CS 380C Compilers — Spring 2015	Programming Assignment #2
Local Optimizations	**Due February 13, 2015**

In this assignment, you will use LLVM to learn interesting properties about programs and to perform local optimizations. As usual, any clarifications and corrections to this assignment will be posted on the class discussion board on Piazza.

1 Inspect the Functions

A compiler pass is the standard mechanism for analyzing and optimizing programs, and your first task is to implement an analysis pass. In general, analysis passes are used to ensure that an optimization pass is safe, ie, it does not alter the meaning of the program, but for this assignment, you will implement a simple `FunctionInfo` pass to learn interesting properties about the functions in a program. Please name your pass `FunctionInfo` and make sure that the `opt` command-line option for it is `-function-info`. Your pass should report the following information about **all** functions that are used in a program:

1. Name.
2. Number of arguments.
3. Number of call sites (i.e. locations where this function is called). All call sites in the module should be counted except for those indirect calls through function pointers.
4. Number of basic blocks.
5. Number of instructions.

For each function, print a line (to standard error) in the following format:

```
functionName: arguments=2, call sites=1, basic blocks=5, instructions=50
```

We recommend that you debug your pass with complex source files, as you can imagine grading will be done with complex programs. Feel free to turn in your additional testing source files in a separate directory, along with your source code.

2 Optimize the Blocks

Now that you know how to create LLVM passes, it's time to write an optimization pass. The LLVM developers have already implemented the most common local optimizations, so you will implement new set of optimizations that apply to invocations of a specific arithmetic library. For example, you could imagine that this library supported arbitrary precision arithmetic, which is useful for things such as public key cryptography. Your pass, then, would optimize programs that invoke these arithmetic library operations; you will not have access to the library's source code, and you will not be expected to change the library itself, just the application code that invokes the library.

More specifically, if the library provided simple bignum operations such as `bignum_add`, `bignum_mul` and `bignum_shl` , you would want your compiler pass to perform the following types of domain-specific local optimizations (here, assume that `x` is a variable of type `bignum`):

1. Algebraic identities: For example,
   - `bignum_add(x, bignum(0))= bignum_add(bignum(0), x)= x`,
   - `bignum_div(x, x)= bignum(1)`
2. Constant folding: For example,
   - `bignum(2)* bignum(4)= bignum(8)`

3. Strength reduction: For example,

   - `bignum(2)* x = bignum_add(x, x)`

   or

   - `bignum(2)* x = bignum_shl(x, 1)`

3 Meet the Library

The `bignum` library discussed in the above section is a motivating example, but it introduces the complexity of dealing with arbitrary length numbers. To avoid this complexity, you will work on a fabricated arithmetic library `libarith.a` that operates on top of built-in LLVM integer types. Specifically, the API exposed by the library is as follows:

- `lib_add()`, `lib_sub()`, `lib_mul()`, `lib_div()`, `lib_rem()`
  These functions perform 32-bit unsigned addition, subtraction, multiplication, division, and remainder operations, respectively. The result of overflowing arithmetic as well as division by zero is undefined (i.e. you can do whatever you want in those cases). Quotient of division is truncated towards zero.
- `lib_add_wrap()`, `lib_sub_wrap()`, `lib_mul_wrap()`
  These functions perform 32-bit unsigned addition, subtraction and multiplication. If overflow happens, the result wraps around using twos-complement representation.
- `lib_and()`, `lib_or()`, `lib_xor()`, `lib_not()`
  These functions perform bitwise and, or, xor, and not operations, respectively.
- `lib_shl()`, `lib_shr()`
  These functions perform bitwise left shift and (logical) right shift operations, respectively.

To motivate strength reduction, each library call is associated with certain cost:

- The cost of addition, subtraction, shifting and all logical operations is 1.
- The cost of multiplication is 3.
- The cost of division and remainder is 26.

Given an LLVM bitcode file which is compiled from C source codes that uses functions from `libarith.a`, your job is to write an LLVM pass that perform local optimizations on the bitcode to reduce the total cost of the input program as much as possible. Use the provided test files for inspiration to figure out what algebraic and strength reduction optimizations might be useful. (*hint*: If you make your optimization framework general it should be easy to support all the various required cases without too much duplication of code.)

4 Implementation Details

You should create a new LLVM pass named LocalOpts.cpp (with name `my-local-opts` inside LLVM for `opt` commandline parameters). This should be in a directory called `assignment2`. Here are the steps to prepare an input bitcode file `input.bc` from a C source file `input.c`:

```
clang -emit-llvm -c input.c -I<path to arith_lib.h>
```

This second step is optional but highly recommended :

```
opt -mem2reg input.bc -o input.bc
```

The `mem2reg` pass converts stack allocated variables (`alloca`) to registers. Look at the output before and after running this and it should make sense. This will greatly simplify your optimizations because you will not need to worry about redundant loads and stores.

You can then run your pass as follows:

```
opt -load <path to LocalOpts.so> -my-local-opts input.bc -o output.bc
```

You can convert a .bc file into a more human-readable .ll file using

```
llvm-dis input.bc -o input.ll
```

If you want to link the .bc or .ll file with the library and produce an executable `input_prog`, here is the command:

```
clang input.bc <path to libarith.a>/libarith.a -o input_prog
```

Your pass will be tested on source files that may contain unrealistic amounts of local optimization opportunities.

5 Your Report

You should include with your submission a report that briefly explains what you did and how you did it. The point of this report is to help the TA understand your solution. Please be clear and concise in your writing.

We do not need a detailed description of your code, but if there are any interesting high-level thoughts about how you attacked the problem or about any interesting or tricky aspects of your implementation, those should be included. If you think that everything was obvious, you can say so. You can also talk about the scope of your solution, for example, if you were not able to complete all of it or if you decided to do anything beyond what we asked.

6 Submission

Submit via Canvas a single `tar.gz` or `tar.bz2` file that contains your source code in a directory with a `Makefile` that will build it. For this project, please name the directory `assignment2`. Make sure that your code builds correctly on the provided virtual machine and does not depend on any files outside the code that you submit.

As with the first assignment, the build system is not the subject of this class so feel free to help each other with it or post useful variants of the `Makefile` or `CMakeLists.txt`.

This assignment is due on Friday February 13th at 5:00pm.

Acknowledgments. This assignment was originally created by Todd Mowry and then modified by Arthur Peters and Jia Chen.