CS 745, Spring 2012
Homework Assignment 1

Assigned: Thursday, January 19
Due: Thursday, February 2, 9:00AM

Welcome to the Spring 2012 edition of Optimizing Compilers (15-745). We will be using the Low Level Virtual Machine (LLVM) Compiler infrastructure from University of Illinois Urbana-Champaign (UIUC) for our programming assignments. While LLVM is currently supported on a number of hardware platforms, we expect the assignments to be completed on x86 machines, since that is where they will be graded. Although LLVM works quite well on both Mac OS X and Windows, it is recommended that assignments be done in Linux, to increase the chances of getting technical support from the teaching staff.

The objective of this first assignment is to introduce you to LLVM and some ways that it could be used to make your programs run faster. In particular, you will use LLVM to learn interesting properties about your program and to perform local optimizations.

Policy

You will work in groups of two people to solve the problems for this assignment. Turn in a single writeup per group, indicating all group members.

Logistics

All clarifications (if any) to this assignment will be posted on the class discussion board on Piazza. Any revisions will be uploaded to the "assignments" page on the class web page.

In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15745-s12/public`

and *ASSTDIR* refers to the subdirectory *HOMEDIR*/`asst/asst1`.

1 Install LLVM

First download, install, and build LLVM 3.0 source code from http:://llvm.org. The reason to build LLVM from source is that pre-built binaries will not contain the symbols that will make it easier for you to debug your work. Download the LLVM-3.0 release rather than the head release from SVN because the head release may contain more bugs than the release. To get started, follow the instructions at http://llvm.org/docs/GettingStarted.html for your particular machine configuration. You do not need to build the gcc and g++ frontends from source, instead follow the provided instructions to install pre-built binaries. Be careful not to overwrite the current gcc install on your system (either /usr/bin/gcc or /usr/local/bin/gcc), rather install your LLVM gcc in a private directory and update your PATH environment variable appropriately so that you have llvm-gcc, llvm-g++, opt, etc., in your PATH. **Note that in order to use a debugger on the LLVM binaries you will need to pass —enable-debug-runtime –disable-optimized to the configure script.**

```
int g;
int g_incr (int c)
{
  g += c;
}
int loop (int a, int b, int c)
{
  int i;
  int ret = 0;
  for (i = a; i < b; i++) {
   g_incr (c);
  }
  return ret + g;
}
```

(a)

```
@g = common global i32 0

define i32 @g_incr(i32 %c) nounwind {
entry:
  %0 = load i32* @g, align 4
  %1 = add nsw i32 %0, %c
  store i32 %1, i32* @g, align 4
  ret i32 undef
}

define i32 @loop(i32 %a, i32 %b, i32 %c) nounwind {
entry:
  %0 = icmp slt i32 %a, %b
  %1 = load i32* @g, align 4
  br i1 %0, label %bb.nph, label %bb2

bb.nph:                                    ; preds = %entry
  %tmp = sub i32 %b, %a
  %tmp7 = mul i32 %tmp, %c
  %tmp8 = add i32 %1, %tmp7
  store i32 %tmp8, i32* @g, align 4
  ret i32 %tmp8

bb2:                                       ; preds = %entry
  ret i32 %1
}
```

(b)

Figure 1: (a) A simple loop source code, and (b) its optimized LLVM bytecode.

Peruse through the documentation at http://llvm.org/docs. The LLVM Programmer's Manual (http://llvm.org/docs/ProgrammersManual.html) and Writing an LLVM Pass Tutorial (http://llvm.org/docs/WritingAnLLVMPass.html) are particularly useful.

2 Create a Pass

The source code for your LLVM passes do not need to be inside the LLVM source tree. The Makefile rules below will help you build your passes regardless of where your source code is, as long as the LLVM that you have built is in your PATH (ie export PATH=/path/to/LLVM:$PATH). Create a directory (e.g, named `FunctionInfo`), and copy `FunctionInfo.cpp` (provided with the assignment) into the new directory. `FunctionInfo.cpp` contains a dummy LLVM pass for analyzing the functions in a program. Currently it only prints out "15-745 Functions Information Pass". In the next section, you will extend `FunctionsInfo.cpp` to print out more interesting information. For now, we will use the dummy LLVM pass to demonstrate how to build and run run LLVM passes on programs. First, create a `Makefile` to build the `FunctionsInfo` pass as follows (these instructions assume that your passes are the only `.cpp` files in the directory. Make sure that there are tabs on lines 6 and 8 below):

```
all: FunctionInfo.so

CXXFLAGS = -rdynamic $(shell llvm-config --cxxflags all) -g -O0

%.so: %.o
        (CXX) -dylib -flat_namespace -shared $^ -o $@
clean:
        rm -f *.o *~ *.so
```

