Nolife Compiler (Part 2):
A Code Generator
CS 5810: Fall 2018

Due Date: Friday, December 7, 2018 at *5pm*

Purpose

This project is intended to give you experience building a code generator. You will use the AST developed in part 1 (or my AST). You will get experience generating assembly language that encodes the semantics of a Nolife program.

Project Summary

Your task is to construct a code generator for the Nolife programming language. The code generator will work as a pass over the AST constructed using your Nolife part 1. The output of your program will be a listing of the Intel IA-32 assembly code for the input program. Specifically, your project must

1. construct a memory layout for each procedure, and
2. generate low-level assembly code for the program.

These tasks can be performed in two separate passes over the tree, or they can be compressed into a single tree-walking pass. The next two subsections describe these tasks in more detail.

Memory Layout

In each procedure, you must allocate space for each of the variables declared locally. In Nolife, all variables can be assigned space in the activation record of the declaring procedure. There is no notion of a static variable — no variable retains its value across multiple invocations of the procedure that declares it. Variables declared with whole-program scope may be put on the activation record for the main procedure, or allocated to the static data area.

All variables should begin on an integral word boundary (4 bytes). The stack frame will begin on an integral word boundary. Variables can be assigned space in the activation record in any convenient order.

For each procedure, your compiler should produce a storage layout. It should list the name and offset of each variable that has space allocated in the current procedure. For arrays, it should list the upper and lower bound. Integer constants need no data storage. They can be represented in the code space as part of the `mov` instructions. String and floating point constants are not so easy. You will need to create space in the global data area to hold these constants. Assume that the locations for constants starts at a the label `_constant`. That is, you can get the offset of the constant data area into a register with the instruction :

```
mov %edi, offset flat:_constant
```

The linker will translate occurrences of `_constant` into the proper run-time address. `_constant` should be in a read-only data segment. The data segment should be before the text segment where the program resides. Begin the data segment with the `.section .ro_data` pseudo-op. Character constants should be declared with the `.string` pseudo-op and the literal should be in double quotes. Floating-point constants should be declared with the `.float` pseudo-op and the literal can be printed in a form acceptable to C.

If you allocate global variables in the global data area (as opposed to the activation record for the main procedure), you will need to use the following pseudo-op listed at the end of the assembly file following the last `ret` instruction:

```
.comm name,expr,align
```

where `name` is the name of the variable, `expr` is the size of the space allocated for the variable and `align` is the byte boundary multiple to which this variable is aligned.

Function Code

Your program should generate Intel IA-32 assembly for the entire Nolife program to which it is applied. See the Intel IA-32 assembly language manual on the CS4131 web page for specifics. All parameters can be passed on the stack to make things easy. Allocate all local variables on the stack for ease of implementation. You may use the general-purpose registers, but be aware of any assembly instructions that modify those registers.

Since local variables will be stored on the stack, we need to make space for them. This is done at entry to the function by modifying `%esp` and `%ebp`. Begin each function with the following sequence:

```
   .globl name;
   .type name, @function
name:
   push %ebp
   mov %ebp, %esp
   sub %esp, s
```

Here `name` is the name of the function and `s` is the space required for local variables and compiler temporaries on the stack. The main function should be named `main`. The locals and temporaries should be addressed as a negative offset off of the frame pointer (`%ebp`). Parameters are addressed with a positive offset off of `%ebp`. Stored at `%ebp` will be the old `%ebp` and stored at `%ebp+4` will be the return address. Therefore, the last parameter pushed on the stack before the call to the current function will be located at `%ebp+8`.

Make sure to end each function with the following sequence:

```
   leave
   ret
```

The `leave` instruction resets `%ebp` and `%esp` for the `ret` instruction. Make sure that you modify the stack pointer after each function call to remove the space for any passed parameters. The `ret` instruction does not do this.

