COP-3402 Systems Software
09.26 Thu

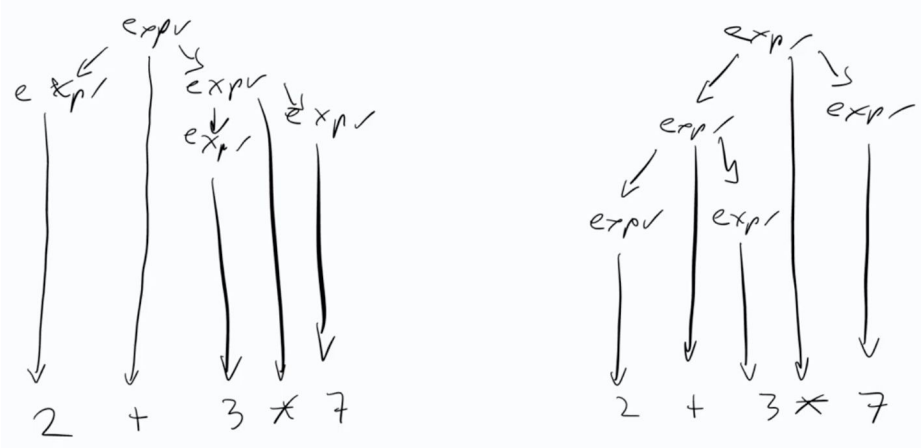
For project1's grammar, the operands of our expressions can take nested expression so there is a recursive structure of our language and context-free grammars are how we can express this type of language. Regular expressions cannot express these types of languages. These are not regular languages.

Demo: Arithmetic Expressions

```
expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | LPAREN expression RPAREN
  | NUMBER
```

Ex: 2 + 3 * 7
exp => exp + exp
=> num + exp
=> num + exp * exp
=> num + num * num

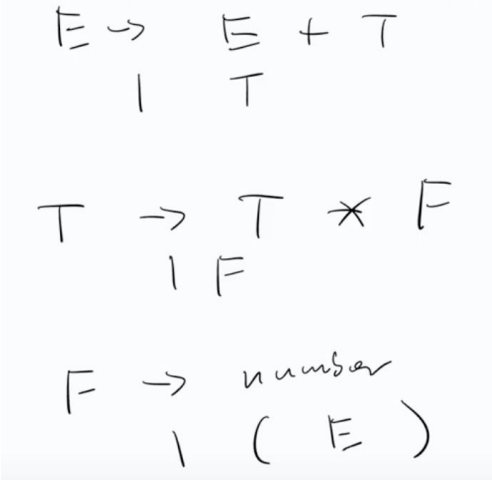
Ambiguity: (2 different solutions that result from the order of operations)



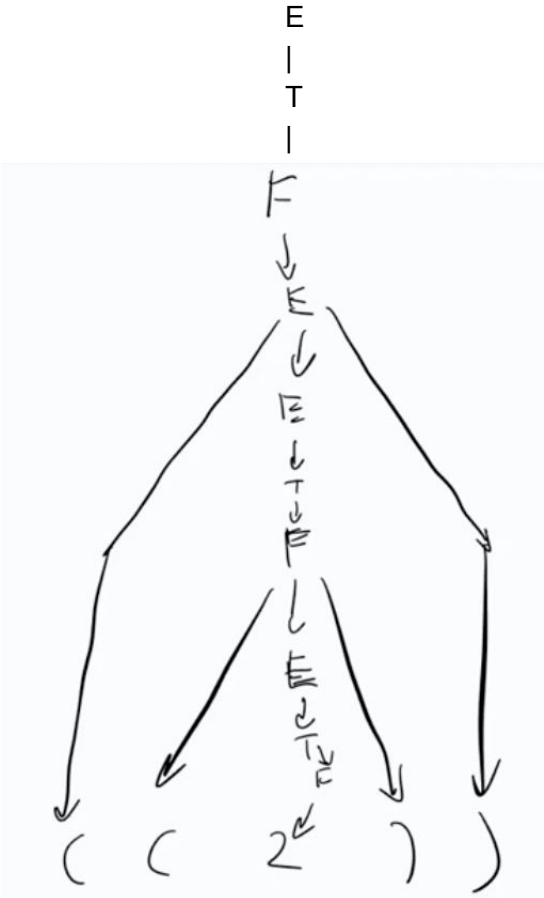
If you want to turn a postfix expression into a tree, you would do a post order traversal in order to compute the value of the expression. Multiplication is lower on the tree meaning it happens first (for the example on the left figure). The left one is 23, right one is 35. We prefer the tree on the left. For our language, we respect the PEMDAS order of operations. The way to resolve this

ambiguity, we modify the grammar. On the top level of tree, you should disallow multiplication/division and allow addition/subtraction:

Our new grammar:



Ex: ((2))



Recursive Descent Parsing:

