COP-3402 Systems Software
11.05 Tue

Please refer to the grammar for Project#4.
https://github.com/cop3402fall19/syllabus/blob/master/projects/project4.md

Grammar

```
program
  = function function*

function
  = INT IDENTIFIER LPAREN ( INT IDENTIFIER (COMMA INT IDENTIFIER)* )? RPAREN LCURLY
declaration* statement* RCURLY

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
  | RETURN expression SEMI

expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | expression EQUALS expression
  | expression LT expression
  | expression AND expression
  | expression OR expression
  | NOT expression
  | LPAREN expression RPAREN
  | NUMBER
  | IDENTIFIER
  | IDENTIFIER LPAREN ( expression (COMMA expression)* )? RPAREN
```

Declarations will be global variables. Question mark in the grammar refers to that you might have a function with no parameters or, 1 identifier and zero or more comma identifiers.

Functions in SimpleC:
Function definitions appear after declarations.
Ex:

```
print square(9)     ->
print (9*9)         ->
print 81            ->
```

```
int square(int x){
        return x * x;
}
```

Return type should match its usage in expression.

Type checking:

- Untrapped errors during runtime

Declaring a function makes a new ‘operator’

- power: (int, int) -> int

```
int power(int base, int exp){
        int result;
        if (exp<0) return 0;
        if (exp==0) return 1;
        result =1;
        while(exp>0){
                result = result * base;
                exp = exp - 1;
        }
        return result;
}
```

Symbol Table for Functions:
Need to distinguish variables from functions
Name, type, address

- Type: function or variable (bonus: distinguish between int and bool)
- Function type: number of parameters (bonus: checking type of parameters and return type)

A function, like a variable, has an address.

Ex:

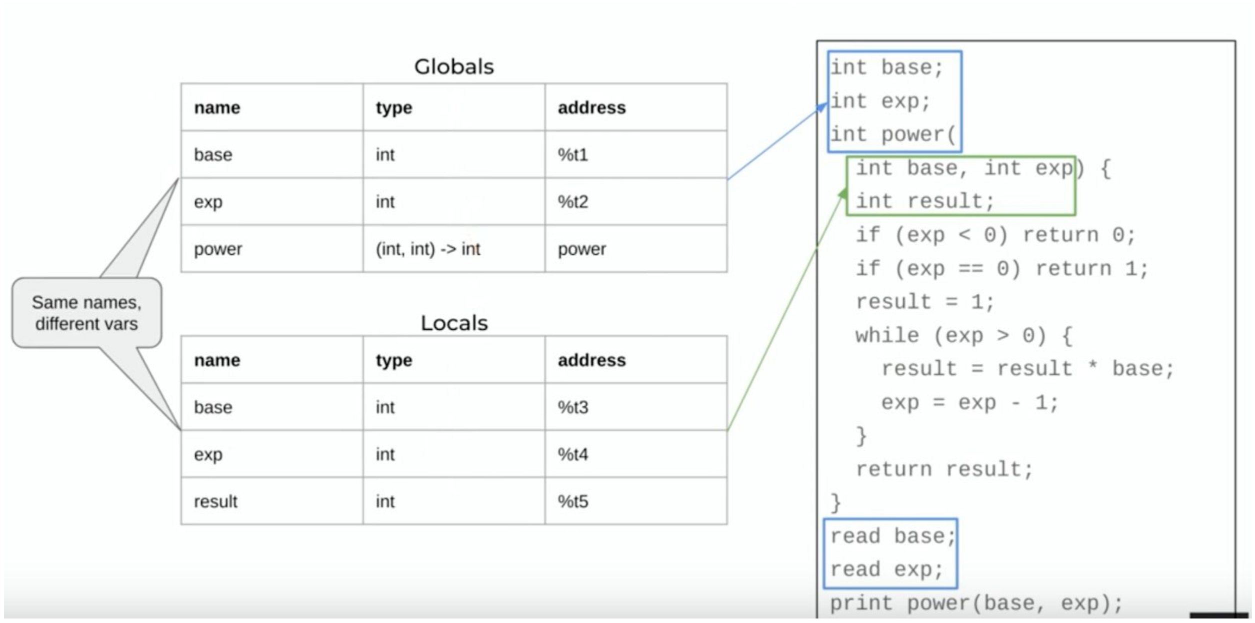
Name	**Type**	**Address**
base	int	%t1
exp	int	%t2
power	(int,int)->int	power

Static Scope: code region where variable is valid; SimpleC has 2 scopes, local and global.
Global: top-level declarations and functions
Local: parameters and declarations inside functions (bonus: compound statements also create new scope)
Local scope variables override global scope.

Ex:

```
int x;          // global variable
int f() {       // global function
        int x;  // a different local variable, global x not accessible
}
```

Symbol table distinguishes global and local
Create one global symbol table
Create a new local symbol table for each function
Destroy local symbol table once function finished



