
CSE 413

Programming Languages &
Implementation

Hal Perkins

Winter 2019

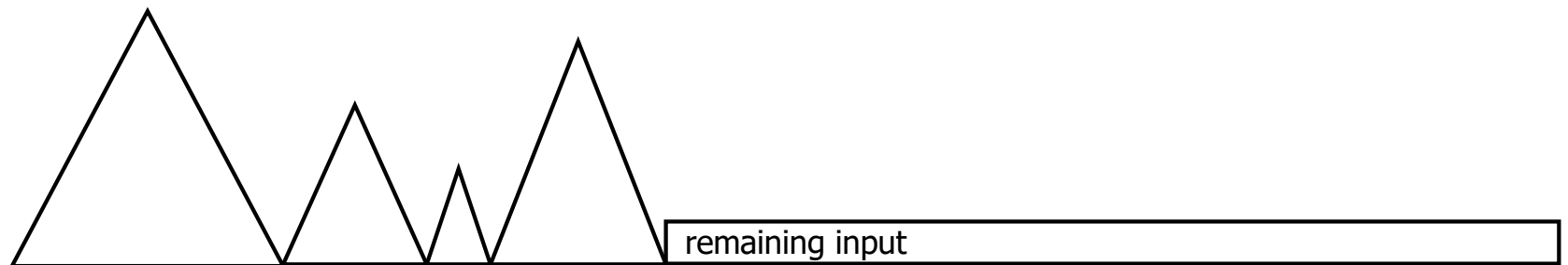
Top-Down and Recursive-Descent Parsing

Agenda

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

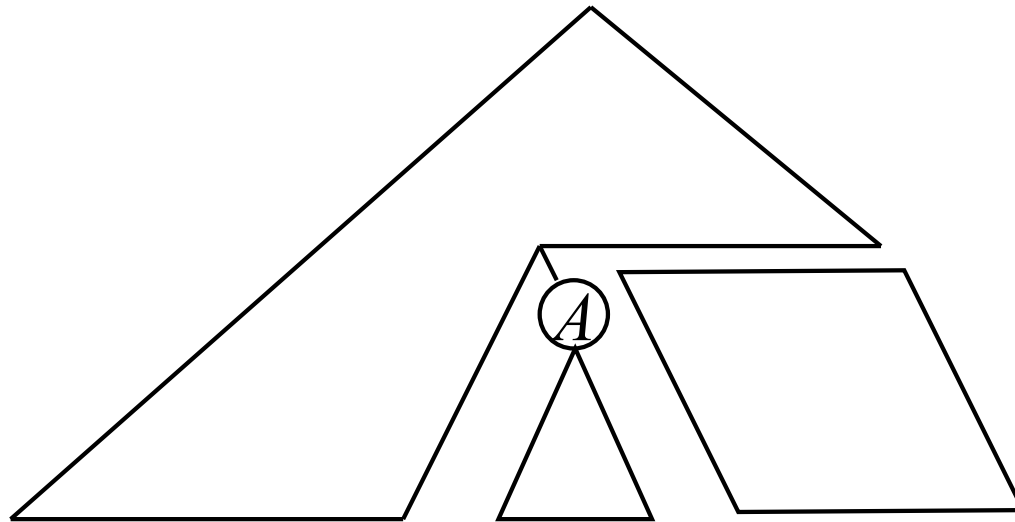
Basic Parsing Strategies (1)

- Bottom-up
 - Build up tree from leaves
 - Shift next input or reduce using a production
 - 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 leftmost derivation

