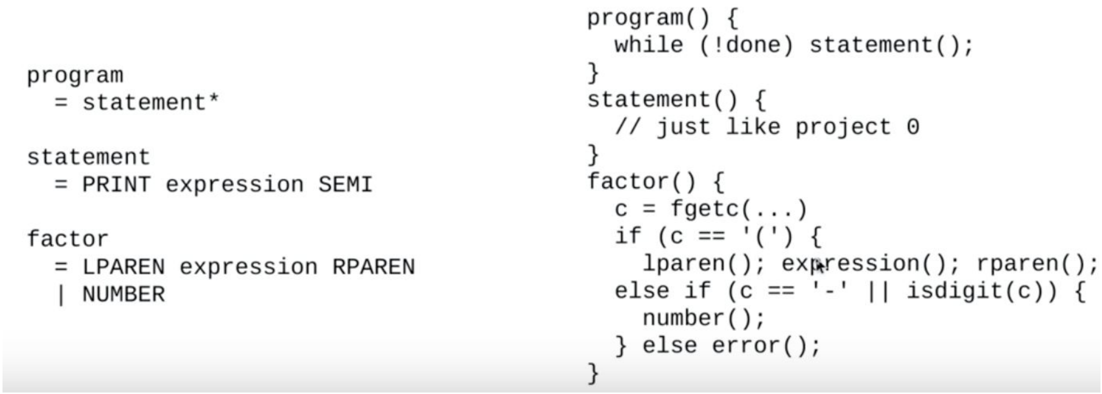
COP-3402 Systems Software
10/01 Thu

Refer to the */syllabus/projects/README.md* for updates on how to submit your projects via git. And another repository */grader-scripts* to test your projects with the test cases and they return the number of cases that passed out of all cases.

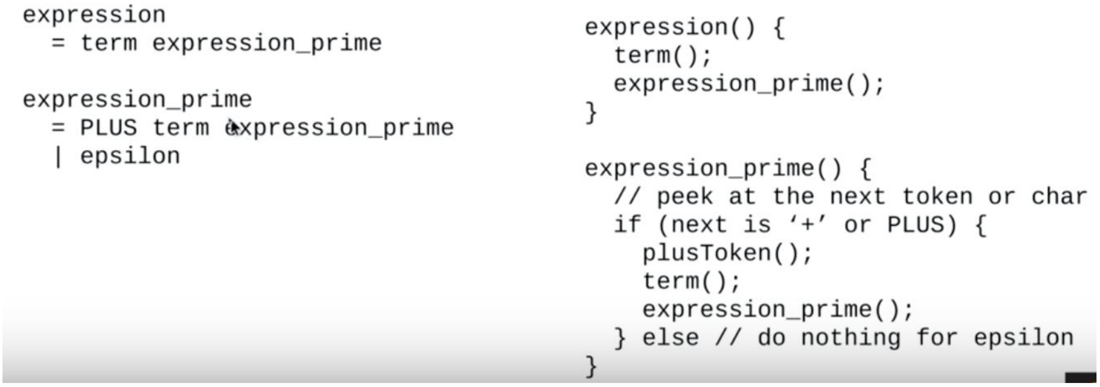
Refer to the *Recursive Descent Parsing* for the review on last lecture.
How we can modify our grammar so that we can not only do recursive descent parsing but do predictive parsing. Have grammar in order to enable recursive descent parsing grammar, make it a right recursive grammar instead of a left recursive grammar.



```
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();
}
```

Elimination enables recursive descent
Look at first token for each alternative: If first symbol is a nonterminal, follow the grammar



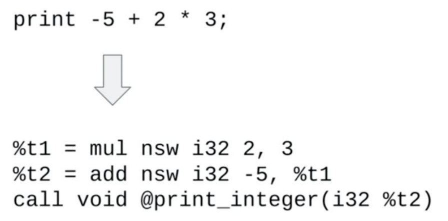
```
expression
  = term expression_prime

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
}
```

If we took the suffix of the production, all the stuff after the recursive part and turned that as prefix of new nonterminal.

