COP-3402 Systems Software
09/24 Tue

cd project-test
git log => show you a list of changes to the code

touch testfile
git add testfile
git status
git commit
git log
git tag proj0
git log
git push --tags

Double-check that your tag is visible on GitHub. https://github.com/cop3402fall19/project-USERID/tags, where USERID is your GitHub ID. This is how we will know you have submitted your project. You will see your tags under Releases. Use these tags for each project:

Project	Tag
0	proj0
1	proj1
2	proj2
3	proj3
4	proj4
5	proj5

Resubmitting:
git status
git tag -f proj0
git push -f --tags

```
# be sure you are in a clean directory that does not have your project already
git clone https://github.com/cop3402fall19/project-USERID.git  # replacing USERID
with your github user id.
cd project-USERID  # replacing USERID with your github user id.
```

```
make
# run your tests
```

cd syllabus/projects/make
ls
make clean
ls
ls -lathr
make
=> will just build all the c files in your directory and then link them together into simple c program.

```
./simplec program.simplec > program.ll

# convert the .ll file to machine code

clang -o program program.ll

# run the resulting binary

./program
```

Project1:

Grammar

```
program
  = statement*

statement
  = PRINT expression SEMI

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

Check dragon book.
Expressions are arithmetic expressions with addition, subtraction, etc. Arbitrarily large number of operations.

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

Since SimpleC respects order of operations, the multiplication should happen first, followed by the addition. In LLVM, this would be

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

Context-free Grammars
Parsing:
Lexing recognizes words(lexemes)
Programming languages have nested structure
Regular languages cannot express nested structure
All regular languages can be pumped. Pump: strings of language are repeating pattern
Finite states for infinite set of strings - states will eventually be repeated (pigeonhole principle)
