15-411: Compiler Design — Fall 2017

Recitation 4: Lab 2, From AST to Assembly — 29 September

In this recitation, we will consider the following `l2` program:

```
1 // compute the log base 10 of a number
2 int main() {
3   int input = 500; // the "input"
4   int output;
5   if (input <= 1) return 0;
6   for(output = 1; input > 0; output++) {
7     input /= 10;
8   }
9   return output;
10 }
```

Parsing vs. Elaboration

A number of design decisions in Lab 2 center around the roles and responsibilities of the parsing and elaboration steps. In general, we recommend that you keep your parser as simple as possible; this means that the syntax tree generated by your parser should reflect the actual **syntactic** structure of L2 as closely as possible. Once you've parsed the program and you have a parse tree, you can then perform an elaboration pass to generate an AST that reflects the **semantic** structure of L2.

For example, the Lab 2 handout recommends that you use a case of the form declare(x, τ, s) to represent the scope of a variable x, but you don't need to use this exact form as part of your parser. Instead, you can parse declarations as regular statements, and then capture their scope during elaboration.

Of course, you're welcome to design your compiler to divide the responsibilities differently (or even not have an elaboration pass at all). However, the structure we're recommending has worked well for students in the past. And, as with most things in this course, you will likely need to revisit most of your choices for later labs.

Checkpoint 0

Given the following parse tree representing the program above, perform an elaboration that captures variable scopes, transforms the `for` loop into a `while` loop, and replaces increment and arithmetic assignment operators with simpler constructs.

```
Seq(Init(input, int, 500),
  Seq(Declare(output, int),
    Seq(If(Compare(input, 1, LEQ),
        Return(0),
        Nop()
      ),
      Seq(For(
          Assign(output, 1),
          Compare(input, 0, GT),
          Incr(output),
          Block(
            AssignOp(input, 10, DIV)
          )
        ),
        Return(output)
      )
    )
  )
)
```

Checkpoint 1

Annotate all the expressions in your elaborated AST with their types.

Checkpoint 2

Fill in the missing instructions in our IR tree. Then, identify all of the basic blocks.

```
_c0_main:
  input <- 500
  if _______________ then goto _____ else goto _____
.L0:
  return(0)
  goto L2
.L1
  goto L2
.L2
  output <- 1
.L3
  if _______________ then goto _____ else goto _____
.L4
  input <- ___________________
  output <- ___________________
  goto _____
.L5
  return(output)
```

Code Generation

Since we have this nice and simple IR tree to work with, code generation is mostly straightforward. We will focus on `if` statements, as the rest is largely the same as `l1`.

Let's examine `cogen` for `if`.

s	cogen(s)	proviso
`if`(e, s_1, s_2)	`cogen`(t_1, e)	t_1, l_t, l_f, l_e new
	`cmp` $t_1, \$0$	
	`jmpe` l_t	
	`jmp` l_f	
	$.l_t$:	
	`cogen`(s_1)	
	`goto` l_e	
	$.l_f$:	
	`cogen`(s_2)	
	`goto` l_e	
	$.l_e$:	

Not only do we now create new temps, but we also need to create new labels. You will likely want a type similar to temps for labels in your implementation.

Checkpoint 3

Perform code generation from the IR tree.