COEN 175

Phase 5

TAs

- Chris Desiniotis: cdesiniotis@scu.edu
  - Office Hours: Friday 12 - 2 pm
- Antonio Gigliotti: agigliotti@scu.edu
  - Office Hours: Thursday 11 - 1 pm

Extra Help/Tutoring

- Tau Beta Pi Tutoring
- Link to Tutoring schedule
  - https://sites.google.com/scu.edu/scutaubetapi/tutoring?authuser=1&pli=1

Phase 5 - Storage Allocation

After each step or substep, do "make". If you modify a header file, do a "make clean all" to make sure any changes to classes are detected.

1. Add offset to Symbol class
2. Write Type::size()
3. Write Function::generate()
4. Write function to generate code for global variables
5. Write stubs for some generate functions
6. Write operand functions
7. Overload stream operator for Expression
8. Complete generate functions

- **Due Wednesday March 3rd, 11:59PM**

1. Add offset to Symbol class

- Store the offset from the %ebp to the location of the symbol on the stack

- Add public member variable _offset to Symbol class
- Initialize to 0 in constructor

2. Write Type::size()

- Each Type has a size associated with it
- Modify Type.h and Type.cpp
- Rules:
  - “array of T”: (size of T) * length
  - “pointer to T”: 4 bytes
  - int: 4 bytes
  - char: 1 byte

3. Write Function::generate()

1. Setup
    a. Create generator.cpp and generator.h (all code generation will go in generator.cpp)
    b. Add virtual declaration for Function::generate() in Tree.h
    c. Modify parser to call Function::generate()
        i. In globalOrFunction() — after parsing a function definition, call generate() instead of write() (write() prints out the abstract syntax tree in a LISP-like syntax)
2. Implementation
    a. Assign offsets
    b. Prologue/Epilogue
    c. Generate code for body of function

3a. Assign Offsets

- Assign offsets to both parameters and declarations
  - Parameters offset should be positive
  - Declarations offset should be negative
- All variables are stored in a Scope, where the first N are parameters

3b. Prologue/Epilogue

- Use standard out to print out a function’s prologue and epilogue
  - Follow class slides for examples of these

3c. Generate code for body of function

- _body->generate()
  - Will write this generate() function in a later step

At this point

- Should be able to test Function::generate() with a simple function definition
  - Generate assembly and run you executable (nothing should happen)
- Example: **int main(int a, int b) { <*declarations*> }**
- What to test for:
  - Correct prologue/epilogue
  - Reserving the right amount of stack space
  - Assigning offsets to variables correctly
    - Can’t tell from your assembly
    - But, you can write offsets to stderr (e.g. a: 8)

4. Write function to generate code for global variables

- Generate the .comm directives for all global variables
- Declare generateGlobals() in generator.h
- Define generateGlobals() in generator.cpp
- Modify parser to call generateGlobals() when closing the global scope
    - Don’t forget to include generator.h

5. Write stubs for some generate functions

- Block::generate, Simple::generate, Call::generate, Assignment::generate
- Modify Tree.h so that each of these nodes declares generate()
- Modify generator.cpp with empty function definitions for generate()
    - Can be empty for now or have a simple cout

6. Write operand functions

- Expression::operand()
  - Create empty virtual function in Tree.h
- Number::operand()
  - $value
- Identifier::operand()
  - If global variable, refer to it by name (e.g. foo)
  - Otherwise, use its offset from the base pointer (e.g. -4(%ebp))

```
/*
 * Function:    Number::operand
 *
 * Description: Write a number as an operand to the specified stream.
 */

void Number::operand(ostream &ostr) const
{
    ostr << "$" << _value;
}
```

7. Overload stream operator for Expression

- Makes it easy to output expressions (rather than having to call operand())
- Call operand() within the stream operator

- Checker writer.cpp for example

8. Complete generate functions

Implement the following generate() functions:

- Block::generate()
  - Generate code for statements within a block
- Simple::generate()
  - _expr->generate();
- Assignment::generate()
  - Generate code for simple assignments
  - Use the *movl* instruction to move right operand into left
- Call:generate()
  - Generate code for a function call
    - Push arguments onto stack
    - Call function
    - Reclaim stack space if needed

Running/Testing

1. Generate assembly
    - ./scc < sample.c > output.s 2> /dev/null
2. Create executable with the gcc assembler
    - gcc -m32 output.s [*additional-source files*]
3. Run program
    - ./a.out

- With Dr. Atkinson's examples, if [test]-lib.c exists, you'll add the [test]-lib.c to gcc compilation steps
    - ./scc < ../examples/array.c > output.s
    - gcc -m32 output.s **../examples/array-lib.c**