IIT CS440: Programming Languages and Translators

Homework 6: IR Generation and Optimization

Prof. Stefan Muller
TA: Xincheng Yang

Out: Tuesday, Apr. 20
Due: Saturday, May 1 11:59pm CDT

Updated Apr. 30

This assignment contains 5 written tasks and 4 programming tasks, for a total of 62 points, in addition to a maximum of 3 bonus points for the winners of the competition described in Section 2.1.

0 Logistics and Submission - Important

The same rules as on HW2-5 apply. In particular:

1. Make sure you read and understand the updated/clarified collaboration policy on the course website.
2. The complicated skeleton code of this assignment will make testing in the `ocaml` toplevel or on Try-OCaml very difficult. Instructions for testing your code are provided in the writeups of the programming problems.
3. Your answers to the programming problems will go in the files `compile.ml` and `optimize.ml`. You only need to submit these files and your written pdf. Do not rename the .ml files.

1 C−−

The source language of our compiler in this assignment will be C−−, the small subset of C we've been using in class[1]. As a reminder, the grammar for C−− is below:

aop	→	`+` | `-` | `*` | `/`
relop	→	`<` | `<=` | `>` | `>=` | `==` | `!=`
iexpr	→	*num* | *var* | *iexpr aop iexpr*
bexpr	→	*iexpr relop iexpr* | *bexpr* `&&` *bexpr* | *bexpr* `||` *bexpr*
stmt	→	*var* `=` *iexpr*`;` | `if` *bexpr stmt* `else` *stmt* | `if` *bexpr stmt* | `while` *bexpr stmt* | `return` *iexpr*`;` | {*stmts*}
stmts	→	ϵ | *stmt stmts*

C−− is valid C, except that there are no functions, and in particular, there's no `main` function, which you'd need in order to make a valid C program. However, we'll allow C−− programs to have unbound variables. If you were to take a C−− program and make it a function of a C program, these variables might be the arguments to the function, so their values would be specified by whoever was calling this function. Or, if this was the `main` function, these unbound variables might be the command-line arguments to the

[1]C−− is also the name of one of the intermediate representations used in the OCaml native code compiler. It's not exactly the same language though.

program. Indeed, when we're actually running C−− programs, you'll specify the values of these variables on the command line.

So, as an example, the following C−− code computes the factorial of `n` for any `n`, which must be specified on the command line.

```
i = n - 1;
while (i > 1)
{
  n = n * i;
  i = i - 1;
}
return n;
```

If this were saved as `examples/fact.cmm`[2], once you finish the assignment, you'd compile it with
`./cmm examples/fact.cmm`
This would output TAC code as `examples/fact.tac`. You could then run this using
`./tac examples/fact.tac n=5`
which will output the correct answer, 120. You can also run the same compiled code with different arguments:

```
$ ./tac examples/fact.tac n=4
24
$ ./tac examples/fact.tac n=3
6
$ ./tac examples/fact.tac n=2
2
```

There are other examples of C−− programs in the `examples` folder.

You'll be compiling C−− to Three-Address Code (TAC), following the algorithms we outlined in class. As a reminder, here's the grammar for TAC:

$$\begin{array}{lcl} instr & \rightarrow & label\texttt{:} \mid var\ \texttt{=}\ addr\ aop\ addr \mid var\ \texttt{=}\ addr \mid \texttt{jump}\ label \mid \texttt{jz}\ label\ addr \mid \texttt{jneg}\ label\ addr \mid \texttt{ret}\ addr \\ addr & \rightarrow & num \mid var \\ code & \rightarrow & \epsilon \mid instr\texttt{\textbackslash n}code \end{array}$$

Note that, unlike many other languages we've worked with, some whitespace is actually meaningful in TAC: the lexer will enforce that only one instruction is given per line, hence the `\n` (newline) in the grammar (this includes labels: labels must be on their own lines). Otherwise, you can include whitespace, including additional newlines, anywhere. As an example, here's TAC code for the factorial example above:

```
  t0 = n - 1
  i = t0
l0:
  t3 = 1 - i
  jneg l1 t3
  jump l2
l1:
  t1 = n * i
  n = t1
  t2 = i - 1
  i = t2
  jump l0
l2:
  ret n
```

[2]In fact, it is!

We've included this as `examples/factcomp.tac` as well.