- Matching parentheses will need infinite states

Recognizing Nested Structure in Language:
Read one symbol at a time, just like lexers, for compilers one token at a time(instead of characters)
Match patterns of symbols, just like lexers
Track nested structure using a stack
Equivalent to a pushdown automaton
Finite automaton plus stack

Context-free Grammars Capture Nesting:
Expressions can contain nested structure

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

Context-free grammars use symbols to represent the structures
Language: infinite set of strings over a finite alphabet
Nonterminals represent constructs, not part of the string
Terminals represent symbols, part of the string
Nonterminals expand to smaller symbols
Terminals terminate the expansion

expression is nonterminal

Terminals are words. They are tokens from the lexer for compiler
Terminals cannot be broken down
Nonterminals represent structures

Productions are substitution rules
The starting symbol is the first nonterminal to replace

Identifier (variable names, etc.) - Terminal

Notations for Context-free Grammars
Chomsky:
Uppercase for nonterminals
Bold or punctuation for terminals
Arrows for productions
Pipe | for alternate productions
Parantheses are terminal

For this class:
lowercase for nonterminals
uppercase for terminals
Equal sign for productions
Pipe | for alternate productions

Derivation:
Demo

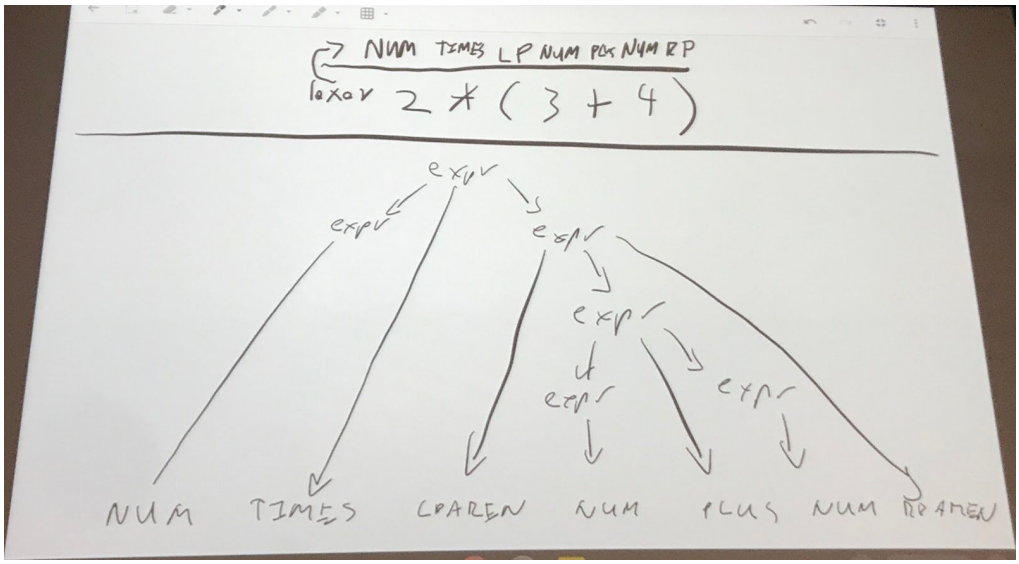
```
expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | LPAREN expression RPAREN
  | NUMBER
```

2 * (3+4)

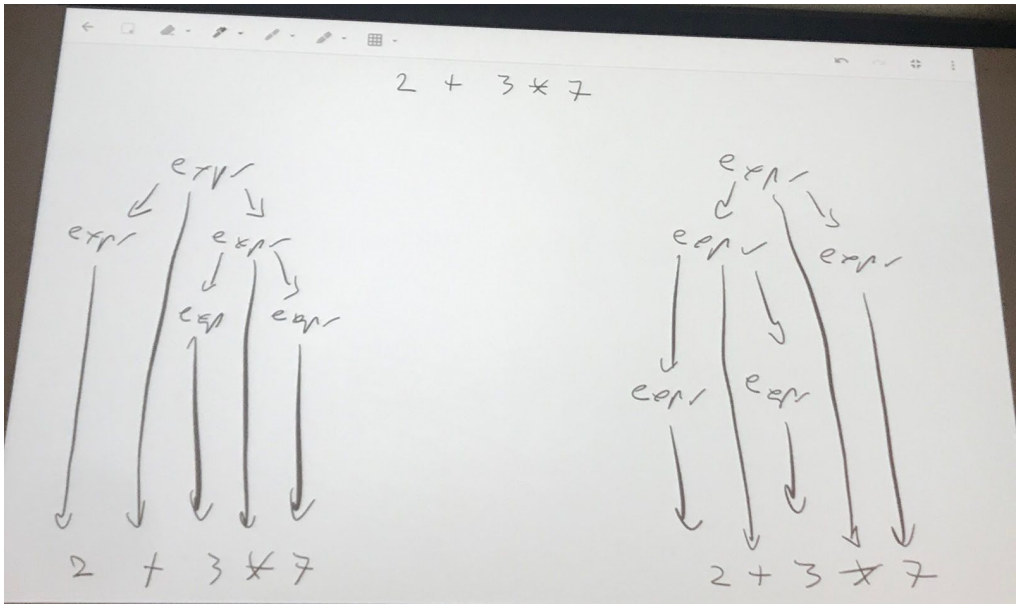
expression => expression TIMES expression
=> NUMBER TIMES expression
=> NUMBER TIMES LPAREN expression RPAREN
=> NUMBER TIMES LPAREN expression PLUS expression RPAREN
=> NUMBER TIMES LPAREN NUMBER PLUS expression RPAREN
=> NUMBER TIMES LPAREN NUMBER PLUS NUMBER RPAREN

Lexer -> NUMBER TIMES LPAREN NUMBER PLUS NUMBER RPAREN

Parse Trees:


NUM TIMES LP NUM PLUS NUM RP
lexer 2 * (3 + 4)
expr
NUM
TIMES
LPAREN
NUM
PLUS
NUM
RPAREN

Ambiguity:


2 + 3 * 7
expr
2 + 3 * 7
2 + 3 * 7