Globals

name	type	address
base	int	%t1
exp	int	%t2
power	(int, int) -> int	power

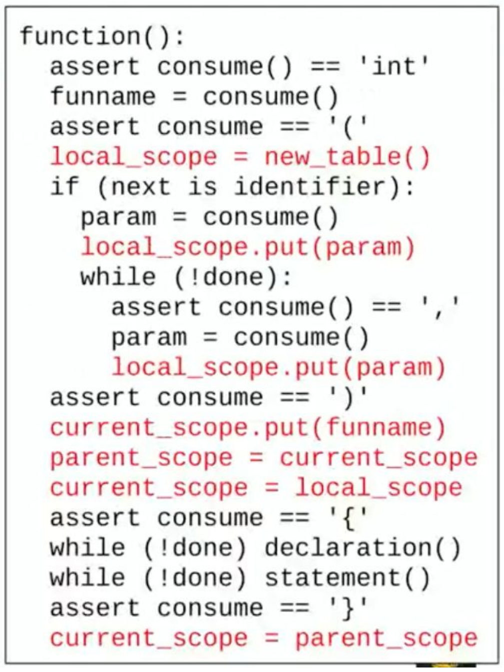
Locals

name	type	address
base	int	%t3
exp	int	%t4
result	int	%t5

```
int base;
int exp;
int power(
  int base, int exp) {
  int result;
  if (exp < 0) return 0;
  if (exp == 0) return 1;
  result = 1;
  while (exp > 0) {
    result = result * base;
    exp = exp - 1;
  }
  return result;
}
read base;
read exp;
print power(base, exp);
```

Managing global and local symbol tables:
Functions added to global symbol table
Functions parameters added to its local symbol table
Functions local variables added to its local symbol table

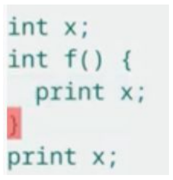
When in global scope, just check global table
When in a function, check local table first and then global table.



```
function():
  assert consume() == 'int'
  funname = consume()
  assert consume == '('
  local_scope = new_table()
  if (next is identifier):
    param = consume()
    local_scope.put(param)
    while (!done):
      assert consume() == ','
      param = consume()
      local_scope.put(param)
  assert consume == ')'
  current_scope.put(funname)
  parent_scope = current_scope
  current_scope = local_scope
  assert consume == '{'
  while (!done) declaration()
  while (!done) statement()
  assert consume == '}'
  current_scope = parent_scope
```

Functions update the scope:
Add function to global scope
Create new local scope
Switch to local scope, save the parent scope
Add parameters to local scope, check for duplicate parameters
Switch back to global scope

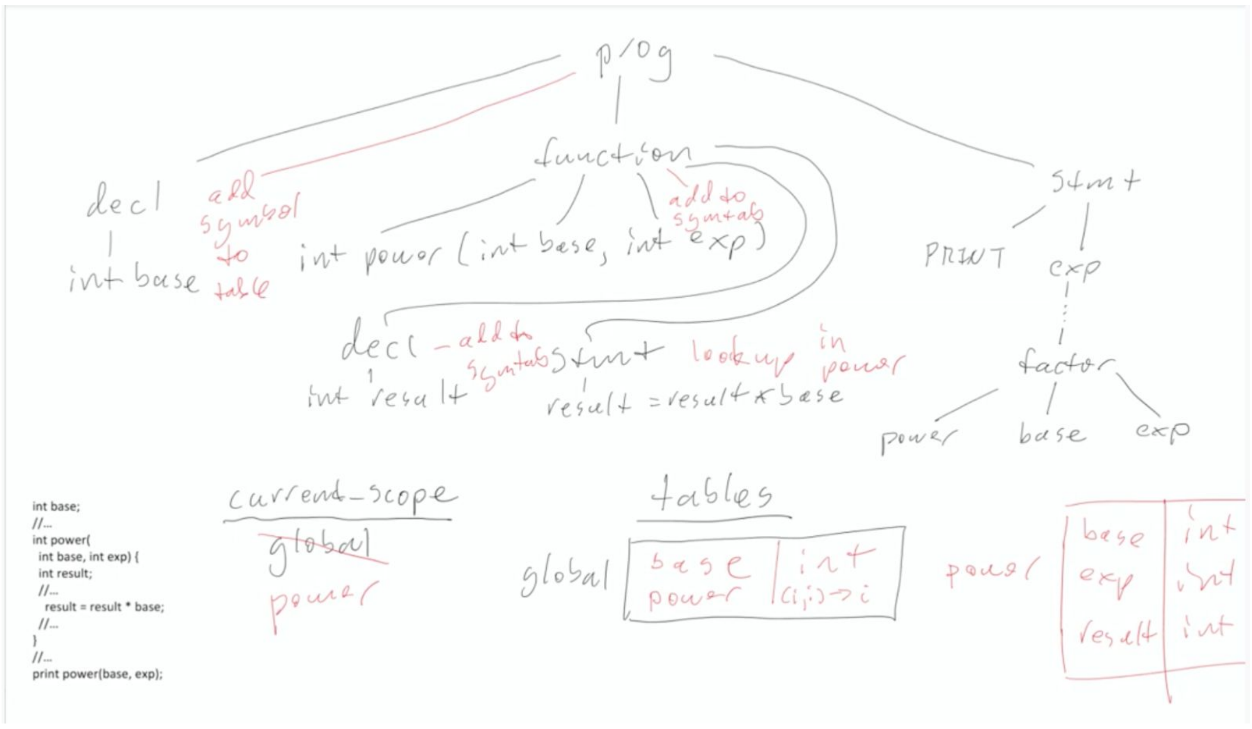
Variable lookups use the current scope.
In the assign(), read(), factor() tokens; we will not just lookup in the global scope, but we will have to lookup in the current scope, if it is not found, we will have to check the parent of the current scope. In our language, first check the local symbol table, if not, the global table, if still not, type error.



```
int x;
int f() {
  print x;
}
print x;
```

We can have a case where x is defined in the global scope, not in local. Then we lookup in the global table.

Ex: Parse tree



```
int base;
//...
int power(
  int base, int exp) {
  int result;
  //...
    result = result * base;
  //...
}
//...
print power(base, exp);
```