Lab 11: Compiling for a Virtual Machine

This lab is due at **2359 on Thursday, 6 December**. It should be submitted to the course SI413 as Lab11, with the files named properly according to the lab instructions. See the **submit page** for details. You are **highly encouraged** to submit whatever you have done by the end of this lab time. Remember that **you must thoroughly test your code**; the tests that are visible before the deadline are only there to make sure you are using the right names and input/output formats.

Contents

1. Starter Code

- `splc.hpp`
- `splc.cpp`
- `spl.ypp`
- `spl.lpp`
- `ast.hpp`
- `ast.cpp`
- `frame.hpp`
- `colorout.hpp`
- `Makefile`

You can get all these files at once by downloading the file **lab11.tar.gz** and running `tar xzvf lab11.tar.gz`

2. Introduction

This is a **double-credit lab** that will be the final crowning achievement of your work with the SPL language. It will be difficult, but it should be fun and rewarding too!

For this lab, you are going to implement a compiler for the SPL language. Specifically, you will write a program `splc` that reads in an SPL program and writes out a program in the LLVM Intermediate Representation language. This is a real IR language that is used by many popular modern compilers, and it allows your program to be ultimately compiled to machine code for just about any computer that you can find today. The IR code your `splc`

program generates can either be executed directly by using the **`lli`** command, compiled to assembly using the **`llc`** command, or compiled directly to an executable using **`clang`**.

The lab is structured in three main parts. First is an overview of the LLVM tools you will use and how the LLVM IR language works. There is nothing to turn in for this part, but it will be a useful reference for the rest of the assignment.

The second part of the lab is getting *most* of SPL supported by your compiler, including arithmetic, reading and writing, variables, and function calls. At this point, you will be able to compile many SPL programs to machine code, with the restriction that your functions won't support any non-local references.

The final part of the lab is partially up to you! We have five possible ways of making your compiler support a larger subset of the SPL language. You need to pick (at least) two of these five options and implement them. As you will discover, each option has some tricky aspects that will challenge you to think hard about how your compiler needs to work.

Writing a compiler is a difficult challenge, since it requires you to have three programs in mind: (1) the SPL program your compiler takes as input; (2) your actual compiler which is written in C++; and (3) the LLVM IR code that your compiler is producing. You are ready for this challenge, but it will take careful, deliberate work. Help each other out (within the bounds of the course policy), give yourself plenty of time, ask your instructors for help as needed, and have fun!

3. LLVM

The **LLVM Language Reference Manual** is the most complete and most useful source of information on how the LLVM IR language works. You may want to bookmark that page now, as you should refer to it frequently. Of particular note is the "Instruction Reference" section, which contains details on all of the instructions in the language.

3.1. Tools

The job of your **`splc`** compiler is to convert an SPL program such as **`myprog.spl`** into an LLVM IR program such as **`myprog.ll`**. Note that LLVM IR code typically ends with a **`.ll`** extension.

There are three main tools that you can use to work with LLVM IR code. These should be installed already on the lab machines; on your virtual machine, you may need to install some extra packages by running:

```
roche@ubuntu$ sudo apt install clang llvm-6.0 llvm-6.0-dev llvm-6.0-doc
```

- **clang**: This is a general-purpose C compiler, but it is built on LLVM and has support for LLVM compilation as well. It can be used both to turn a C program into LLVM IR, but also to turn LLVM IR into an executable.

  ```
  roche@ubuntu$ # Produce myprog.ll from C program myprog.c
  roche@ubuntu$ clang myprog.c -O0 -S -emit-llvm


  roche@ubuntu$ # Produce executable myprog from myprog.ll
  roche@ubuntu$ clang myprog.ll -O2 -Wall -o myprog
  ```

- **lli**: An LLVM IR *interpreter* that will run your **`.ll`** program directy. What this does under the hood is really just compile your program and then run it.

  ```
  roche@ubuntu$ # Run myprog.ll
  roche@ubuntu$ lli myprog.ll
  ```

