COP-3402 Systems Software
10.22 Tue

Project.2 Overview:
We are doing type checking just making sure that symbols have been declared before they're used and they're not multiple-defined.

Ex.

```
int integer;
integer = 0;
//It's valid.
```

Ex.

```
int x;
x = 5;
x = x + 5;
```

```
program() {
while (next token is INT) {
        declaration();
}
while (next token is not EOF) {
        statement();
}
}
```

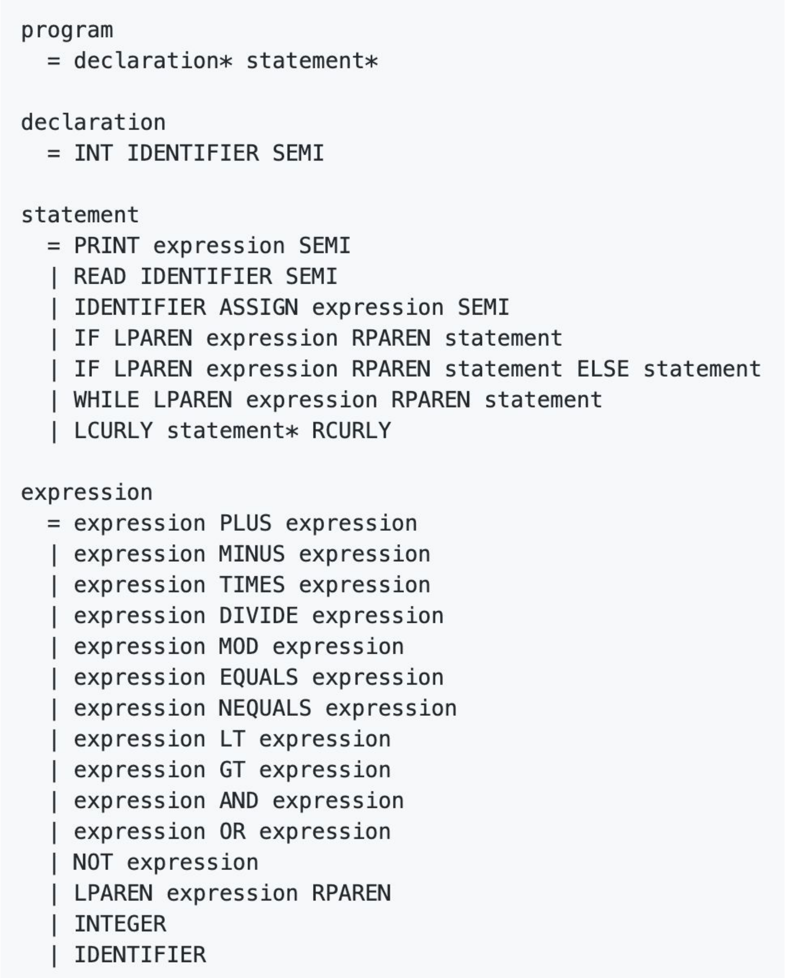
```
declaration(){
c= fgetc()
}
```

```
programprime(){
while (!done) {
        char *myident = identifier();
        if (myident is INT) {
        }
        else if (myident is PRINT) {
        }
        else if (myident is READ) {
        }
        else {
        }
        } }
```

// programprime = (declaration|statement)* // superset
// program = declaration*statement*

Project.3 Concept
https://github.com/cop3402fall19/syllabus/blob/master/projects/project3.md

Grammar:

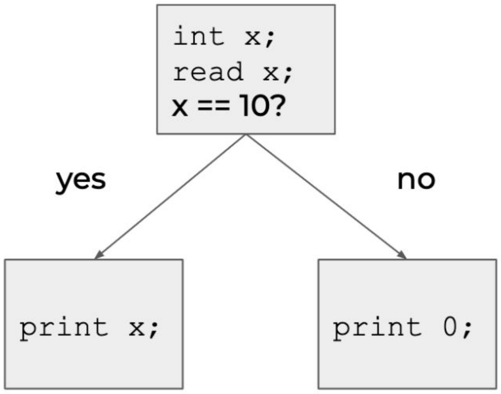


```
program
  = declaration∗ statement∗

declaration
  = INT IDENTIFIER SEMI

statement
  = PRINT expression SEMI
  | READ IDENTIFIER SEMI
  | IDENTIFIER ASSIGN expression SEMI
  | IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN statement ELSE statement
  | WHILE LPAREN expression RPAREN statement
  | LCURLY statement∗ RCURLY

expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | expression EQUALS expression
  | expression NEQUALS expression
  | expression LT expression
  | expression GT expression
  | expression AND expression
  | expression OR expression
  | NOT expression
  | LPAREN expression RPAREN
  | INTEGER
  | IDENTIFIER
```

Control Flow:
If-statements encode the decisions.
Uses boolean logic.

Ex:
int x;
read x;
if (x==10) {
print x;
}

```
else {
        print 0;
}
```



Decides which statement comes next as our program runs.

While-loops encode repetition.
Repeats instructions until a certain condition is met.

