Step 7: Choose Your Own Adventure

Due: December 4th, 5 pm EST (10 pm UTC) [Free extension until December 7th]

Setting up your repository

Set up your GitHub classroom repository for step 7 using this link

Then clone your repository as directed in the cloning and submitting instructions

Background

Up until now, we have worked on adding different features to our compiler. This has involved making changes either to the front end (adding new syntax), the intermediate representation (adding AST Nodes or three address code) or to the backend (adding new code generation routines, or new optimization passes like register allocation), or some combination of all three.

In this step, we are asking you to go through that exercise for a problem of your own choosing. We have laid out a couple of options of final steps that you can do, each of which will have you work through the same process of figuring out what parts of your compiler to modify to add a given feature.

There is no starter code for this assignment. You should start either with your Step 6 code (if you are selecting a project pertaining to adding pointer-based features to the language) or your Step 5 code (if you are selecting a project pertaining to performing code optimization of 3AC).

We encourage you to start working on this project ASAP, and chat with us over Piazza or office hours to get some guidance on how to implement the various pieces of the project. You should also refer to the steps we took to add new features to our language in previous project steps as a roadmap.

Here are your project choices:

Option 1: Recursive data structures

Without the ability to add new named types to the language (like C `struct`s), we cannot easily define recurisve data structures like linked lists, where one field of a structure is a pointer to another structure of the same type. However, we *can* get the effect of recursive data structures by borrowing a couple of tricks from C and Lisp: *void pointers*, *casts*, and *cons pairs*.

Void Pointers

A void pointer is a pointer to a memory location that does not specify what the type of that memory location is (remember the return type of `malloc` in C). This means that you can create a generic "address" variable by calling it a void pointer: your compiler knows that it is dealing with an address, but does not know what the type of the data at that address is.

To derference a void pointer, you first tell the compiler what the void pointer actually points to, using a *cast* expression:

```
int * x;
void * y;
y = malloc(...)
x = (int *) y;
```

Casts

Casts are expressions with the following syntax:

```
cast_expr : '(' type ')' expr
```

Which is an expression that takes a piece of code (`expr`) and tells the compiler to treat its result as if it were of type `type`.

Casts in languages like C sometimes require code to be generated (e.g., casting an `int` to a `float`), but for our purposes, we will only be casting between pointers. The purpose of the cast is to give the compiler the information it needs to generate code correctly (and for everything to typecheck).

So you can allocate a `void` pointer, cast it to an `int` to assign it, then print it:

```
void * x;
int * y;

x = malloc(4);
y = (int *) x;

*y = 7;

print (* y);
```

Cons Pairs

Data structures in Lisp are built out of a simple building block called a *cons pair*. Think of a cons pair as a two-element array. The first element points to a piece of data, and the second element either points to `null` (`0`) or to another cons pair. So cons pairs are the building blocks of linked lists.

Cons pairs in Lisp don't require any fancy type information because Lisp doesn't have types. But we can make something like cons pairs work in uC by using void pointers. We will create our own cons pairs by allocating a 2-element array of `void` pointers to be our cons pair. The first element can then point to whatever data you want (an integer, a float, or even another cons pair), while the second element can point to another array of `void` pointers.

The following C code (which is *almost* valid uC, or will be, once you add support for void pointers and casts 😉 builds a two element linked list using this `void` pointer trick:

```
int main() {

    void ** cell1; /* a cons pair */
    void ** cell2; /* a cons pair */
    void ** curr;

    int * d; /* used to access first cells -- a pointer to an int*/

    cell1 = malloc(8); /* allocate a cons pair: two element void * array
*/
    cell1[0] = malloc(4); /* allocate an int for the data */

    cell2 = malloc(8); /* allocate a cons pair: two eleemnt void * array
*/
    cell2[0] = malloc(4); /* allocate an int for the data */

    d = (int *) cell1[0];
    * d = 8;

    cell1[1] = (void *) cell2; /* cell1's next pointer points to cell2 */

    d = (int *) cell2[0];
    * d = 10;

    cell2[1] = 0; /* cell2's next pointer points to null */

    /* loop through linked list */
    curr = cell1;
    while (curr != 0) {
        d = (int *) curr[0];
        printf("%d\n", * d); <-- this isn't valid uC code, obviously.

        curr = (void **) curr[1];
    }
}
```

TODO

This option builds on Step 6.