- **llc**: The LLVM *compiler* which transforms your LLVM IR code to actual machine assembly code. This is the first tool used by the clang command above which fully compiles LLVM IR to an executable.

```
roche@ubuntu$ # Compile myprog.ll to x86 assembly myprog.s
roche@ubuntu$ llc myprog.ll
```

3.2. LLVM IR Types

Every register or global variable in LLVM IR code has a type. It's important to emphasize right away that **these types don't necessarily have to match exactly with the types in SPL**. In fact, all of the variables in SPL will be stored as 64-bit integers in your compiled code, whether they actually represent a number, a boolean value, or a lambda. This is because, at *compile-time*, you don't know the type of anything in SPL; therefore all SPL values must be stored in the same way (as 64-bit integers).

The most important types you will need to use in LLVM IR are:

- **i1**: This is a 1-bit integer, i.e., a boolean value. This is the type returned by comparison instructions in LLVM, and it is the type that you must use for the condition of conditional branches.
- **i8**: An 8-bit integer, i.e., a **`char`**. You have to use this type when calling system functions like **`printf`** which expect (arrays of) characters.
- **i32**: A 32-bit integer, i.e., a **`int`**. You will actually *not* use this type very much in your LLVM IR, except as the return value of system calls such as **`printf`**, **`scanf`**, or **`main`**.
- **i64**: A 64-bit integer, i.e., a **`long`**. This is the type you will use in LLVM to store any SPL value, whether it be a boolean, a number, or a function pointer.
- **[*N* x *type*]**: An array of *N* elements of type *type*. The most common you will see is an array of characters such as **`[5 x i8]`**.
- ***type****: A pointer to an object of the given type.
- ***type*(*types*)**: This is how you write the type of a function, specifying the return type as well as the parameter types (a comma-separated list). For example, the type of **`main`** is typically **`i32()`** if you don't care about command-line arguments, or **`i32(i32,i8**)`** if you do.

There are various instructions in LLVM to convert between any of these types. Most useful are **trunc**, to truncate a larger integer type to a smaller one, **zext**, to extend a smaller integer to a larger one by padding with 0 bits, **ptrtoint**, to convert a pointer to an **`i64`**, and (you guessed it) **inttoptr**, to do the opposite.

3.3. Names and instructions

LLVM IR is an SSA (Static Single Assignment) language, so every name can only be assigned exactly once in your program. Global names are preceded with a **`@`** and are typically used for global constants or function names, for example **`@main`**.

Within each function, you will more commonly use local register names, which are preceded with a **`%`**. Because this is an intermediate representation, these names don't really matter, and because the language is SSA, there need to be a lot of them, so it's common to use some numbering scheme like **`%v1`**, **`%v2`**, etc. You will notice that running **`clang`** produces register names tha are just numbers like **`%1`**, **`%2`**, **`%3`**, but you should *not* use these kind of names in your own LLVM IR because they have to follow some special (and annoying) rules.

LLVM IR is a 3AC language, and most typical instructions have this form:

```
%dest = instruction type %arg1, %arg2
```

For example, here is an instruction to subtract 3 from the 64-bit integer stored in register **%x**, storing the result in register **%y**:

```
%y = sub i64 %x, 3
```

You can think of **sub i64** as the entire name of the instruction in the previous case. Some instructions like this are **add**, **sub**, **mul**, **sdiv**, **and**, **or**. The **icmp** instruction is similar, except there is an extra component for the type of comparison. For example, the following instruction compares two 64-bit values in registers **%v1** and **%v2** storing the **i1** value of 1 or 0 in register **%v3**, depending on whether **%v1** is strictly greater than **%v2**:

```
%v3 = icmp gt i64 %v1, %v2
```

3.4. Labels and terminators

Instructions within each LLVM IR function are grouped into *basic blocks* or normal instructions. Each basic block may optionally start with a label such as

```
somelabel:
```

and each basic block *must* end with a special "terminator instruction", probably either **br** or **ret**.

Labels are actually stored in registers; for example the label above would be stored in a register called **%somelabel**.