(Note: you can also copy this code from *ASSTDIR*/`FunctionInfo/Makefile`.) Before moving on to the next section, make sure you can run this dummy pass properly. Copy the `loop.c` source code (shown in Figure 1(a)) from *ASSTDIR*/`FunctionInfo/loop.c` into your local directory. Compile it to an optimized LLVM bytecode (`loop.o`) as follows:

```
llvm-gcc -O -emit-llvm -c loop.c
```

Inspect the generated bytecode using `llvm-dis` as follows:

```
llvm-dis loop.o
```

This will create a disassembly of the testcase named loop.o.ll that should look very similar to Figure 1(b).

Now try running the dummy `FunctionInfo` pass on the bytecode using the `opt` command. (If you did not compile with debug information, the shared library (`FunctionInfo.so`) will be in the `Release` directory). Note the use of the command line flag "`-function-info`" to enable this pass. (See if you can locate the declaration of this flag in `FunctionInfo.cpp`). **Note that you must provide the correct path to FunctionInfo.so. You can use "./" if they are in the same directory.**

```
opt -load path/to/FunctionInfo.so -function-info loop.o -o out
```

If all goes well, "15745 Functions Information Pass" should be printed to stderr.

3 Meet The Functions

Program analysis is an important prerequisite to applying correct optimizations: i.e. without breaking the code. For example, before the optimizer can remove some piece of code to make a program run faster, it must examine other parts of the program to determine whether the code is truly redundant. A compiler pass is the standard mechanism for analyzing and optimizing programs.

You will now extend the dummy `FunctionInfo` pass from the previous section to learn interesting properties about the functions in a program. 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).
4. Number of basic blocks.
5. Number of instructions.

To assist you in writing this pass, the expected output of running `FunctionInfo` on the optimized bytecode (Figure 1(b)) is shown in Figure 1. As you can see, the output in Figure 1 is not interesting, since `loop.c` is a trivial piece of code. It is therefore recommended that you debug your pass with more complex source files, as you can imagine grading will be done with complex programs. Feel free to handin your additional testing source files in a separate directory together with your source code.

Name	# Args	# Calls	# Blocks	# Insts
g_incr	1	0	1	4
loop	3	0	3	9

Table 1: Expected FunctionInfo output for the optimized bytecode of `loop.c`

4 Optimize The Block (New Dragon Book 8.5)

Now that you are an expert writing LLVM passes, it is time to write a pass for making programs faster. You will implement optimizations on basic blocks as discussed in class. More details on local optimizations are available in Chapter 8.5 of the new Dragon book. While there are many types of local optimizations, we will keep things quite simple in this section and focus only on the algebraic optimizations discussed in Section 8.5.4 of the book. Specifically, you will implement the following local optimizations:

1. Algebraic identities: e.g, `x + 0 = 0 + x = x`
2. Constant folding: e.g, `2 * 4 => 8`
3. Strength reductions: e.g, `2 * x => (x + x) or (x << 1)`

4.1 Implementation Details

You should create a new LLVM pass named `LocalOpts.cpp` following the steps in Section 2. While it is possible to implement more than one pass in the same directory or file, it is probably much easier at this point to simply create a new `LocalOpts` directory in `llvm/lib/Analysis`. Since the `llvm-gcc` optimizer performs local optimizations, your `LocalOpts` pass should be run on unoptimized LLVM bytecodes. Unoptimized bytecode of `loop.c` can be prepared as follows:

```
llvm-gcc -O0 -emit-llvm -c loop.c
```

Now assuming the command line flag for enabling your local optimization pass is `-my-local-opts`, then you can run your pass as follows:

```
opt -load llvm/Debug/lib/LocalOpts.so -my-local-opts loop.o -o out
```

In addition to transforming the bytecode, your pass should also print out a summary of the optimizations it performed: e.g., how many constants were folded. We will provide toy source files with unrealistic amounts of local optimization opportunities for you to debug your pass in: *ASSTDIR*`/LocalOpts/test-inputs`. In addition to using these test inputs, we recommend that you test your pass on more realistic programs.