Floating-point Code

The `x87` floating-point co-processor has 8 registers that are organized as a register stack. To push a floating-point value on the top of the register stack, use the `fld` instruction. To perform addition, subtraction and multiplication use `fadd`, `fsub` and `fmul`, respectively. A `p` may be added to the mnemonic to also pop the stack after the operation. To store the value at the top of the stack to memory, use the `fst` instruction, or the `fstp` instruction if you wish to store and pop the stack. The top of the register stack is referenced using `%st`. The i^{th} register off of the top of the stack is referenced using `%st(i)`. For example, if `x` is 4 bytes off the frame pointer, `y` is 8 bytes and `z` is 12 bytes of the frame pointer the statement

```
z = y + x
```

may be implemented with the following sequence:

```
fld dword ptr [%ebp-8]
fadd %st, dword ptr [%ebp-4]
fstp dword ptr [%ebp-12]
```

The first operand is put on the register stack, the top element on the stack is added to a value in memory and the result is stored at the current top of the register stack (not pushed). Finally, the new value on top of the stack is stored in memory and popped from the stack. All floating-point arithmetic operations may use two register stack operands or one register stack operand and a memory location. If more than 8 items are pushed on the register stack, it will overflow and wrap around.

Type conversions between floating-point and integer values are done using the x87 register stack. To convert a floating-point value stored 4 off of the frame pointer to an integer value stored at 8 off of the frame pointer, emit the following:

```
fld dword ptr [%ebp-4]
fistp dword ptr [%ebp-8]
```

The `fistp` converts the floating point value on the top of the register stack to an integer, stores it in memory and pops the stack. To convert the same integer value to a floating point value use

```
fild dword ptr [%ebp-8]
fstp dword ptr [%ebp-4]
```

The `fild` instruction converts the integer value to a floating-point value and pushes it on the register stack.

I/O

To print an integer with a `WRITE(1)` statement, you might generate code like

```
     ...

     .section   .ro_data
.int_format:
     .string   "%d\012\0"

     ...

     push 1
     push offset flat:.int_format
     call printf
     add %esp,8

     ....
```

I/O for floating-point values requires the data to be in quadword format. A single-precision floating-point value can be converted to double precision (quadword) using the x87 floating-point stack. The following code will print a floating-point value.

```
     fld dword ptr [%ebp-4]
     sub %esp, 8
     fstp qword ptr [%esp]
     push offset flat:.float_format
     call printf
     add %esp,12
```

For reading and writing values of various types, create a C program that does what you want to do in Nolife, use the following command to generate assembler

```
gcc -fno-stack-protector -masm=intel -S file.c
```

Look at the resulting assembler in `file.s` to determine the sequence of instructions to generate a call to `scanf` and `printf` for various types. This can also be used to see how to generate code for various other constructs.

Branches

Conditional branches are implemented based upon a condition code. Use the `cmp` instruction to set the condition code for integer compares and the appropriate jump instruction to branch (*e.g.*, `jge`). You do not need to use short-circuit code. To compare floating-point values, use the `fcomi` instruction to compare floating-point values and set the EFLAGS register so that conditional jumps can use the result of the compare directly. Note that `fcomip` additionally pops the x87 register stack.

Requirements

Your code will be tested under 32-bit Ubuntu 16.04. Your code should be well-documented. You will submit all of your files, including a `Makefile` to build your compiler, or an Eclipse project, in a zip file name `Nolife2.zip`.

f you use Eclipse, submit a zip file containing your Eclipse project. You must include a jar file named nlc.jar that includes where your main routine is. I expect to type `java -jar nlc.jar foo.nl` and have the assembly dumped to the console.

If you choose not to use Eclipse, you must submit all of your files, INCLUDING a `Makefile` to build your compiler. I will just type "make" and expect everything to "come out right." (Come out right in this context means I can type "`nlc foo.nl`" and I will get the assembly for foo.nl dumped to the console.

The storage layout should be written to a file called `<file>.map`, where `<file>` is the input file name.

Your program will be graded on whether it produces correct `IA-32` assembly code for all correct programs and a correct storage layout. "Correct code" means that assembling and executing the program produces the correct answer. The set of test programs for this project can be found in the file `CodeGeneratorTestfiles.zip` on eLearning.

As a final note, the object of this project is not to generate the best code, but rather working code. Working slow code is better than broken fast code. If you are interested in compiler optimization, consider taking `CS 6810`.