Branch instructions (**br**) can be either unconditional, like

```
br label %somelabel
```

or conditional, like

```
br i1 %v3, label %iflabel, label %elselabel
```

3.5. Functions

Functions in LLVM IR are defined with their return type, name (which should always be a global name with **@**), and arguments. For example, here is the definition of a function that takes a 64-bit integer n and returns n^2:

```
define i64 @sqr(i64 %n) {
  %nsq = mul i64 %n, %n
  ret i64 %nsq
}
```

To call a function, you use the **call** instruction, like:

```
%twentyfive = call i64(i64) @sqr (i64 %five)
```

Notice that in function calls, we have to specify the type of the function (**i64(i64)** in the previous example), the name of the function, and then the types and names of all the arguments. This can feel a little redundant, but anyway it's a requirement of the language.

3.6. Program structure

Every LLVM program (technically called a *module*) starts out with a specification of the target architecture (and possibly other global attributes):

```
target triple = "x86_64-pc-linux-gnu"
```

Next we should see any global variables. These will most commonly be string literals, for example like:

```
@msg = constant [14 x i8] c"Hello, world!\00"
```

Next we have function prototypes for any external functions or system calls, using the **`declare`** keyword, such as

```
declare i32 @puts(i8*)
```

After this, all that remains is function definitions. One of them should of course be called **`@main`**, like

```
define i32 @main() {
  initial:
    call i32(i8*) @puts(i8* getelementptr([14 x i8], [14 x i8]* @msg, i32 0, i32 0))
    ret i32 0
}
```

Putting all this together gives you a complete, working LLVM IR program.

Note that *the order of function and constant definitions does not matter*, which means that we don't need to have declarations (prototypes) for any locally-defined functions.

4. Starting to fill in your compiler

Now let's get down to writing your SPL compiler! Remember what you're doing here: reading in SPL code and printing out (to the **`resout`** stream) LLVM IR code. You won't actually be *running* the program, but *printing the instructions to run the program later*. Your scanner, parser, and AST generation will be the same, but your **`main()`**, as well as the **`eval`** and **`exec`** methods in the AST, will work differently.

If you inspect the starter code for this lab, you'll notice a few differences from the previos interpreter labs:

- The **`main()`** function has been moved out of **`spl.ypp`** and into its own file in **`splc.cpp`**. So now **`spl.ypp`** is just a parser. In fact, you shouldn't need to modify the parser (**`spl.ypp`**) or scanner (**`spl.lpp`**) for this lab.
- There is no **`value.hpp`** file, and the **`eval`** methods in AST expression nodes return a **`string`**, not a **`Value`** object. This is because the actual values are not known until run-time, so they have no place in the compiler. Instead, each **`eval`** method will return a string representing that value in LLVM IR. This will typically be a register name like **`"%v8`** where the value of that expression has been computed.
- Each **`eval`** and **`exec`** method takes not only a pointer to the current **`Frame`**, but also a pointer to a **`Context`** object. You can see the (very short!) definition of the Context class in **`splc.hpp`**. Right now, it just maintains a counter and implements a simple function to return the next unused register name. This will be frequently useful, since you have to come up with a new name for every instruction in LLVM IR.

If you compile and run the starter code as-is, you will get an **`splc`** program that works only for literal numbers (**`class Num`**), arithmetic operators (**`class ArithOp`**), and write statements (**`class Write`**). Looking at how those **`eval`** and **`exec`** methods are implemented will be useful to get started here.

Exercises

1. Get basic arithmetic, comparison, and boolean operators working, along with both read and write.
   These won't be particularly useful yet, but you may as well also implement expression statements and blocks.
   Here is a program which should successfully compile at this point:

```
write 10;
write 5 + 6;
write 20 + (-8);
write -7 + 4*5;
write true;
write 5 < 1;
write not 1 = 2 and 4 != 5;
write read * 2;
20;
{ write 30; }
{ write 40; { write 50; write 60; } write 70; }
```

5. Variables

Recall from class that variables can pose a challenge with SSA languages like LLVM IR, because each register can only be assigned at one point in the program. One solution to this is to use **phi** instructions, and that is certainly an option here.

