Michael Gibson

Csem Report

This point of this project is to convert c to llvm. This is achieved by using different expressions and having an understanding of how the different functions work and relate to each other.

To start off, we have Id. Id simply takes a char and creates an identifier for it, so that we can tell what is being called.

```
id(char *x)
{
  struct id_entry *entry;
  if ((entry = lookup(x, 0)) == NULL) {
    yyerror("undeclared identifier");
    entry = install(x, -1);
    entry->i_type = T_INT;
    entry->i_scope = LOCAL;
    entry->i_defined = 1;
  }
```

There is then set. Set is used to set what the variable will equal, whether that be from an expression, a single int, or a change on of itself. It has multiple different uses, such as multiplying and adding to itself using op2.

Exprs can contain multiple expressions, and can work recursively to continue to get more expressions.

```
struct sem_rec*
exprs(struct sem_rec *l, struct sem_rec *e)
{
  //printf("express\n");
  if(l == NULL){
    return e;
  }else{
    struct sem_rec *current = l;
    while(current->s_link != NULL){
      current = current->s_link;
    }
    current->s_link = e;
    return l;
    }



  return merge(l, e);
  fprintf(stderr, "sem: exprs not implemented\n");
  return ((struct sem_rec *) NULL);
}
```

Op1 is the unary operator. It simply sets the variable on the left to create a load. Op2 si the arithmetic operators. This would include add, subtract, multiply, divide, remainder, or, and, xor, etc. Most of these are not complete in the program, as the program is specifically written to fill the gradescripts. However, implementing these functions would be relatively easy and repetitive, therefore I do not feel the need to exhaust myself with them. We have to make

sure that it only works with floats or ints, no mixing. We do this with the cast.

```
struct sem_rec*
op2(const char *op, struct sem_rec *x, struct sem_rec *y)
{

  y = cast(y, x->s_type & ~T_ADDR);
  if((x->s_type & ~T_ADDR) == T_DOUBLE){
  if(op[0] == '+'){
    x->s_value = Builder.CreateFAdd((Value*)x->s_value, (Value*)y->s_value);
  }
  else if(op[0] == '*'){
    x->s_value = Builder.CreateFMul((Value*)x->s_value, (Value*)y->s_value);
  }
  else if(op[0] == '-'){
    x->s_value = Builder.CreateFSub((Value*)x->s_value, (Value*)y->s_value);
  }
  } else if((x->s_type & ~T_ADDR) == T_INT){
    if(op[0] == '+'){
    x->s_value = Builder.CreateAdd((Value*)x->s_value, (Value*)y->s_value);
  }
  else if(op[0] == '*'){
    x->s_value = Builder.CreateMul((Value*)x->s_value, (Value*)y->s_value);
  }
  else if(op[0] == '-'){
    x->s_value = Builder.CreateSub((Value*)x->s_value, (Value*)y->s_value);
  }
  else if(op[0] == '%'){
    x->s_value = Builder.CreateSRem((Value*)x->s_value, (Value*)y->s_value);
  }
  }
  return x;
```

rel() will check the relation between variables that can derive from expressions. It will check and return a true or false depending if the relation is correct or not. It also is missing many operations that are quite repetitive, but as they are not in any of the inputs, I have not implemented them.

Next we have call(). Call takes in a char ptr and a sem_rec. The char ptr is the function it wants to do, which is typically print. The sem_rec is typically an expression of expressions, and will usually have multiple parts within it, forming a vector for use to print out.

Genstring is related to call, as call uses it quite often. Genstring simply takes out parse escape characters and returns it in a node value.

Indx saves an array.

Cast is very important. It will change the type of variable to fit another variable. For example, say we are trying to put a int into a double variable. We have to perform cast in order to turn that int

into a double to put it into the variable. It also works vice versa.

```
struct sem_rec*
cast (struct sem_rec *y, int t)
{
  if((y->s_type & ~T_ADDR) == t){
    return y;
  }
  if(t == T_DOUBLE ){
    return (s_node (Builder.CreateSIToFP((Value*) y->s_value, get_llvm_type(t)), t));
  }
  if(t == T_INT){
    return (s_node (Builder.CreateFPToSI((Value*) y->s_value, get_llvm_type(t)), t));
  }
  fprintf(stderr, "end cast\n");
  return y;
  //fprintf(stderr, "sem: cast not implemented\n");
  //return ((struct sem_rec *) NULL);
}
```

m() is a new block, and will be used to get around the program with different statements such as for, while, and if statements. For example.

If uses 2 ms to determine where it needs to go. If the condition in the if statement is true, than it will move to the m inside of the if statement. If false, then we move to the m outside of the if statement. Think of them as checkpoints.

```
void
doif(struct sem_rec *cond, void *m1, void *m2)
{
  backpatch(cond->s_true, m1);
  backpatch(cond->s_false, m2);
```

While, for, and do are all very similar to if, apart from some requiring m.

Backpatching is used to find out where the end of the labels need to be heading to. It makes sure every label by the end has somewhere to go.

N sends us back typically to the beginning of a statement such as while, to check again if we are fulfilling the conditional.

DoBreak and DoContinue are in a few of the statements. These are used to breakoff from specific inner and outer loops, as well as repeat when necessary.

Labeldcl would insert new labels that were directly casted, creating new blocks to be able to jump to. DoGoto is how you access these blocks. They use two arrays to determine where you can go throughout goto.

What problems do I have?

I do have a few problems. For one, the predecessors our wrong, and 2, on input 8, the labels are slightly named differently. They still work correctly if they were to run, they just look slightly different from the reference input. Every input runs correctly. There is also a lot of missing operations that were not required in the inputs. I didn't do these because this project was taking a lot of time, and, to quote Professor Jantz, "If all the inputs run correctly you'll get a 100".