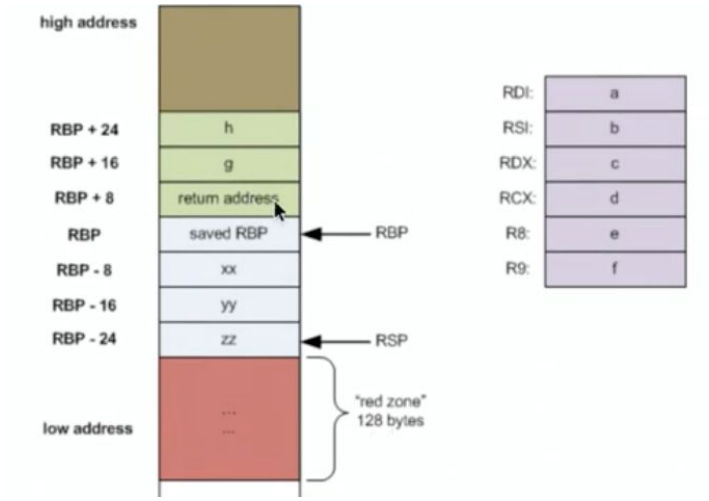
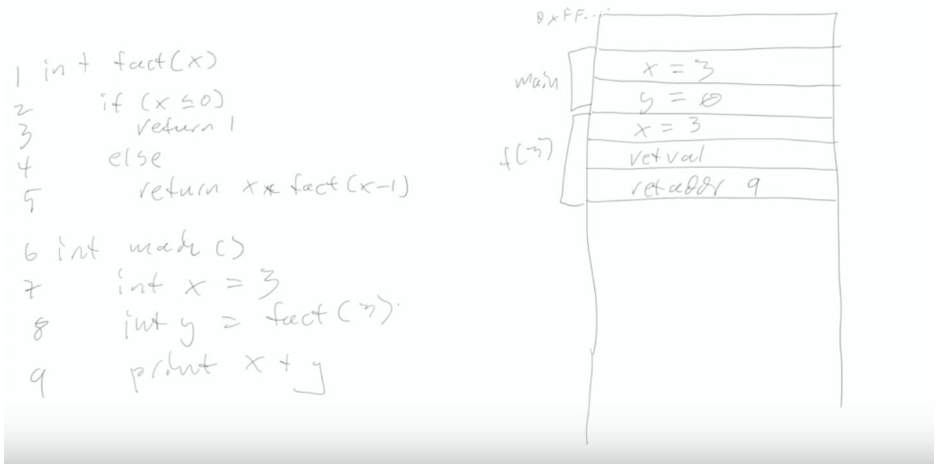
COP-3402 Systems Software
11.07 Thu

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

Whenever the function is called, the main function which is the caller here in this case, constructs a stack frame for factorial. And that stack frame will contain parameters, local variables, space for the return value and space for the return address. All this information is pushed onto the stack. And it is done by the main method.



Ex:



⇒

Simplification for PROJECT#4

GRAMMAR IS UPDATED!

No global variables; avoiding making global variables in LLVM

Global table only for functions; no need to lookup in the parent scope

Local table only for local variables and parameters

Return statement at the end of the function (bonus: return anywhere)

Functions should be defined before they are used.

You have to create your own main method of type () -> int. Therefore, you do not need the main in the template. You can just write a main function in simpleC and your compiler will turn that into an LLVM main method.

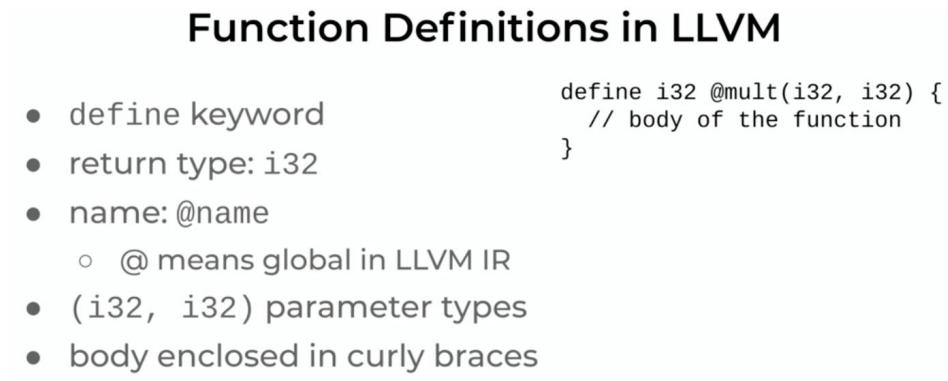
Grammar

```
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
```