But instead, we'll take an easier route: memory. The idea is that, for each declared variable, your program will reserve a space in memory (on the stack) to store the variable's value, and only the *address* will be stored in a register. That way, the variable can change multiple times (via load/store operations) but *the address never changes*, so we don't violate SSA.

You will need to use the instructions **alloca**, **load**, **store** to get this working. Look at the documentation and examples there for help in getting started.

Notice that the AST nodes still make use of **`Frame`** objects, but the **`Frame`** class itself has changed in two very important ways. First, instead of associating a **`Value`** to each name in the symbol table, you associate a **`string`** which will store *the name of the register holding the address of that variable*. For this reason, you no longer need a **`rebind`** method in the Frame class, since the address of a variable never changes.

Rather than try the full test case shown below, think about starting very small and working your way up. For example, you could first just try to get a program to compile that declares a new variable. Then try assigning a variable and printing it out right away. Starting slowly like this and working carefully, examining the code your compiler produces for very simple cases first, will make this whole lab much easier.

Exercises

2. Get variable declaration with **`new`**, assignment, and lookup working. Here's a program which should work now:

```
new x := 10;
write x + 3;
new y := x * x;
write y;
write y + x;
{ new x := 20;
  write x; # should be 20 of course
  x := -3;
```

```
  write x; # now it's changed
}
write x; # should be back to 10 here
y := 1101;
write x+y;
```

6. Control

In order to write any really interesting programs in SPL, you need support for conditionals and loops. Getting those working will be reminiscent of the assembly programming you did in IC210, and will involve careful use of labels and jumps.

For example, a code fragment such as

```
ifelse 3 < 5
  { write 100; }
  { write 200; }
write 300;
```

might be compiled to

```
%v1 = icmp slt i64 3, 5
    br i1 %v1, label %b1, label %b2
  b1:
    call void(i64) @write (i64 100)
    br label %b3
  b2:
    call void(i64) @write (i64 200)
    br label %b3
  b3:
    call void(i64) @write (i64 300)
```

Exercises

3. Get if/else and while statements working. After this, a program like the following should compile and run successfully:

```
new secret := 42;
new guess := read;
while guess != secret {
  ifelse guess < secret
  { write -1; }
  { write 1; }
  guess := read;
}
write secret;
```

7. Function calls

The last part of your SPL compiler which everyone is required to write is the implementation of function calls. To make things easier, for this part your compiler may insist that the bodies of functions never contain any non-local references. That means that the communication to/from the function consists entirely of the argument and return values, and it makes the implementation much simpler, albeit more restricted in the SPL programs you can compile.

Getting functions to work really consists of two tasks. First, you have to implement function *definitions*, as triggered by **`lambda`** expressions in the code. Each lambda in the SPL code will correspond to one function in the LLVM IR code that your compiler produces. But at the point of the **`lambda`** itself, your compiler is in the middle of emitting code for **`main()`**, so it's not time to print out the function definition yet. Instead, your compiler should just add that **`Lambda*`** node to some list of functions, whose definitions will be output later, after **`main()`** is over.

At the point of the **`lambda`** itself, all your compiler needs to do is convert the function pointer to an **`i64`** type in LLVM, using the **ptrtoint-to** instruction. For example, the very simple (and mostly pointless) SPL program

```
lambda x { write 123; };
write 456;
```

might compile to something like

```
; ... header definitions etc up here ...
define i32 @main() {
    %v1 = ptrtoint i64(i64)* @fun1 to i64
    call void(i64) @write (i64 123)
    ret i32 0
}
define i64 @fun1 (i64 %arg) {
    call void(i64) @write (i64 456)
    ret i64 0
}
```

(Note, this program is NOT correctly handling arguments and return values yet. You have to figure that out next!)

Once you have **`lambda`**s working well, emitting code for function names like we see above, the next step is to implement function calls. Just as with your interpreter labs, I recommend starting by ignoring arguments and return values and just get the control to go to the function and come back. Just as the example above uses **`ptrtoint`** to convert the function pointer to a saved **`i64`** value, at the function call site you will have to convert an **`i64`** back to a function pointer using **inttoptr-to**.

