Assignment 5
Closures and Bytecode Interpreters

15-411: Compiler Design
Rob Arnold (`rdarnold@andrew`) and Eugene Marinelli (`emarinel@andrew`)

Due: Thursday, November 6, 2008 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Thursday, November 6. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1

[30 points]

Assume we have function pointers added to L4. Their typing rules follow that of SML. For example:

```
int add(x : int, y : int) { ... }
int main() {
  var addp : (int,int) -> int = add;
  var mainp : () -> int = main;
  // Recursion forever!
  return mainp();
}
```

This means that functions are included in the local scope of every function for the purposes of typechecking, and that the name spaces for functions and variables are unified. The only valid values for a function pointer is the address of an existing function (as seen above) or a closure (see below).

Closures are a language feature often seen in high level languages such as Python, Javascript, and SML. They can be 'faked' in lower level languages like C, C++ and Java by using a class or struct to hold any state.

We would like to add support for closures to L4. In this context, a closure is a function declared within another function which may use local variables. A closure takes a list of arguments and returns a single output just like regular functions. A closure can be declared and used within a function or passed as an argument to another function just like a function pointer. For example, the following $f(n)$ returns $n + 7$:

```
int f(n:int) {
  var x : (int, int) -> int;
  x = lambda(y : int, z : int) { return y + z + n; };
  return x(2,5);
}
```

A closure should share variables with its parent function; changes made in either will affect the other. Similarly, if two closures share the same parent function, then they will also share variables. Example:

```
int f(n : int) {
  var add2 : () -> int,
      add3 : () -> int;
  add2 = lambda() : int { n += 1; return n + 2; };
  add3 = lambda() : int { return n + 3; };
  return n + add2() + add3();
}
```

In this example, $f(n)$ would return $3n + 7$. Note that you can use this to mimick objects with hidden state (like private member variables in C++/Java/C#):

```
struct MyClass {
  var x : int;
  var dtor : () -> ();
  var gety : () -> int;
  var sety : int -> ();
};

MyClass *newMyClass(x : int, y : int) {
  var this : MyClass*;
  this = new (MyClass);
  this->x = x;
  this->dtor = lambda() { this->x = 0; y = 0; };
  this->gety = lambda() { return y; }
  this->sety = lambda(newy : int) { y = newy; }
  return this;
}
```

Here x is a public variable and y is private. With a little additional syntactic sugar, this could be a start to adding OOP concepts to L4.

1. Describe the typechecking that needs to be added to ensure type safety.

2. Describe how you would implement closures in L4. Remember that allocations in L4 are garbage collected. Be sure to include changes to both the compiler and runtime as needed.

Problem 2

[30 points]

For some applications, compiling to machine code may not be desired, for example, for portability or security reasons. In such a case, a quick compiler to an intermediate form with a fast interpreter is needed. Traditional optimizations may be too expensive so there is a heavy focus on having a fast interpreter (since the compiler is usually not self-hosting). Like CPUs, at the core of the interpreter is the instruction dispatcher which decodes and executes instructions. The interpreter loop is often a hot performance spot and critical to the overall speed of execution.

The dispatch loops are often written in C for readability and maintainability, but how they translate to assembly is significant. Consider the following loop:

```
Operation *ip;
...
for(;;) {
  switch(*ip) {
    case OP_ADD:
      ...
      break;
    case OP_MUL:
      ...
      break;
    case OP_CALL:
      ...
      break;
    case OP_JUMP:
      ... set ip to new location ...
      continue;
    ... more cases ...
  }
  ip++;
}
```

a. Describe how the control flow of this code translates to assembly. You may assume that the opcodes are defined such that a jump table is possible.

b. How could the control flow graph be optimized at a high level (i.e., within the bounds of mostly-portable C)? Hint: think of the optimizations covered in class and how they might help you simplify the flow control inside the loop. Your solution should reduce the number of jump targets by 1.

c. How might the computed goto extension to C eliminate another jump?

d. As with hardware instructions, virtual machine instruction sets vary in size. For instance, Lua uses 38 opcodes but Microsoft's CLR and SpiderMonkey (Javascript) have more than 200. Since interpreted languages tend to be very dynamic, there are a lot of behaviors and features that need to represented in the bytecode. Give some pros and cons for large and small instruction counts and how they effect overall performance (memory and speed) of the interpreter. Keep in mind that unlike with hardware instructions, you need to also be concerned with the implementation of the VM which is running on real hardware.

e. BONUS: The compiler generates an additional jump that is not required when writing the interpeter loop in assembly. Why? What other optimizations can you do with handwritten assembly?