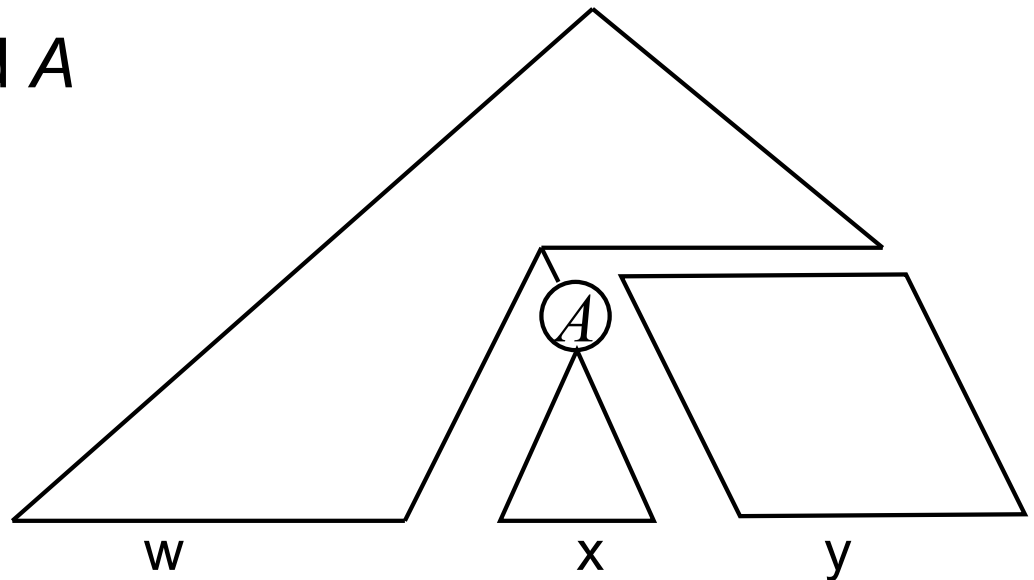
$$S \Rightarrow^* wA\alpha \Rightarrow^* wxy$$

- 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

Sounds hard, but ...

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

$stmt ::= id = exp ; \mid return\ exp ;$
 $\mid if (exp) stmt \mid while (exp) stmt$

If the remaining 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 , when there are two productions

$$A ::= \alpha$$

$$A ::= \beta$$

in the grammar, then:

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

($\text{FIRST}(\alpha)$ = set of terminals that begin any possible string derived from α)

- Assumption, neither α nor β can expand to ϵ . There are ways to handle this if it happens, but we will avoid the issue

- 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 realistic programming language grammars
 - LL(k) for $k > 1$ is very rare in practice

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

LL vs LR (2)

- ∴ LR(1) is more powerful than LL(1)
 - Includes a larger set of grammars

But

- It is easier to write a LL(1) parser by hand
- There are some very good LL parser tools out there (ANTLR, JavaCC, ...)

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 the next part of the input with the non-terminal it recognizes

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 Parser Functions

- The parser functions need to agree on where they are in the input
- Useful (and typical) 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 terminates, it must have completely consumed input corresponding 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

expr ::= expr + term
| term

- Code

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

- And the bug is????

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?

One 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
$$term + term + term + \dots + term$$
- We can sugar the original rule to match
$$expr ::= term \{ + term \}^*$$
- This leads directly to parser code
 - But need to fudge things to respect the original precedence/associativity

Code for Expressions (1)

```
// parse
//  expr ::= term { + term }*

void expr() {
    term();

    while (next symbol is PLUS) {
        // consume PLUS
        getNextToken();

        term();
    }
}
```

```
// parse
//  term ::= factor { * factor }*

void term() {
    factor();

    while (next symbol is TIMES) {
        // consume TIMES
        getNextToken();

        factor();
    }
}
```

Code for Expressions (2)

```
// parse
// factor ::= int | id | ( expr )

void factor() {
    switch(nextToken) {

        case INT:
            process int constant;
            // consume INT
            getNextToken();
            break;
        ...

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

        case LPAREN:
            // consume LPAREN
            getNextToken();
            expr();
            // consume RPAREN
            getNextToken();
    }
}
```

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
- “Official” 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:

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

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();
    }
}
```

Top-Down Parsing Concluded

- Works with a somewhat 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