Exercises

4. Get function definitions and function calls working *without local references*. After this, you can compile some neat programs like this:

   ```
   new f := false;
   ifelse read = 2
     { f := lambda x { ret := x*x; }; }
     { f := lambda x { ret := x*x*x; }; }
   new i := 1;
   ```

```
while i <= 10 {
  write f@i;
  i := i + 1;
}
```

8. Choose your own adventure

There are 5 more exercises in this section. **For full credit on the lab, you must complete at least two of them**. Of course, you are encouraged to try and implement even more than two!

Exercises

5. Implement run-time type information in your compiler. This means that you will store - at run time - an extra value alongside every actual SPL value, to hold the type of that value.
   It is up to you how to represent the types in your LLVM IR program. I recommend using an **`i8`** value with something like 0 indicating unset, 1 indicating a number, 2 indicating a boolean, and 3 indicating a function.
   The most visible change in your program is that the **`write`** command in SPL should now respect the type of its argument, printing the value of a number, or "true"/"false" for a boolean, or "function" for a function pointer.
   In addition, you should do run-time type checking everywhere your program makes use of a value. For example, an SPL program like `write 1 * false;` should still compile, but produce a run-time error from the failed coversion of **`false`** to a number.
   (You do *not* need to implement the type checking builtins like **`isnum`**.)

6. Implement full lexical scope with closures, allowing non-local references in function calls. This challenging task will allow all kinds of more sophisticated SPL programs to compile, such as:

   ```
   new showx := lambda x {
     ret := lambda ignored { write x; };
   };
   new fiver := showx@5;
   new sixer := showx@6;
   fiver@0;
   sixer@0;
   fiver@0;
   fiver@0;
   ```

   To get this working, you will need to first of all allocate storage for your variables on the *heap* rather than the stack. Because there is LLVM instruction to do heap allocation, you will have to do this with system calls to **`malloc`**.
   Then, you'll need to store a *closure* for each function declared, rather than just a pointer to the function itself. This closure needs to contain this function pointer, but also the addresses of all of the variables in scope at the point of the function definition. Then when the function begins, you will need to unwrap this closure and load those addresses back into registers.
   This will be a good challenge, but worthwhile! Ask your instructor if you run into trouble and need some help.

7. Implement the debug statement in SPL, which is any string enclosed in double-quote characters. That string should print out when the compiled program is executed, at that point in the program where the debug statement appears. This is already part of the SPL scanner and parser you have, so all you need to do is figure out how to get **`Debug::exec`** method to work.
   After this, a program like the following should compile:

```
"Please enter a positive number."
new x := read;
ifelse x > 0
  { "Thank you and good job." }
  { "You will now be punished with 50 punches."
    new i := 0;
    while i < 50 { "punch" i := i + 1; }
  }
```

8. Implement built-in functions in SPL like you did for **Lab 9**. You should have (at least) built-in functions for **`sqrt`**, **`rand`**, and one other function of your choosing.
   The trick here will be to write the code for the built-in functions in LLVM IR and emit those function definitions every time your compiler runs. Inside your **`main()`** you'll probably have to have instructions to store those function pointers just like any other variable would be stored, so that function calls to your built-in functions work just like any other function call.
   (You do *not* need to implement the type checking builtins like **`isnum`**.)
9. Implement short-circuit evaluation of **`and`** and **`or`** expressions, using **phi** expressions in LLVM.
   Remember that short-circuiting means the second part of a boolean operation (**`and`** or **`or`**) is only evaluated if it's necessary. For example, if the first part of an **`and`** is false, then we don't need to bother with evaluating the seecond part, because we know already that the entire expression is definitely false.
   This would not be too difficult to do with a simple if/else construct, except that after the blocks come back together, you need to have just a *single register* that stores the result of the boolean operation. Because of the rules of SSA, you'll find that this isn't straightforward to do. To solve it, you will need to use a **phi** function in the first basic block after the short-circuited boolean expression completes. Look at the documentation there for guidance and ask your instructor if you need any help!