Basic LLVM Installation for Mac OS X (Mavericks)

1. Prerequisites

As the preferred development environment, Mac is the simplest to configure. The only prerequisites are:

1. Mac OS X 10.9 (Mavericks) or higher
2. Xcode 5 or higher
3. ~5 GB free space (when built, LLVM will take about almost 4 GB) With Xcode 5 installed, several command line tools that are necessary should also be installed. These installation instructions require using the command line. However, once installed you can edit, build, and debug the code directly in Xcode. Test suites will still need to be run from the command line though.

2. Installation of LLVM and CLang

In case it's unclear, in the instructions below $ is not something you should type, but it just represents the command prompt.

1. Open up a terminal and cd to the directory you want to work in. You will be making multiple directories and links, so it's not recommended to do this in ~.

2. Download the LLVM 3.5 and CLang sources from http://llvm.org/releases/download.html

3. Extract the archive:
   ```
   $ tar xJf llvm-3.5.0.src.tar.xz
   ```

4. Rename the directory to llvm:
   ```
   $ mv llvm-3.5.0.src llvm
   ```

5. Enter the llvm directory:
   ```
   $ cd llvm
   ```

6. in the folder 'tools' include and expand the Clang archive
   ```
   $ tar xJf llvm-3.5.0.src.tar.xz
   ```

7. Configure LLVM to build a debug build:
   ```
   $ ./configure --disable-optimized
   ```

8. Build LLVM (this may take 30 minutes to 1 hour depending on the system, but you only have to do it once):
   ```
   $ make
   ```

9. Once you've built LLVM, you need to make a couple of links in the parent directory, so:
   ```
   $ cd ..
   $ ln -s llvm/Debug+Asserts/lib/ lib
   $ ln -s llvm/Debug+Asserts/bin/ bin
   ```

   In some cases the clang binary will be installed in the "Release+Asserts" folder so you will need to change the links above.

   One the 'llvm' directory level you should create your own project folders by creating a subfolder, say, with the name "cs565". The 'cs565-start.zip' constains a series of test with a Makefile for compiling and printing the LLVM assembly instructions. You can now build your own passes based on this project simple template.

3. LLVM opt

LLVM uses a fairly low-level IR that's basically three-address code. The full instruction set reference is here: http://llvm.org/docs/LangRef.html. There are some higher-level parts of the IR. For example, there's a phi instruction that LLVM uses to implement SSA with its virtual registers.

The approach taken by the LLVM developers is to implement most "optimizations" on the low-level IR, corresponding to the "middle" pass in a three-pass compiler. Typically only very platform-specific passes such as instruction selection and register allocation are saved for the back-end. It is conceivable for optimization passes to request further information about the target; for example the autovectorizer does this since there's no reason to autovectorize for a target that doesn't support SIMD instructions.

The LLVM optimizer, `opt`, has quite a few different transformation passes it can run on the IR in an attempt to improve it. If you compile C code with `-O0` (as we will), none of the passes will run, and in `-O3` most (if not all) will run. The list of all the passes that are supported is described in: http://llvm.org/docs/Passes.html.

The framework is flexible so that it's possible to create a dynamic library that contains custom optimizers, and then have `opt` run it on the IR using a command line interface. This is really helpful as `opt` already builds many of the data structures you normally would want for optimization purposes. For example, it builds a call graph, a control flow graph, and a dependency graph between the instructions. This means rather than spending time implementing and debugging these data structures, it's possible to focus on the implementation of a particular optimization algorithm.

So in the `llvm/cs565` directory, there's a tests subdirectory with a handful of simple .c files for test cases. You can just build them all with `make`. For example, for test1.c it'll build:

1. test1.bc = The unoptimized LLVM bytecode that clang emits
2. test1.ll = The disassembled LLVM bytecode, just so you can look at it if you want to (generated by llvm-dis from test1.bc)
3. test1.s = The bytecode compiled into platform-specific assembly (generated by llc from test1.bc)
4. test1 = Executable version of the program (can be generated by any assembler like gcc from test1.s)

You can add more test cases in the Makefile just by adding more .bc files to the FILES definition on line 4.

Once you build the test cases, you can cat the .ll files, or run the executables, etc. But the main idea behind the optimization passes is that they will transform the initial bytecode that's generated by clang into "optimized" bytecode. To verify that this optimized bytecode produces the same behavior, you can always build an executable from it.

4. Run LLVM

LLVM is composed of many separate pieces. To use LLVM to turn a C source file into an x86 (or x86_64) executable, we need to:

1. transform the C source into LLVM bitcode

2. optionally optimize the bitcode
3. transform the bitcode into assembly
4. assemble the program

Just as C files use the extension .c, LLVM bitcode uses the extension .bc, assembly uses the extension .s, and LLVM human-readable assembly uses the extension .ll.

Suppose we have a C program in the file foo.c. Below are the steps needed to create an executable (with no optimization):

```
    clang -emit-llvm -O0 -c foo.c -o foo.bc    // create bitcode .bc
    llc foo.bc                                 // create assembly .s
    gcc foo.s -o foo                           // create executable "foo"
```

To run your program:

```
  foo
```

To turn LLVM bitcode into human-readable LLVM assembly (foo.ll):

```
  llvm-dis -f foo.bc
```

The above LLVM commands (clang, llc, and llvm-dis) are all available in the /unsup directory.

5. LLVM Instructions

Let's get a little more familiar with LLVM's instructions. Consider the following C program, sum.c:

```
#include <stdio.h>

int main() {
  int n;
  int sum;
  sum = 0;
  for (n = 0; n < 10; n++)
    sum = sum + n*n;
  printf("sum: %d\n", sum);
}
```

Running clang and llvm-dis produces the following LLVM assembly code (on my 64-bit machine):

```
; ModuleID = 'sum.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-
S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [9 x i8] c"sum: %d\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %n = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 0, i32* %sum, align 4
  store i32 0, i32* %n, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32* %n, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i32* %sum, align 4
  %2 = load i32* %n, align 4
  %3 = load i32* %n, align 4
  %mul = mul nsw i32 %2, %3
  %add = add nsw i32 %1, %mul
  store i32 %add, i32* %sum, align 4
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %4 = load i32* %n, align 4
  %inc = add nsw i32 %4, 1
  store i32 %inc, i32* %n, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %5 = load i32* %sum, align 4
  %call = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([9 x i8]* @.str,
i32 0, i32 0), i32 %5)
  %6 = load i32* %retval
  ret i32 %6
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-
elim"="true" "no-frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-
nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-
frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-nans-fp-
math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

Some remarks about this assembly code:

- Anything on a line after ; is a comment.
- There are 5 basic blocks named entry, for.cond, for.body, for.inc, and for.end,
- Note that the entry and for.cond blocks, as well as the for.body and for.inc blocks are each really a single basic block (the first block ends with an unconditional branch to the second block) -- I don't know why LLVM has separated them.
- Names that start with a percent sign, (like %0, %1, %n, and %for.cond) are either virtual register names (more about this later) or block labels.

You'll want to become comfortable with LLVM assembly (the human readable form of LLVM bitcode) because the first three projects that you will write for this class will accept LLVM bitcode as input and will emit LLVM bitcode as output.