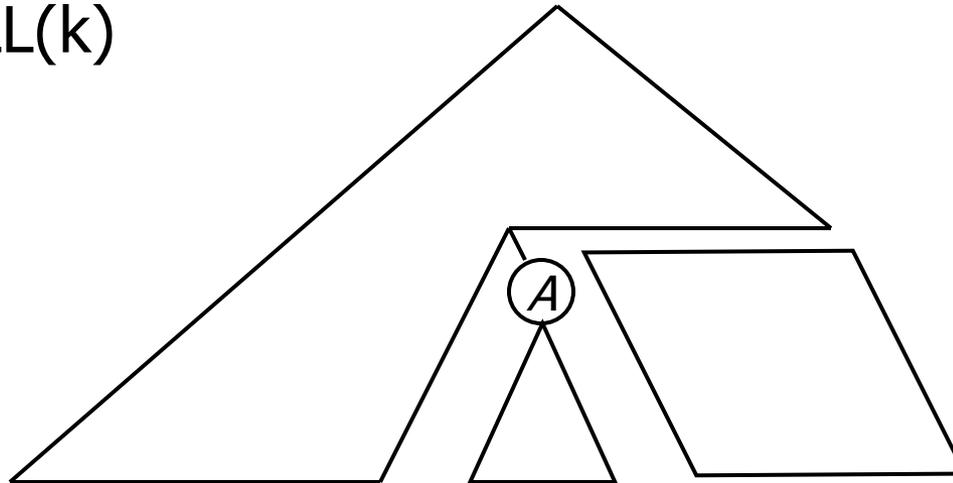
Basic Parsing Strategies (1)

- 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 derivation

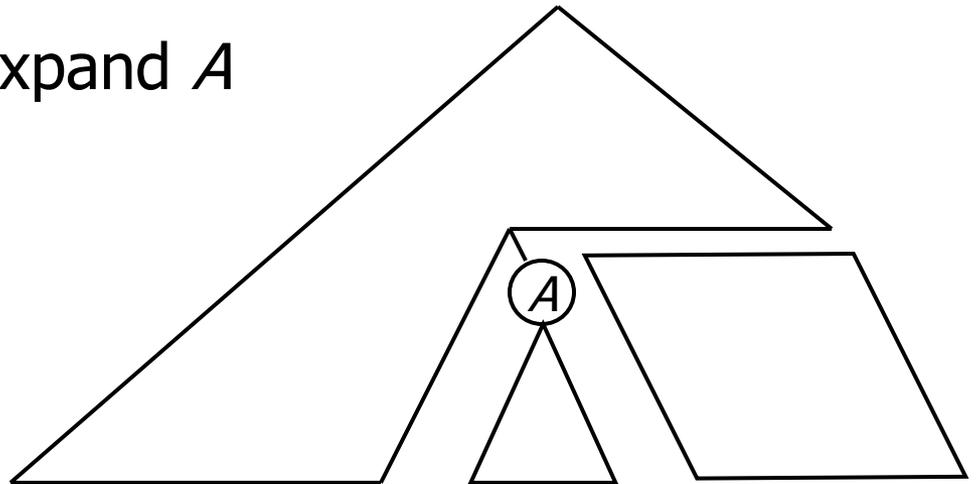
$$S \Rightarrow^* wA\alpha \Rightarrow^* wxy$$

- 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





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

$$\begin{aligned} stmt ::= & id = exp ; \mid \text{return } exp ; \\ & \mid \text{if } (exp) stmt \mid \text{while } (exp) stmt \end{aligned}$$

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

IF LPAREN ID(x) ...

we should expand *stmt* to an if-statement



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
$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$
- 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 **L**eft to right
 - Constructs a **L**eftmost derivation
 - Looking ahead at most **k** symbols
- 1-symbol lookahead is enough for many practical programming language grammars
 - LL(k) for $k > 1$ is very 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. $S ::= \epsilon$

- Table

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



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 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, ...) that might win for non-LLvsLR reasons



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

- 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 (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();
}
```



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 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., $E ::= E + T \mid \dots$)
 - Common prefixes on the right hand side of productions



Left Recursion Problem

- Grammar rule

$$\begin{aligned} \text{expr} ::= & \text{expr} + \text{term} \\ & | \text{term} \end{aligned}$$

- And the bug is????

- Code

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

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

- Why isn't this the right thing to do?



Left Recursion Solution

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



Another Way to Look at This

- Observe that

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

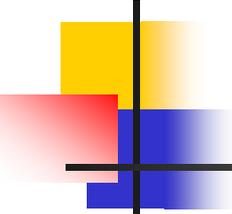
generates the sequence

$term + term + term + \dots + term$

- We can sugar the original rule to show this

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

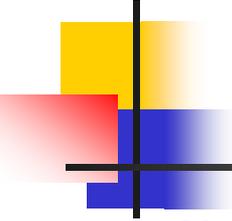
- This leads directly to parser code



Code for Expressions (1)

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

What About Indirect Left Recursion?

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

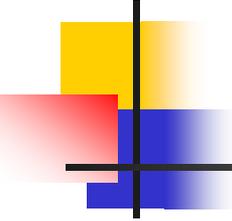
$$A \Rightarrow \beta_1 \Rightarrow^* \beta_n \Rightarrow A \gamma$$

- There are systematic ways to factor such grammars
 - See the book



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

$$\begin{aligned} \textit{ifStmt} ::= & \textit{if} (\textit{expr}) \textit{stmt} \\ & | \textit{if} (\textit{expr}) \textit{stmt} \textit{else} \textit{stmt} \end{aligned}$$

- Factored grammar

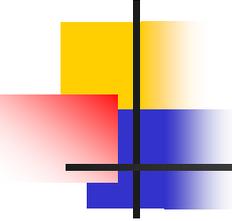
$$\begin{aligned} \textit{ifStmt} ::= & \textit{if} (\textit{expr}) \textit{stmt} \textit{ifTail} \\ \textit{ifTail} ::= & \textit{else} \textit{stmt} | \epsilon \end{aligned}$$



Parsing if Statements

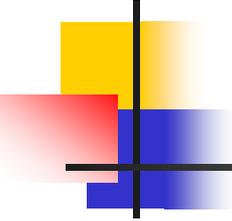
- But it's easiest to just code up the "else matches closest if" rule directly

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



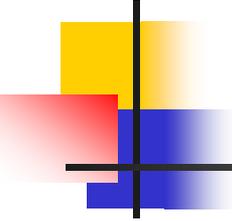
Another Lookahead Problem

- In languages like FORTRAN, parentheses are used for array subscripts
- A FORTRAN grammar includes something like
$$factor ::= id(subscripts) \mid id(arguments) \mid \dots$$
- When the parser sees “*id* (”, how can it decide whether this begins an array element reference or a function call?



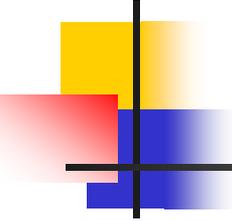
Two Ways to Handle *id* (?)

- Use the type of *id* to decide
 - Requires declare-before-use restriction if we want to parse in 1 pass
- Use a covering grammar
 - factor ::= id (commaSeparatedList) | ...*
 - and fix later when more information is available



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



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...