SimpleC Functions

COP-3402 Systems Software

Paul Gazzillo

Functions in SimpleC's Grammar

- (See project 4 grammar)
- Function definitions appear after declarations
- Factors now have function call syntax
- Same syntax for statements in/out or functions
- Our compiler implements the function *semantics*
  - Functions are unique global symbols
  - Functions have their own local variables and parameters
  - Function calls return a value in expression at the call site

Function Application

- (Conceptually) replace function with its result

```
print square(9) + 2;
```

```
print (9 * 9) + 2;
```

```
print (81) + 2;
```

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

- Return type should match its usage in expression

Type Checking

- What's the benefit of type-checking?

Type Checking

- What's the benefit of type-checking?
  - One is preventing untrapped errors during runtime

Type Checking

- Just like operators
  - < : (int, int) -> bool
    - "Less than" takes two integers and returns a boolean
- 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
- Original table: (name, address)
- New table: (name, type, address)
  - Type: function or variable
    - Bonus: distinguish between int and bool as well
  - Function type: number of parameters
    - Bonus: also need to check type of parameters and return type

Symbol Table for Functions

- Functions, like variables, have an address
- The address is its location in the code segment
  - Assembly: use its label
  - LLVM: use its LLVM function name

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

Static Scoping

- Static scope: code region where variable is valid
- SimpleC has only two scopes: local vs 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

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

Symbol Table and Scoping

- Symbol table distinguishes global and local
- Nested scopes implemented by either
  - A chain of symbol tables, or
  - An entry symbol table column for the scope
- SimpleC implementation is simpler
  - Only two scopes: global and local
  - Create one global symbol table
  - Create a new local symbol table for each function
  - Destroy local symbol table after the function

Symbol Table for Function Types and Scopes

Globals

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

Same names, different vars

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);
```

Simplified SimpleC Symbol Table

- All type are int
- Distinguish only variable and function
- Check number of arguments for functions

name	type	address
base	variable	%t1
exp	variable	%t2
power	function(3)	power

Managing Global and Local Symbol Tables

- Variable declarations still add to symbol table
  - Need to distinguish between global and local
- Functions added to global symbol table
  - (Some languages do allow nested functions)
- Function parameters added to its local symbol table
- Function local variables added to local symbol table
- Variable usage performs symbol table lookup
  - When in global scope, just check global table
  - When in a function, first check local table, then global

Pseudocode for Symbol Table Management

- Track the scope: current_scope (this can be a global var)
  - Start compiler in global scope
  - Enter function: switch to a new local scope
  - Exit function: destroy local table and restore scope
- Be sure to add functions to the global scope
  - Easy mistake is adding function name to its own scope
- Might need to check multiple tables for variable usage

Functions Update the Scope

- Add the function to global scope
- Create new local scope
- Switch to local scope
  - Save parent scope
- Add parameters to local scope
  - **Check for duplicate params**
- Switch back to global scope

```
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
```

Variable Lookups Use the Current Scope

```
assign():
  ident = consume();
  assert consume() == '='
  result = expression()
  assert consume() == ';'
  addr =
current_scope.lookup(ident)
  emit "store " result ", " addr
```

```
read():
  assert consume() == 'read'
  ident = consume()
  assert consume == ';'
  addr =
current_scope.lookup(ident)
  result = newtemp()
  emit result " = read_integer()"
  emit "store " result ", " addr
```

```
factor():
  // ... (the rest of the function)
  elif (next is IDENTIFER):
    ident = consume()
    addr = current_scope.lookup(ident)
    result = newtemp()
    emit result " = load " addr
    return result
  // ... (the rest of the function)
```

Demo: Functions and Scope