Parsing first and then code generation.
Proj0: statements only have one operation;
Proj1: arbitrary expressions use many

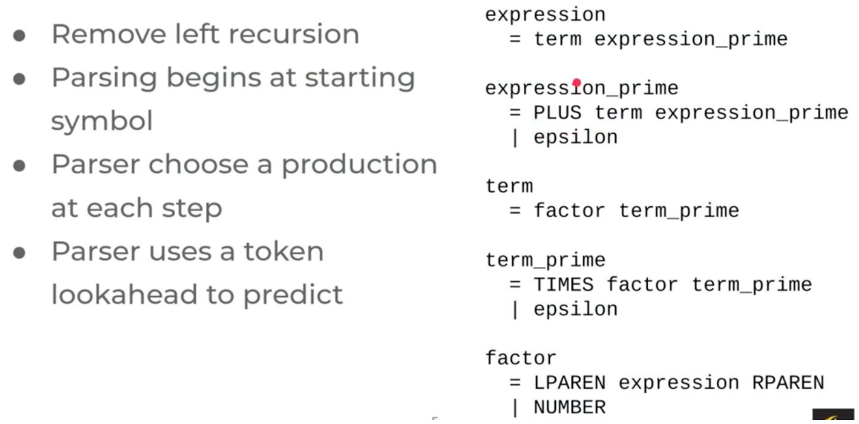
LLVM IR: one operation at a time.
Compiler needs to emit (print) and store each operation. We make temporary variables to store intermediate values as we are computing an expression.
Expressions use intermediate values:



```
print -5 + 2 * 3;
```

```
%t1 = mul nsw i32 2, 3
%t2 = add nsw i32 -5, %t1
call void @print_integer(i32 %t2)
```

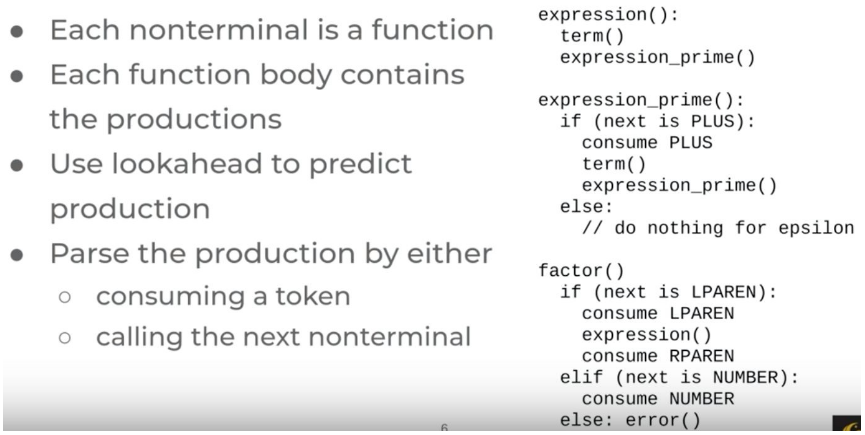
Review on *predictive parsing grammar* and its algorithm: (right recursive grammar)



- Remove left recursion
- Parsing begins at starting symbol
- Parser choose a production at each step
- Parser uses a token lookahead to predict

```
expression
  = term expression_prime

expression_prime
  = PLUS term expression_prime
  | epsilon

term
  = factor term_prime

term_prime
  = TIMES factor term_prime
  | epsilon

factor
  = LPAREN expression RPAREN
  | NUMBER
```

Recursive Descent Parsing algorithm:



- Each nonterminal is a function
- Each function body contains the productions
- Use lookahead to predict production
- Parse the production by either
  - consuming a token
  - calling the next nonterminal

```
expression():
  term()
  expression_prime()

expression_prime():
  if (next is PLUS):
    consume PLUS
    term()
    expression_prime()
  else:
    // do nothing for epsilon

factor()
  if (next is LPAREN):
    consume LPAREN
    expression()
    consume RPAREN
  elif (next is NUMBER):
    consume NUMBER
  else: error()
```

Adding code generation:

```
expression():
        left = term()
        result = expression_prime(left)
        return result
```

```
expression_prime(left):
        if (next is PLUS):
                consume PLUS
                right = term()
                result = newtemp()
                emit result “ = add “ left “, “ right
                expression_prime(result)
        else:
                // do nothing for epsilon
                return left
```