5 Questions

5.1 CFG Basics

For the code provided below (i) find basic blocks (ii) build the CFG (Control Flow Graph). Be sure to give your basic blocks clear labels (and label the original code to match).

```
    x = 100
    y = 0
    goto L2
L1: y = x * y
    if (x < 50) goto L2
    y = x - y
    goto L3
L2: y = x + y
L3: print(y)
    if (y < 1000) goto L1
    if (x <= 0) goto L5
L4: x = x - 1
    goto L1
L5: return y
```

5.2 Available Expressions, New Dragon Book 9.2.6

An expression x op y is *available* at a point p if every path from the entry node to p evaluates x op y, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y. For the *available-expressions* data-flow schema we say that a block *kills* expression x op y if it assigns (or may assign) x or y and does not subsequently recompute x op y. A block *generates* expression x op y if it definitely evaluates x op y and does not subsequently define x or y.
Based on this definition and the corresponding data flow analysis description(See Table 2 from New Dragon Book 9.2.7) perform Available Expressions analysis on the code in Figure 2.
In the tables below list the EVAL and KILL sets, and the final IN and OUT sets after AE is

Domain	**Direction**	**Transfer Function**	**Boundary**
Sets of expressions	Forwards	$gen_B \cup (x - kill_B)$	$OUT[entry] = \oslash$
Meet $\wedge$	**Equations**	**Equations**	**Initialize**
$\cap$	$OUT[B] = f_B(IN[B])$	$IN[B] = \bigwedge_{P,pred(B)} OUT[P]$	$OUT[B] = \cup$

Table 2: Available Expressions Analysis.

performed. You may ignore expressions inside conditional statements (e.g., (z < c)).

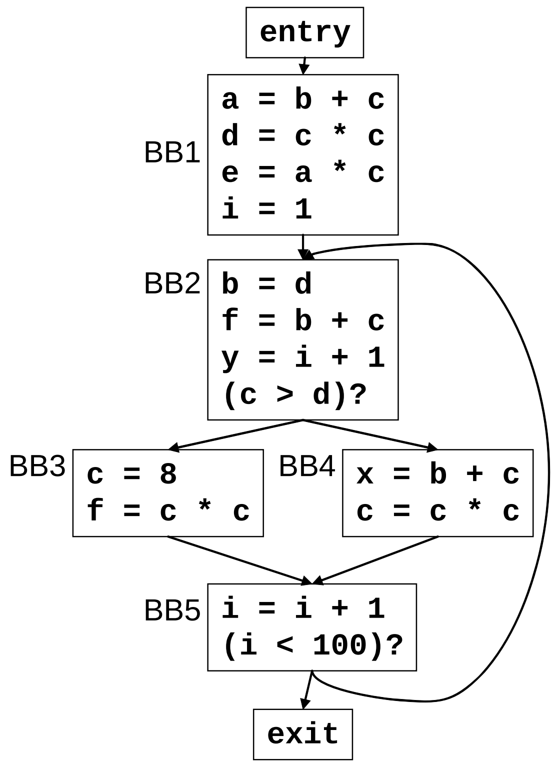

Figure 2: Code for Available Expressions Analysis.

BB	EVAL	KILL
1		
2		
3		
4		
5		

BB	IN	OUT
1		
2		
3		
4		
5		

5.3 New Dataflow Analysis: Use-Without-Def

You have been hired to help develop a software analysis package that will detect bugs and errors in programs. In particular, your job is to design a dataflow analysis pass specifically for finding use-without-def errors (a use of a variable without it being previously defined). Your analysis should be as simple as possible (i.e., it should not gather unnecessary information), and as fast as possible. Your analysis will be plugged into a generic dataflow framework (e.g., New Dragon Book 9.2-9.3).

1. What is the set of elements that your analysis operates on?

2. What is the direction of your analysis?

3. What is your transfer function? Be sure to clearly define any other sets that your transfer function uses (eg., GEN or KILL etc).

4. What is your meet operator? Give the equation that uses the meet operator.

5. To what value do you initialize exit and/or entry?

6. To what values do you initialize the in or out sets?

7. Does the order that your analysis visits basic blocks matter? What order would you implement and why?

8. Will your analysis converge? Why (in words, not a proof)?

9. Clearly describe in pseudo-code an algorithm that uses the result of your analysis to identify use-without-def errors

6 Hand In

Electronic submission:

- The source code for your passes (`FunctionInfo` and `LocalOpts`), the associated `Makefile`s, and a `README` describing how to build and run them. Do this by creating a tar file with the last name of at least one of your group members in the filename, and copying this tar file into the directory

  `/afs/cs.cmu.edu/academic/class/15745-s12/public/asst/asst1/handin`

  Include as comments near the beginning of your source files the identities of all members of your group. Also remember to do a good job of commenting your code.

Hard-copy submission:

1. A report that briefly describes the implementations of both passes.
2. A listing of your source code.
3. Any additional tests that you used for verification of your passes.
4. Answers to the questions in Section 5.