COEN 175

Phase 5 - Week 8

TAs

Stephen Tambussi

Email: stambussi@scu.edu

Office Hours: Tuesday 1:00 - 2:00PM / Wednesday 4:00 - 5:00PM (Heafey Atrium)

Jordan Randleman

Email: jrandleman@scu.edu

Office Hours: Tuesday 9:10-10:10AM / Thursday 9:10-10:10AM (Heafey Atrium)

Extra Help / Tutoring

Tau Beta Pi Tutoring

- Thursday 2-3pm Heafey Atrium or its alcove prior the conference room
  - Location depends on table availability

How to Ask for Asynchronous Help

Do:

- Email Dr. Atkinson, Stephen, and Jordan all in the same email
- Include a *clean* (make clean) copy of your phase*N*.tar
- Include a detailed description of what's wrong
- Include a detailed description of what you've done to try to solve the problem

Don't:

- Email Dr. Atkinson, Stephen, or Jordan individually
- Send screenshots or individual files
- Just say "I have no idea what's going on" and dump a ton of code

The "don't"s are a sure-fire way to shoot your email to the bottom of the priority list!

Reminder: the New Lab Policy

- From Dr. Atkinson: You can only attend the lab you registered!
  - No more staying past 5 to the next lab
  - No more going to other labs to ask questions till you understand.
    - Dr. A is really emphasizing the need for y'all to learn how to write/debug programs on your own.
- If a TA figures out you already went to another lab, you will be kicked out.
  - Anything else risks the TAs directly going against a directive straight from Dr. Atkinson, and could literally get us fired (believe it or not, this does happen 🙃).
- If you want to go another lab ***instead*** of (not as a supplement to) your registered class, ***make sure you get permission from the TA before lab***.

Review of what the TAs ***are*** and ***are not*** here for!

There are 3 kinds of questions you can ask:

1. I don't understand how to build a compiler
    a. **That's ok!** So long as you've checked the lab slides, PDF, and lecture slides–and you're still confused as to how to structure your compiler–that's what we're here for! Ask away :)
2. I don't understand C++
    a. **These are problems you have to figure out on your own.** You can Google each and every single one of these questions, C++ is a massive language with an even more massive amount of documentation on the web. You will get fired if you ask your employer how to use a string, where to find a function, or which method you should call on an object.
3. I don't understand how to program
    a. **This is a call for introspection if you truly should even be taking compilers.** Everybody makes silly mistakes now and again, but consistently neglecting to run the code in your head, or not visualizing how the compiler is navigating the data as you write your program, is a recipe for a disastrous failure in this course.

High-Level Overview: Phase 5

Goal: Begin generating code for storage allocation

Week 8 Objectives to do in lab (more on these in the following slides)

1. Add offset to symbol
2. Write Block::generate()
3. Write Simple::generate()
4. Overload Ostream and Write Operand Functions
5. Output ".comm" directives
6. Call Function::generate() from parser.cpp
7. Write Function::generate()

Week 8 Objectives to do after lab (more on these in the following slides)

1. Write Type::size()
2. Write Assignment::generate()
3. Write Call:generate()

General Guideline: All "::generate()" methods return void, are declared in Tree.h, and are implemented in generator.cpp (which you'll create)

Submission

- Submit a tarball of your cpps and makefiles in a folder called *phase5*
- Worth 10% of your project grade
- Due **THIS** Sunday, March 5th by 11:59 PM – ***THIS IS A 1 WEEK LAB***

Lab Objective 1: Add Offset to Symbol Class

- Why? To track the offset for symbols stored on the stack
  - So that we can refer to that symbol later on in a function body by offsetting from %rbp
- “_offset” should be a public int member in the Symbol class
- Initialize “_offset” to 0 in Symbol’s constructor

Lab Objective 2: Write Block::generate() in generator.cpp

1. Add “virtual void generate();” to Tree.h in Block’s class definition
   a. In order to virtually override the Block’s base Node::generate() function
2. A block is literally just a series of statements
   a. Hence loop through and generate each statement in the block’s vector of Statement pointers
3. Logic:

```
for (auto stmt : _stmts) {
    stmt->generate();
}
```

4. ***Remember to add “generator.o” to your Makefile!***

Lab Objective 3: Write Simple::generate() in generator.cpp

1. Add “virtual void generate();” to Tree.h in Simple’s class definition
   a. In order to virtually override the Simple’s base Node::generate() function
2. The Simple class just represents a simple expression statement, so generating the code of a Simple object just means generating the code of its internal expression.
3. Logic:

```
_expr->generate();
```

Lab Objective 4: Overload Ostream and Write Operand Functions

- Add ostream declaration as the first function declaration at **the top** of generator.cpp

```
static ostream &operator<<(ostream &ostr, Expression *expr);
```

