UCF | College of Engineering and Computer Science
UNIVERSITY OF CENTRAL FLORIDA

Recursive Descent Parsing

COP-3402 Systems Software
Paul Gazzillo

Top-Down Parsing

- Goal: derive a given input from the grammar
- Two ways to go
  - Top-down: construct tree from root to leaves
  - Bottom-up: construct tree from leaves to root

Recursive Descent Parsing

- Each nonterminal is a function
- Each production defines the body of the function
- Terminals consume lexer tokens

```
program
  = statement*

statement
  = PRINT expression SEMI

factor
  = LPAREN expression RPAREN
  | NUMBER
```

```
program() {
  while (!done) statement();
}
statement() {
  // just like project 0
  // calls expression();
}
factor() {
  // how do we distinguish between
  // productions?
}
```

Predictive Parsing with a Lookahead

- How do we know which production to use?

```
program
  = statement*

statement
  = PRINT expression SEMI

factor
  = LPAREN expression RPAREN
  | NUMBER
```

```
program() {
  while (!done) statement();
}
statement() {
  // just like project 0
}
factor() {
  c = fgetc(...)
  if (c == '(') {
    lparen(); expression();
rparen();
  else if (c == '-' || isdigit(c)) {
    number();
  } else error();
}
```

Left Recursion

- What happens to our parser for the addition grammar?

```
expression
  = expression PLUS term
  | term
```

```
expression() {
  if (...) {
    expression();
    plus();
    term();
  } else …
}
```

Left Recursion Elimination

- Can we still parse the language recursively?
- How do we alter the grammar?
  - Is the resulting grammar equivalent?
- Intuition: convert left recursion to right recursion
  - Add new nonterminal for symbols after the recursion

Demo: Left Recursion Elimination

General Solution for Elimination

- In Dragon Book
  - (recall the notation differences in grammar)
- If the grammar has left recursion

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

- We can rewrite it as an equivalent grammar

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Elimination Enables Recursive Descent

- Intuition: look at first token for each alternative
  - If first symbol is a nonterminal, follow the grammar

```
expression
  = term expression_prime

expression_prime
  = PLUS term expression_prime
  | epsilon
```

```
expression() {
  term();
  expression_prime();
}

expression_prime() {
  // peek at the next token or char
  if (next is '+' or PLUS) {
    plusToken();
    term();
    expression_prime();
  } else // do nothing for epsilon
}
```

The FIRST and FOLLOW sets

- How do we know which production to use?
- Use the first token of the production
  - The FIRST set is the set of all first tokens for each nonterminal
    - Can include epsilon
  - The FOLLOW is the set of all tokens after a nonterminal
    - Needed to compute FIRST when a production starts with a nonterminal that can be epsilon
- Does not work for all grammars, but should for ours

Generating Code for Nested Expressions

- Postorder traversal of parse tree
  - Just like evaluating an expression tree
- Generate each node's LLVM IR
  - Store operation in new temporary variable
- Return temporary variable to parent
  - Parent uses child variables for its own operation

Conclusion

- Remove ambiguity by rewriting the grammar
- Remove left recursion to enable recursive descent
- Use a lookahead to determine which production to parse
- Generate code by postorder traversal
    - Each call returns a temporary variable storing the intermediate value