UCF | College of Engineering and Computer Science

UNIVERSITY OF CENTRAL FLORIDA

# Code Generation for Functions

COP-3402 Systems Software

Paul Gazzillo

UCF

# Simplification of Functions

- No global variables
    - Avoids making global variables in LLVM
- Global table only for functions
    - No need to look in parent scope
- Local table only for local variables and parameters
- Grammar is updated
    - program is a list of function definitions
- return statement must be at end of function
    - I won't give you test cases otherwise
    - Bonus: allow return anywhere

# Example Function: Multiply

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

# Function Definitions in LLVM

- `define` keyword
- return type: `i32`
- name: `@name`
    - @ means global in LLVM IR
- `(i32, i32)` parameter types
- body enclosed in curly braces

```
define i32 @mult(i32, i32) {
  // body of the function
}
```

# Function 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
}
```

# Setup Symbol Table

- Treat parameters just like locals
    - No need for any 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 parent_scope

# 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
```

# Emit the Return Statement

- return is another kind of statement
- Just emit "ret" keyword followed by register

```
ret i32 %t3
```

# Handling Function Calls

- Store all functions in the global symbol table
  - Bonus: allow global variables as well
- Convert function call to LLVM

```
%t5 = call i32 @mult(i32 %t3, i32 %t4)
```

# Pseudocode for Call Code Generation

```
factor():
  funname = consume()
  retval = newtemp()
  emit retval " = call @" funname
  assert consume() == '('
  emit "("
  if (next is not RPAREN):
    actualparam = expression()
    emit actualparam
    while (next is COMMA):
      assert consume() == ','
      emit ","
      actualparam = expression()
      emit actualparam
  assert consume == ')'
  emit ")"
```

# Demo: Function Code Generation

- mult
- factorial

# Programs Need main to be Executable

- Input programs need a main
- If omitted, need linking
  - Need to allow external declarations
  - clang -o myprog nomain.ll main.ll