- Override *operand(ostream &ostr)* function in Expression, Number, and Identifier classes in Tree.h

```
virtual void operand(ostream &ostr) const
```

- Implement Ostream in generator.cpp
  - Call the polymorphic “operand” method on the expression (remember to pass in <ostr>!)
- Implement the *operand* member functions in generator.cpp
  - Expression::operand
    - Output nothing (it’s just an empty function)
  - Number::operand
    - Output $value (e.g. $4)
  - Identifier::operand
    - If the identifier is global (_offset == 0), just output its symbol name
    - Else output <offset>(%rbp) for the _offset field in its Symbol (e.g. -4(%rbp))

Lab Objective 5: Output “.comm” Directives in generator.cpp

- Write *void generateGlobals(Scope *scope)* in generator.cpp
  - “generateGlobals” takes in the global scope as its argument
  - Outputs .comm directives to stdout for all non-function symbols in the given scope
    - Format: .comm *name*, *size*
- Add *generateGlobals*’s declaration to generator.h
  - This should be the only declaration in “generator.h”
- Call it in parser.cpp by passing it the return value of closeScope in main()
  - E.g. add in “generateGlobals(closeScope());” in parser.cpp’s main()

Lab Objective 6: Call Function::generate() from parser.cpp

Replace “function->write(cout);” with “function->generate();” in parser.cpp

- Instead of printing out a function’s information, we now want to generate its assembly code!
- You’ll implement *void Function::generate()* later in this assignment.

Lab Objective 7: Write Function::generate() in generator.cpp

- Add “virtual void generate();” to Tree.h in Function’s class definition
  - In order to virtually override the Function’s base Node::generate() function
- When referring to params + local decls, use the Symbols array from the _body Block’s scope in Function
  - This array contains all the function’s parameters and THEN its declarations
- Allocate (e.g. set the offsets) for the function’s symbols (stored in *_body->declarations()->symbols()*)
  - offsetCounter starts at 0
  - For each symbol:
    - Subtract that symbol’s size from the offsetCounter
    - Set that symbol’s “_offset” to be “offsetCounter”
      - This represents where in the stack we place the symbol’s value relative to %rbp
- Make sure the stack is 16-byte aligned
  - E.g. make sure that offsetCounter is a multiple of 16, and make it so if it isn’t after allocating symbols
- Output the label for the function name followed by a colon: *_id->name()* followed by “:”
- Generate the function prologue (from lecture slides)
  - In the 3rd prologue instruction, make sure to “subq” “-offsetCounter” to only allocate the amount of space needed to hold local vars in the function
- Spill parameters (stored in registers from the caller) using “movl” to the stack
  - movl register_i, symbol_i->_offset(%rbp)
  - register_i ::= %edi, %esi, %edx, %ecx, %r8d, %r9d
  - symbol_i ::= one of the parameters in the Function object (stored in *_id->type().parameters()*)
- _body->generate()
- Generate the function epilogue (from lecture slides)
- Output the “.global” directive with the function name stored in *_id->name()*
  - .globl *functionName*

After Lab Objective 1: Write Type::size() in Type.cpp

- Delete the old implementation from Type.h (you'll rewrite it)
- Remember to declare it in Type.h
- Int is signed and needs 4 bytes
- Long is signed and needs 8 bytes
- Char is signed and needs 1 byte
- Pointers are always 8 bytes
- Arrays need to account for length of array
  - Arrays are contiguous sets of objects in memory
  - Hence: size(array of type T) = (size of T) * (length of array)

After Lab Objective 2: Write Assignment::generate() in generator.cpp

- Left will always be a scalar variable of type int
- Right will always be a literal of type int
- Hence we can just directly “movl” the right argument into the left argument

After Lab Objective 3: Write Call::generate() in generator.cpp

1. “movl” each _arg into the appropriate register
   a. Argument registers are: %edi, %esi, %edx, %ecx, %r8d, %r9d
2. Call the function
   a. call *functionName*

Checking your code

- **$ scc < file.c > file.s**
- **$ gcc file.s [additional-source-files]**
- **$ ./a.out**

- You don’t need to change any report() since those go to stderr
- Make sure you are sending your generated code to stdout
- Run with CHECKSUB.sh before submission
  - See the lab PDF for more details on how to check the assembly that GCC generates to compare it against yours (and the caveats inherent to doing so!)

Tips

- ***Carefully read through and understand the data structures in Tree.cpp/h***
  - The TAs will **NOT** answer questions along the lines of “how do I access X in Y?”
- Recompile your code frequently to make sure it still works
- ***Don’t forget to add your generator to your Makefile***
- Since you are overriding superclass member functions, you don’t need to have the others completed to *compile*
- Only need one generate() call in parser, the rest will be called recursively
- ***Generate comments*** with the “**#**” prefix (instead of “//”) when generating your assembly code (e.g. “**#** Calling a function”, etc.)
  - Helps to debug what your compiler spits out