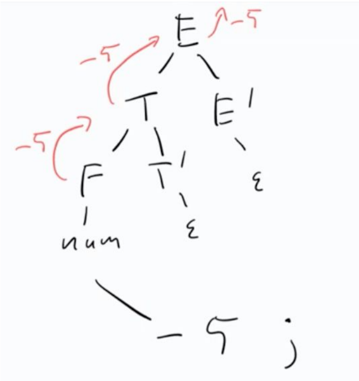
Ex. Parsing and Code Generation:

Grammar:

```
E -> T    E’
E’ -> +    T    E’
     | epsilon
T -> F     T’
T’ -> *      F      T’
    |   epsilon
F -> (E)
    |   num
```

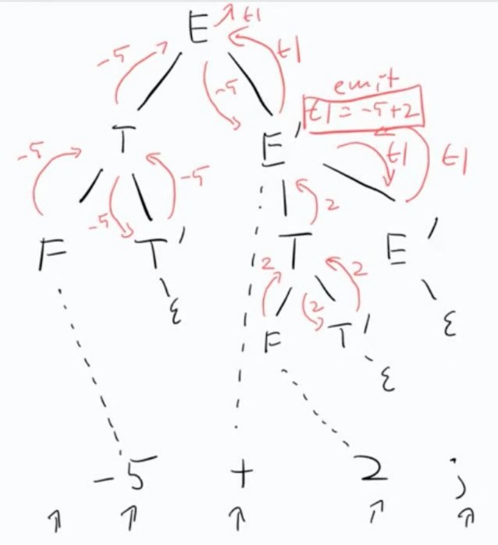
Ex.1:

print -5;



Ex.2:

print -5 + 2;



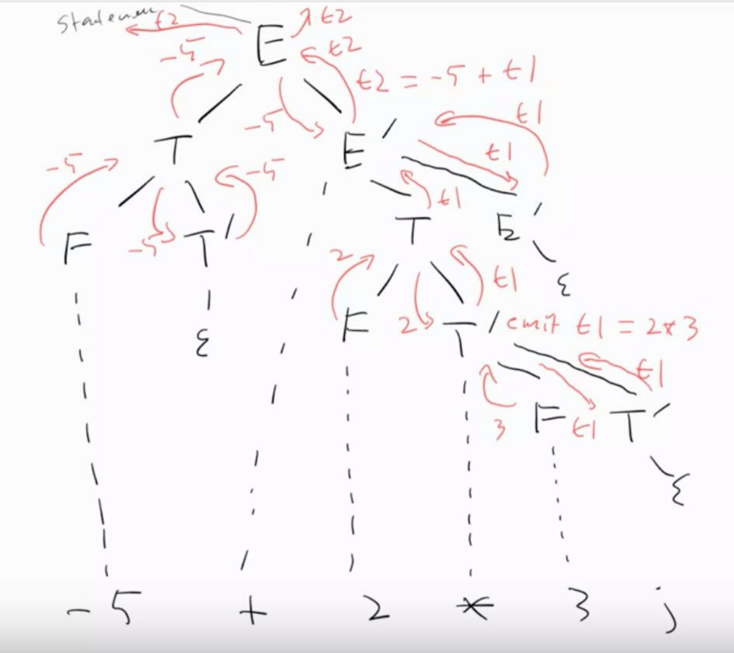
Post order tree traversal.

From left to right, -5 is passed from F to T, T to T', T' to T and T to E.

And -5 is passed to E', and then E' handles the addition. Similarly, same operations for 2 are applied. E passed -5 to E' and T passed 2 to E' and E' applies the addition. And E' creates temporary variable t1=-5+2.

Ex.3:

print -5 + 2 * 3;


Statement
t2
↑t2
E
t2
-5
t2 = -5 + t1
-5
t1
T
E'
t1
-5
-5
t1
-5
F
T'
T
E'
ε
2
t1
F
2
T
/emit t1 = 2*3
t1
3
F
t1
T'
ε
-5
+
2
*
3
;