If you choose this project option, you need to do the following:

1. Add support for `void` pointers to the language (you'll need to alter `Scope.Type`)
2. Add support for casts to the language, which involves:
    1. Modifying `MicroC.g4` to add support for a new expression variant.
    2. Adding a new AST Node for cast exressions
    3. Adding a new code generation step for casts (you can assume we'll only cast pointers, so no need to do any type conversions.)

You should then be able to write linked list code in your language, as you see above.

Option 2: Type conversions

Up until now, we have not supported any kind of type conversion in our language. There is no way to convert ints to floats, or vice versa. In this project option, you will add it.

Explicit Conversion

The set of possible type conversions is vast. In this step, we will focus on both explicit and implicit conversions. An explicit conversion uses a cast operation:

```
float f;

f = (float) 1 /* converts 1 to a floating point 1.0, then assigns */
```

while an implicit conversion uses type promotion rules to automatically convert numbers:

```
int x;
float f;

x = 2;
f = 1.0;

print(x + f); /* implicitly casts x to a float, then adds */
```

There are three implicit conversion rules in uC:

1. In a binary expression, if one operand is an `int` and the other is a `float`, the `int` is converted to a `float`. The result of the expression is a `float`.
2. In an assignment, if the LHS is an `int` and the RHS is a `float`, the RHS is cast to an `int`.
3. In an assignment, if the LHS is a `float` and the RHS is an `int`, the RHS is cast to a `float`.

TODO

This option builds on Step 6.

To implement this option, you need to do the following:

1. Add support for cast expressions in uC (as in option 1). Unlike in option 1, you have to worry about casting between ints and floats, which requires generating code. We have added the custom instructions `FMOVI.S` and `IMOVF.S` to RiscSim to facilitate this. You will need to add those `Instruction`s to your compiler (look at `Instruction.java` to see how this can be done).
2. Modify your code generation for assignments and binary expressions to implement implicit conversion.

Option 3: Global Liveness and Dead Code Elimination

Using the principles of dataflow analysis, you should perform *global* liveness analysis on your code prior to register allocation. This means that you will no longer need to make conservative assumptions about what local variables are live at the end of basic blocks, because your liveness analysis will have told you this information.

TODO

This option builds on Step 5. (Note that this means your liveness analysis does not need to handle pointers.)

To implement this option, you need to do the following:

1. Add a liveness analysis pass to your program. This should run *before* register allocation on a per-function basis, and use dataflow aanlysis to compute liveness across basic blocks. At each return statement, assume that global variables are live. (Hint: build a `Map` that uses `Instruction`s as keys and live sets as values)
2. Perform dead code elimination using this liveness information. Remove any instructions that have destination operands that are dead. Note that after DCE, you will need to re-do liveness, and then re-do DCE, so you will need to add a check to see if DCE has converged.
3. Use this global liveness information during register allocation. (Hint: pass in the `Map` you generate in Step 1.)
4. Add an option to your `runme`: if we pass "`--dfa`" to your `runme`, it should turn global liveness analysis and DCE *on*. If we do not pass that option to your `runme`, it should use your Step 5 code. (Note that turning on this optimization *should* result in your code taking fewer cycles to complete).

What you need to do

Pick one of the three options above to implement in this step. Modify your `runme` to print "OPTION X" before starting compilation (where "X" is `1`, `2`, or `3`) so we know which option you chose.

Running your code

To run your code, you can use the Risc simulator, which you can clone from https://github.com/milindkulkarni/RiscSim.

You can run an assembly file by running:

```
> python3 RiscSim/driver.py [assembly file]
```

Sample inputs and outputs

We will post some sample inputs and outputs for Options 1 and 2 over the course of the project period. Stay tuned on Piazza.

(The inputs for Option 3 will be the same as the inputs for Step 5.)

What you need to submit

- All of the necessary code for building your compiler.
- A Makefile with the following targets:
    1. `compiler`: this target will build your compiler
    2. `clean`: this target will remove any intermediate files that were created to build the compiler

- A shell script (this *must* be written in bash, which is located at `/bin/bash` on the ecegrid machines) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the compiler and second, the filename where you want to put the compiler's output. You can assume that we will have run `make clean; make compiler` before running this script, and that we will invoke the script from the root directory of your compiler. Note that your `runme` should print out which option you have chosen, as described above, and, if you chose Option 3, should take the `--dfa` option, as well.

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation.

See the submission instructions document for instructions on how to submit. You should tag your step 7 submission as `submission`