15-745, Spring 2013
Homework Assignment 1

Assigned: Thursday, January 17
Due: Thursday, January 31, 9:00AM

Welcome to the Spring 2013 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 a Linux VM that we will provide.

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-s13/public`

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

1 Install VirtualBox and the 15-745 System Image

To keep you from having to build LLVM yourself and to ensure that all assignments are graded in the same environment, we are distributing a system image for VirtualBox based on Ubuntu 10.04.4 LTS (Lucid Lynx, a Facilities-supported operating system). You must ensure that all of your code works in this image, but you are of course not required to do all of your development with it.

The VirtualBox software is available on several platforms from `http://www.virtualbox.org`. We will use version 4.2.6. You may need to enable your machine's virtualization extensions in your BIOS setup (on some office machines, reboot, press F12 to get the boot menu, choose System Setup, then Virtualization Support, then make sure that the box is checked).

Once you have VirtualBox installed, you can retrieve the virtual machine image from

`/afs/cs.cmu.edu/academic/class/15745-s13/www/vm-images/15-745.ova`

There is a checksum file in the same directory, `15-745.ova.sha1`, that you may use (`sha1sum -c 15-745.ova.sha1`) to verify that the image transferred correctly.

The machine name is `s1315745`; you may log in with username `user` and password `user`. LLVM binaries are in `/home/user/llvm/llvm-3.2.install` (or `LLVM_ROOT`) and source files are in `/home/user/llvm/llvm-3.2.src`. `LLVM_ROOT/bin` is also added to the `PATH`.

We built LLVM 3.2 in the following way:

```
mkdir ~/llvm/llvm-3.2.build ; cd ~/llvm/llvm-3.2.build
../llvm-3.2.src/configure --prefix=/home/user/llvm/llvm-3.2.install \
    --enable-optimized=no --enable-assertions=yes \
    --enable-debug-runtime=yes
make -j 4
make install
```

Moving files between the VM and your host machine is easy using VirtualBox. If you set up a shared directory in the VirtualBox GUI (pointing somewhere on your local filesystem), you can mount it inside the VM with:

```
mkdir /home/user/sharepath
sudo mount -t vboxsf sharename /home/user/sharepath
```

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 (i.e. `export PATH=/path/to/LLVM:$PATH` like the VM is configured). Create a directory 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 prints out "15-745 Function Information Pass". In the next section, you will extend `FunctionInfo.cpp` to print out more interesting information. For now, we will use the dummy LLVM pass to demonstrate how to build and run LLVM passes on programs. First, create a `Makefile` to build the `FunctionInfo` 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
```

```
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, align 4

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

define i32 @loop(i32 %a, i32 %b, i32 %c) nounwind {
entry:
  %cmp2 = icmp slt i32 %a, %b
  %0 = load i32* @g, align 4, !tbaa !0
  br i1 %cmp2, label %for.body.lr.ph, label %for.end

for.body.lr.ph:   ; preds = %entry
  %1 = sub i32 %b, %a
  %2 = mul i32 %1, %c
  %3 = add i32 %0, %2
  store i32 %3, i32* @g, align 4
  br label %for.end

for.end:  ; preds = %for.body.lr.ph, %entry
  %.lcssa = phi i32 [%3, %for.body.lr.ph], [%0, %entry]
  ret i32 %.lcssa
}
```

(b)

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

(Note: you can also copy this code from *ASSTDIR*/`FunctionInfo/Makefile`.) Before moving on, make sure you can `make` this dummy pass. 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 object (`loop.o`) as follows:

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

(`clang` is the LLVM project's frontend for the C language family.)

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. 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 Function Information Pass" should be printed to stderr.

3 Meet The Functions

Program analysis is an important prerequisite to applying correct optimizations: we want to improve code, not break it. 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 (or `*` if variable).
3. Number of direct call sites in the same LLVM module (i.e. locations where this function is called, ignoring function pointers).
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 Table 1. As you can see, the output in Table 1 is not

Name	# Args	# Calls	# Blocks	# Insns
g_incr	1	0	1	4
loop	3	0	3	10

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

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.

You can debug your code with `gdb` as follows:

```
gdb --args opt -load ./FunctionInfo.so -function-info loop.o -o out
(...)
(gdb) b printFunctionInfo
Function "printFunctionInfo" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
(gdb) r
```

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)`

This is a somewhat open-ended question. Please handle at least the above cases, as well as one more in each category that you come up with, for (scalar) integer types.

4.1 Implementation Details

You should create a new LLVM pass in a file named `LocalOpts/LocalOpts.cpp` following the steps in Section 2. Provide an appropriate makefile at `LocalOpts/Makefile`. (Note that it is possible to implement more than one pass in the same directory or file, but we're trying to keep things clean.) `clang` may apply these kinds of local optimizations during the course of regular compilation. To better test your pass, you should build mostly unoptimized LLVM bytecode from the test cases:

```
clang -O0 -emit-llvm -c loop.c
opt -mem2reg loop.o -o loop-m2r.o
```

You may assume that all input to your pass will first go through `mem2reg` as shown above.

We should be able to run your local optimization pass in the following way, modulo the location of the shared library:

```
opt -load ./LocalOpts.so -some-local-opts loop-m2r.o -o out
```

In addition to transforming the bytecode, your pass should also print to standard out a summary of the optimizations it performed. There is no canonical format for this output, but you should at least try to categorize and count the transformations your pass applies:

```
Transformations applied:
  Algebraic identities: 2
  Constant folding: 1
  Strength reduction: 3
```

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
    switch (x) { 0 => L6 | 1 => L4 | 101 => L7 | default => L5 }
L4: print("!")
L5: x = x - 1
    goto L1
L6: return y
L7: goto L7
```

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.

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

Table 2: Available Expressions Analysis.

In the following tables, list the EVAL and KILL sets, then the final IN and OUT sets after AE is performed. You may ignore expressions inside conditional statements (e.g., (z < c)).

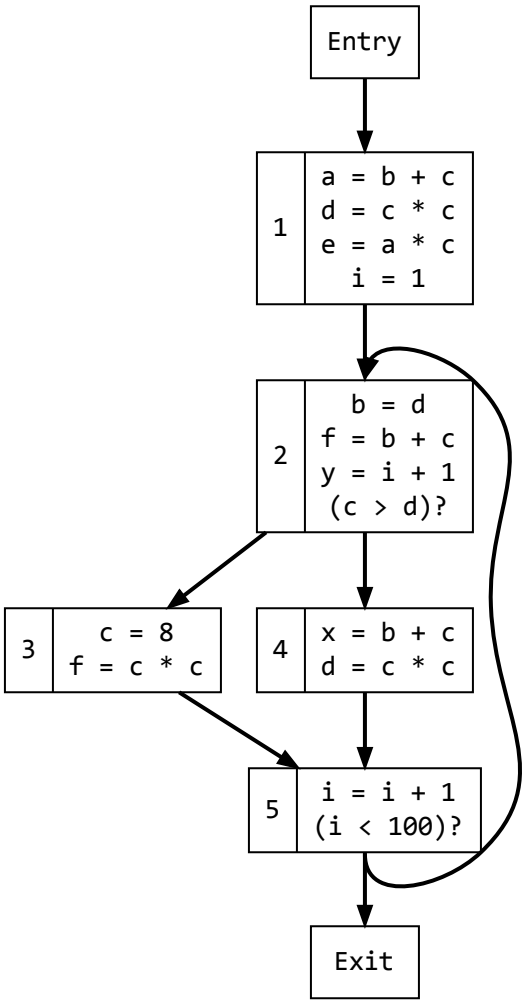

Figure 2: Code for Available Expressions Analysis.

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

Hard-copy submission:

1. A report that briefly describes the implementations of both passes.
2. A listing of your source code. One possible way to generate this is by using `enscript`:

   ```
   enscript -q -DDuplex:true -r -2 -E -fCourier7 --tabsize=2 -p listing.ps
   ps2pdf listing.ps listing.pdf
   ```

3. Listings of additional tests that you used for verification of your passes, as well as their expected results.
4. Answers to the questions in Section 5.

Electronic submission:

- A PDF of your writeup report and answers to the questions, named `writeup.pdf`.
- The source code for your passes (`FunctionInfo` and `LocalOpts`), the associated `Makefile`s, and a `README` describing how to build and run them (especially if you for some reason diverge significantly from what the assignment requires). Place all of these files in a directory with the same name as the Andrew ID of one of your group members. Archive this directory and name the file with the same Andrew ID (`bovik.tar.gz`):

  ```
  tar czvf bovik.tar.gz bovik
  ```

  When the file is extracted with `tar xf`, we expect to see these required files in these locations:

  ```
  ./bovik/README
  ./bovik/FunctionInfo/FunctionInfo.cpp
  ./bovik/FunctionInfo/Makefile
  ./bovik/LocalOpts/LocalOpts.cpp
  ./bovik/LocalOpts/Makefile
  ./bovik/writeup.pdf
  ```

  It is fine if there are other files included; please also include any additional tests you used for verification.
  Copy the tar.gz file into the directory

  ```
  /afs/cs.cmu.edu/academic/class/15745-s13/public/asst/asst1/handin
  ```

  Include as comments near the beginning of your source files the identities of all members of your group. Please don't forget to comment your code.