15-411: Compiler Design — Spring 2023

Recitation 2: Lab 2, From AST to Assembly — 3 February

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 AST generated by your parser should reflect the source-code **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 explicitly represent 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.

```
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 }
```

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 more primitive 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

Fill in the missing instructions in our IR tree.

```
_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)
```

Liveness Analysis (With Loops!)

For Lab 1 you have already implemented liveness analysis with straight-line code. For Lab 2, you will need to extend this implementation to work with branching and conditionals.

We reviewed liveness with branching last week, so we'll go over liveness with loops today!

For reference, here are the rules again for live-in and live-out sets:

$$\text{LiveIn}(\ell) = \text{Uses}(\ell) \cup (\text{LiveOut}(\ell) - \text{Defs}(\ell))$$

$$\text{LiveOut}(\ell) = \bigcup_{s \in \text{succ}(\ell)} \text{LiveIn}(s)$$

When there are cycles in the control flow graph, i.e. loops in the program, these rules can become self-referential. However, these rules will always have a unique *fixpoint*. By this we mean there is a unique solution to the live-in and live-out sets such that the (self-referential) rules hold.

To compute this fixpoint, we start by assuming all live-in and live-out sets are empty, and then repeatedly apply the rules above until the live sets stop changing. As before, you can work bottom up in the program.

Checkpoint 2

Analyze the program from Checkpoint 1 to determine the live-out and live-in sets at each of the lines. Then draw the interference graph.