Top-down parsing

Goal: drive a given input from grammar

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
}
factor() {
  c = fgetc(...)
  if (c == '(') {
    lparen(); expression(); rparen();
  else if (c == '-' || isdigit(c)) {
    number();
  } else error();
}
```

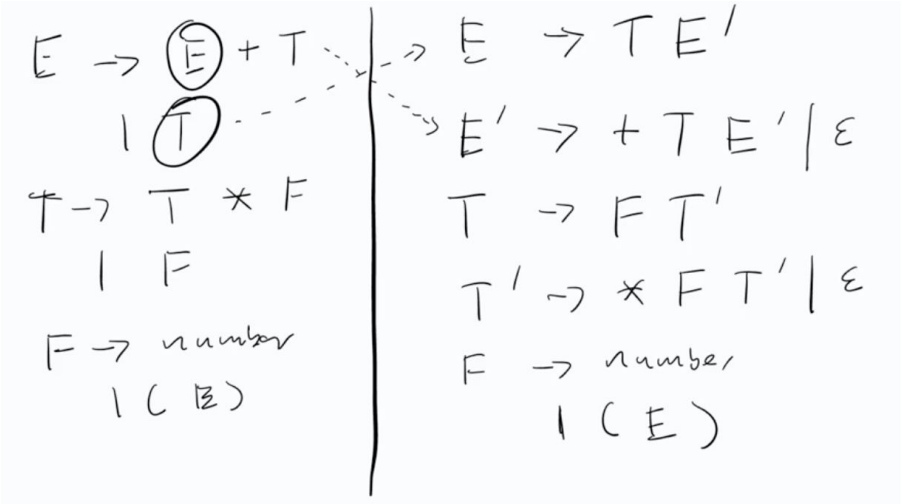
Left Recursion:

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

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

Left Recursion Elimination:

Convert left recursion to right recursion; add new nonterminal for symbols after the recursion:



If you have a production and it looks like a recursion with the base case, which we can have some left recursive call followed by some suffix, we take the base case put our new step after it, and it turns into a right recursive by taking the suffix from the left recursive part and adding the right recursive part after it, otherwise empty string.

Elimination Enables Recursive Descent:

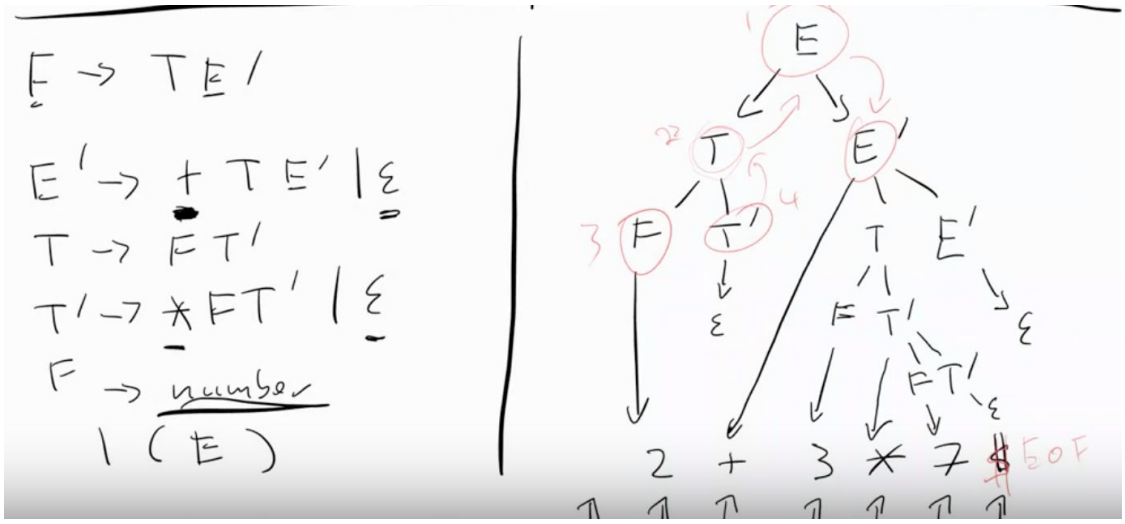
```
expression
  = term expression_prime

expression_prime
  = PLUS term expression_prime
  | epsilon
```

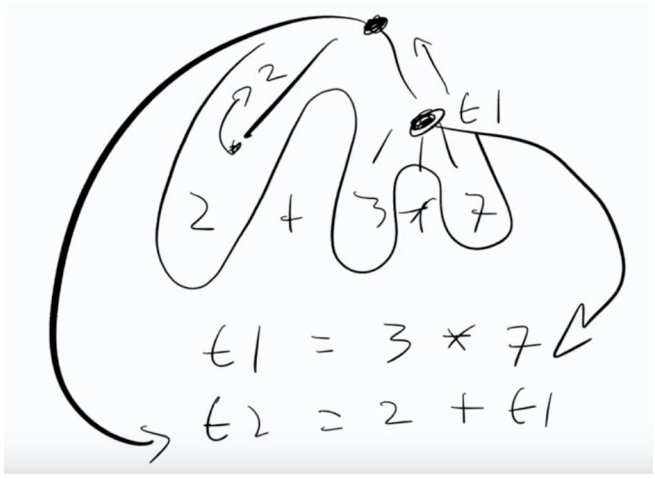
```
expression() {
  term();
  expression_prime();
}

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

To know which production to use:
Use the FIRST set 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.

Generating code for nested expressions:
Postorder traversal of parse tree.
Generate each node's LLVM IR
Return temporary variable to parent

Ex: 2 + 3 * 7




2
+
3
*
7
∈1
∈1 = 3 * 7
∈2 = 2 + ∈1