Ex. Gives us y^8:

```
int x;
int y;
read y;
x = 3;
while (!(x==0)) {
        y = y * y;
        x = x - 1;
}
print y;
```

OR;

```
x = 3;
loop:
if (!(x==0)) {
        y = y * y;
        x = x - 1;
        goto loop;
}
print y;
```

For-loops: (optional bonus project)
SimpleC does not have a for loop
We can express for-loops with while-loops.

Ex. Same as the while-loop above:

```
for (x=3; !(x=0); x = x - 1) {
        y = y * y;
}
print y;
```

Check out the new grammar for the Project3!

Boolean Expressions:
SimpleC has only AND, OR, EQUALS, and unary NOT.
See Project3 semantics for order of operations.

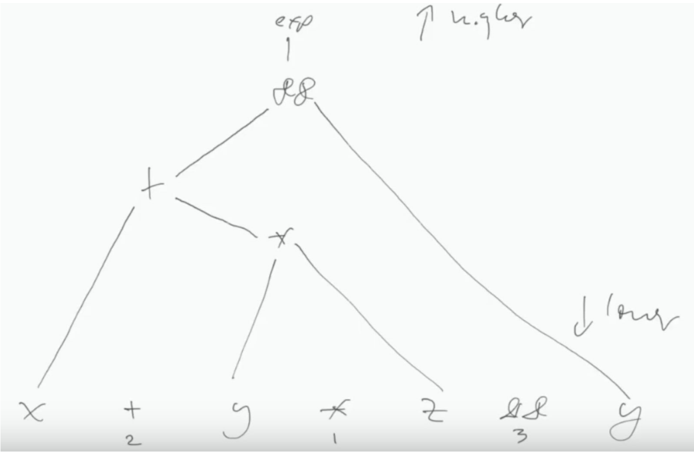
Operators have precedence, highest to lowest. Same line is equal precedence

```
 !
  * / %
  + -
  > <
  == !=
  &&
  ||
```

Parenthesized expressions have highest precedence.

Ex:

```
        x + y * z && y
Order:     2   1    3
```



Ex: x && y == !a || b
Order: 3 2 1 4

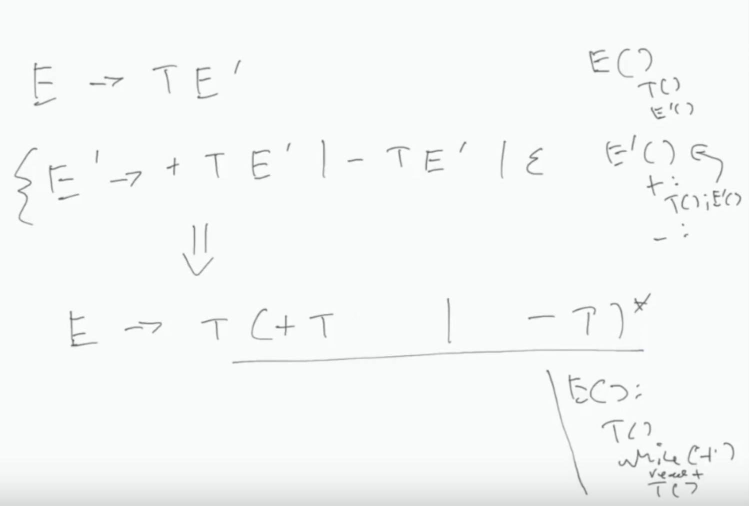
Boolean expression grammar is left recursive

```
expression = expression OR andexpr

andexpr = andexpr AND equalsexpr

equalsexpr = equalsexpr EQUALS addexpr

addexpr
  = addexpr PLUS term
  | addexpr MINUS term
```

```
term
  = term TIMES factor
  | term DIVIDE factor
  | term MOD factor

factor:
  = NOT expression
  | LPAREN expression RPAREN
  | NUMBER
  | IDENTIFIER
```

You can turn the right recursion into a while-loop:

```
expression():
  andexpr()
  while (lookahead is OR):
    andexpr()

andexpr():
  addexpr()
  while (lookahead is AND):
    addexpr()

addexpr():
  term()
  while (lookahead is PLUS or MULT):
    term()
```



$$E \rightarrow T E'$$

$$\left\{ E' \rightarrow + T E' \mid - T E' \mid \varepsilon \right.$$

$$\Downarrow$$

$$E \rightarrow T \underline{(+T \mid -T)^*}$$

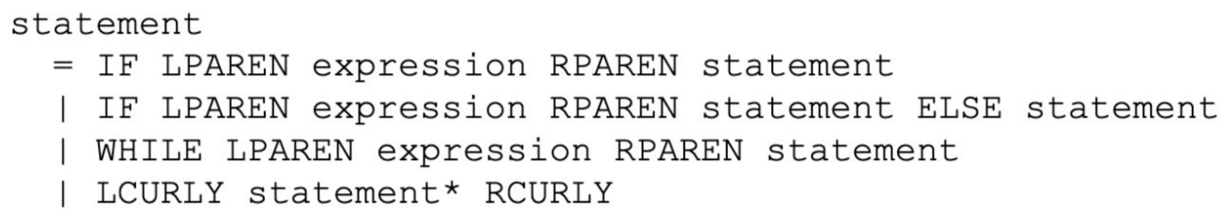
Control Flow Statement Grammar

3 new statements:

if-then-else

if-then

while

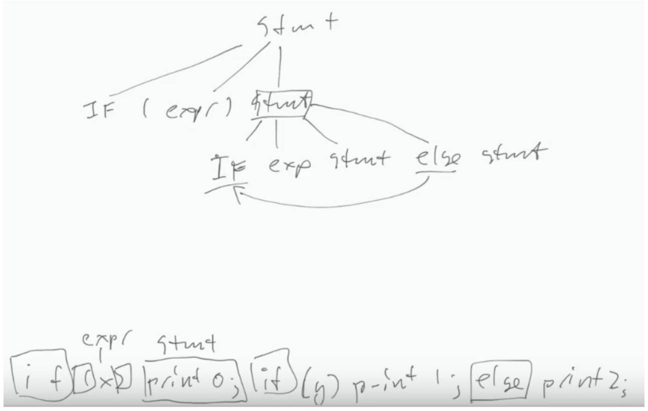


```
statement
  = IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN statement ELSE statement
  | WHILE LPAREN expression RPAREN statement
  | LCURLY statement* RCURLY
```

Demo: Parse Tree for If-Statements

Ex:

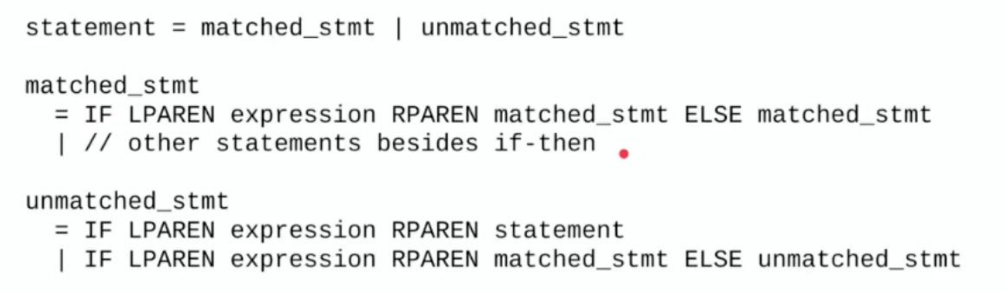
if (x) print 0; if (y) print 1; else print 2;



The if-statement grammar is ambiguous.

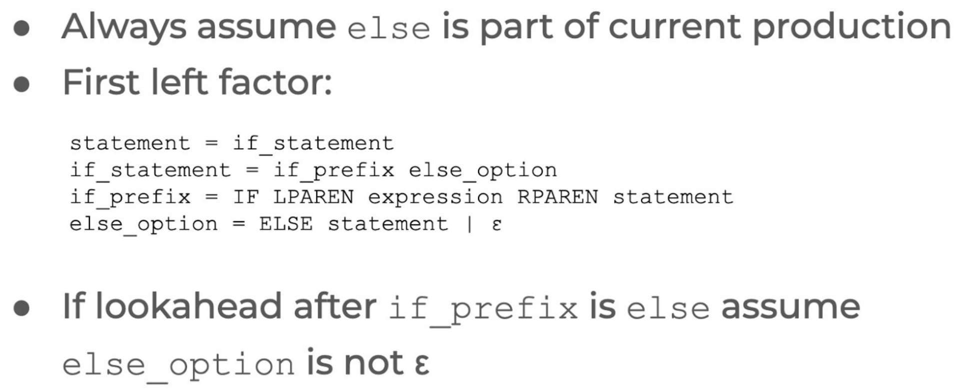
Resolving the dangling else: Match *else* to the nearest *if*

Method.1: Make matching explicit in the grammar



```
statement = matched_stmt | unmatched_stmt

matched_stmt
  = IF LPAREN expression RPAREN matched_stmt ELSE matched_stmt
  | // other statements besides if-then

unmatched_stmt
  = IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN matched_stmt ELSE unmatched_stmt
```

Method.2:

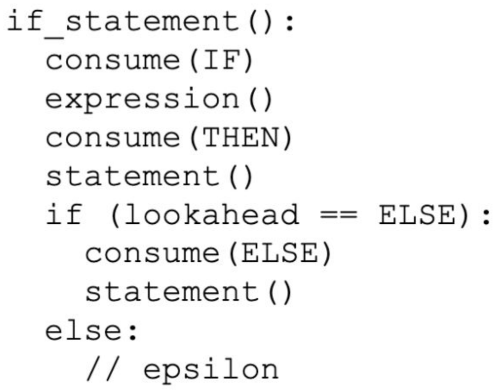


- Always assume `else` is part of current production
- First left factor:

```
statement = if_statement
if_statement = if_prefix else_option
if_prefix = IF LPAREN expression RPAREN statement
else_option = ELSE statement | ε
```

- If lookahead after `if_prefix` is `else` assume `else_option` is not ε

Pseudo-code for resolving dangling Else:



```
if_statement():
  consume(IF)
  expression()
  consume(THEN)
  statement()
  if (lookahead == ELSE):
    consume(ELSE)
    statement()
  else:
    // epsilon
```