Example in SimpleC:

```
int mult(int left, int right) {
  return left * right;
}
```



LLVM sets up the special registers as the parameters to the function. In our definition, there is no named parameters, instead LLVM sets the registers to hold our parameters:

- Parameter values are in %0, %1, etc
  - LLVM sets these up
- Allocate stack space for params
  - Why not just use %0, %1, etc?

```
define i32 @mult(i32, i32) {
  %left = alloca i32, align 4
  store i32 %0, i32* %left, align 4
  %right = alloca i32, align 4
  store i32 %1, i32* %right, align 4
  // body of function
}
```

Set up Symbol Table: (Treat parameters as locals, no need for special handling)

```
define i32 @mult(i32, i32) {
  %left = alloca i32, align 4
  store i32 %0, i32* %left, align 4
  %right = alloca i32, align 4
  store i32 %1, i32* %right, align 4
  // body of function
}
```

Local Scope

name	address
left	%left
right	%right

Generating Functions

- Emit the LLVM function return type and name
- Collect the parameter names (and types)
  - and emit the LLVM function parameters
- Create the local scope
  - and update the current_scope
- Emit the stack allocation for each parameter
  - and store its value
- Call declaration() and statement()
- Restore the current_scope back to pointer

Pseudocode for Function Code Generation

```
function():
  assert consume() == 'int'
  emit "define i32"
  funname = consume()
  emit "@" funname
  assert consume == '('
  parameters = []
  emit "("
  if (next is identifier):
    param = consume()
    parameters.add(param)
    emit "i32"
    while (!done):
      assert consume() == ','
      param = consume()
      emit ", i32"
      parameters.add(param)
  assert consume == ')'
  emit ")"
  current_scope.put(funname)
  assert consume == '{'
  emit "{"
  local_scope = new_table()
  parent_scope = current_scope
  current_scope = local_scope

  for i = 0 to parameters.len - 1:
    param = parameters[i]
    paramreg = newtemp()
    local_scope.put(param, paramreg)
    emit paramreg "= alloca"
    emit "store %" i "," paramreg

  while (!done) declaration()
  while (!done) statement()
  assert consume == '}'
  emit "}"
  current_scope = parent_scope
```

In the black script above, we have the parsing part of the grammar for project#4. For the red part code generation, for each piece of SimpleC, we are generating the corresponding piece of LLVM, just like 1-to-1 mapping from SimpleC to LLVM.

Emit the return statement:
Ex:
ret i32 %t3

Handling Function Calls:
Ex: (in the call of function)
%t5 = call i32 @mult(i32 %t3, i32 %t4)

Pseudocode for Function Code Generation

```
function():
  assert consume() == 'int'
  emit "define i32"
  funname = consume()
  emit "@" funname
  assert consume == '('
  parameters = []
  emit "("
  if (next is identifier):
    param = consume()
    parameters.add(param)
    emit "i32"
    while (!done):
      assert consume() == ','
      param = consume()
      emit ", i32"
      parameters.add(param)
  assert consume == ')'
  emit ")"
  current_scope.put(funname)
  assert consume == '{'
  emit "{"
  local_scope = new_table()
  parent_scope = current_scope
  current_scope = local_scope

  for i = 0 to parameters.len - 1:
    param = parameters[i]
    paramreg = newtemp()
    local_scope.put(param, paramreg)
    emit paramreg "= alloca"
    emit "store %" i "," paramreg

  while (!done) declaration()
  while (!done) statement()
  assert consume == '}'
  emit "}"
  current_scope = parent_scope
```

Input programs need the main to be executable.
clang -o myprog nomain.ll main.ll

Example:

Corresponding LLVM in the following link:
https://github.com/cop3402fall19/syllabus/blob/master/projects/examples/functions_example.ll

```
int factorial_recursive(int x) {
  int result;
  result = 1;
  if (x <= 0) result = 1;
  result = x * factorial_recursive(x - 1);
  return result;
}

int mult(int left, int right) {
  return left * right;
}

int factorial_iterative(int x) {
  int result;
  result = 1;
  while (x > 0) {
    result = mult(result, x);
    x = x - 1;
  }
  return result;
}

int main() {
  print factorial_recursive(4);
  print factorial_iterative(4);
  return 0;
}
```