As part of the code handout, we gave you the code for the TAC interpreter `tac`. Even before writing any code, you can compile this and try it out:

```
$ make tac
$ ./tac examples/factcomp.tac n=1
1
$ ./tac examples/factcomp.tac n=5
120
```

The type definitions for C−− and TAC are given in `tac.ml`. This file also defines three functions that will be helpful in building your compiler: `new_var ()` generates a new temporary variable, `new_addr ()` generates a new temporary *address* (all variables are addresses, but these are separate types in the code: a variable `x` can be turned into an address `AVar x`, and, in fact, `new_addr` is implmented as just `AVar (new_var ())`). Finally, `new_label ()` generates a new label. Most of the rest of the code we gave you handles the lexing and parsing of C−− and TAC, as well as the TAC interpreter. You'll be writing your compiler in `compile.ml`.

Task 1.1 (Programming, 5 points).

Implement the function `compile_intexpr : intexpr -> code * addr` which compiles an *iexpr* and returns the code (a list of TAC instructions) to compute the value, as well as the address containing the answer. In class, we had the expression translation return a variable rather than an address, but note from `tac.ml` that any variable can be turned into an address.

Task 1.2 (Programming, 20 points).

Implement the remaining cases of the function `compile_boolexpr : boolexpr -> label -> label -> code`. As we saw in class, if `t` and `f` are labels, `compile_boolexpr be t f` should generate TAC code that jumps to `t` if `be` is true and jumps to `f` if it's false.

Updated 4/30: We didn't talk about the `Not` case in lecture, and it's not implemented in the parser, so you can't test it by writing .cmm files (you can still use asserts). So the Not case of the match is now a 1-point bonus task: you can get the full 20 points for Task 1.2 by implementing all the other cases correctly, and one extra point by doing the Not case (if you don't do it, please leave the `raise ImplementMe` so it still compiles).

Task 1.3 (Programming, 20 points).

Implement the remaining cases of the function `compile_stmt : stmt -> code`, which compiles a *stmt* to produce the equivalent TAC code.

1.1 Testing the Compiler

When you've finished this section, you can compile the C−− compiler by running

```
make
```

and then compile an example program by running, for example,

```
./cmm <file>.cmm
```

which will compile the program into `<file>.tac`. You can then run it using `./tac` as in the introduction. If the input `.cmm` file had any unbound variables, compiling it with `./cmm` won't generate errors, but you need to specify the values of these variables when running it with `./tac`, which takes as arguments the TAC file, and then a list of arguments in the form `<var>=<val>`. For example, if the input program had unbound variables `x` and `y`, e.g.

```
return x + y;
```

we'd have to run the code using

```
./tac example.tac x=5 y=10
```

2 Optimization

We discussed in class a few ways to improve the TAC code generated by the compiler. Here, you'll try some out.

Note: In this assignment, we are not using Static Single Assignment (SSA) form. Be careful about what optimizations are valid.

Task 2.1 (Written, 5 points).

Consider the following TAC code.

```
1     x = 5
2     y = 0
3   lbegin:
4     jneg lret y
5     y = x - 1
6     x = y
7     jump lbegin
8   lret:
9     return x
```

Can the copy on line 1 (x = 5) be propagated? If not, explain why in 2-3 sentences. If so, give the code with this propagation performed.

Task 2.2 (Written, 5 points).

Consider the following TAC code.

```
1     x = 1
2     a = 0
3     jump l1
4   l0:
5     z = y + 2
6     jump l3
7   l1:
8     y = x + 1
9     jump l0
10  l2:
11    a = x + 3
12    jump l4
13  l3:
14    b = y + a
15    ret b
```

Which lines represent *dead code* (i.e., lines that are never reached or contain assignments that are never used)?

In the next task, you'll implement some optimization on the code generated by your compiler. For the purposes of this task, we'll define an optimization as something that does not change the result of the produced TAC code and results in either fewer lines of TAC or fewer instructions executed while interpreting the TAC code on most inputs (for example, moving a line of code out of a loop will not make the code any shorter but, as long as the loop is run at least twice, will result in fewer total instructions executed at run time). Note that the file `optimize.ml` contains a function `optimize: code -> code`, which currently just returns its argument. This function is called by `./cmm` on the code returned by your `compile_stmt` function after compilation.

Task 2.3 (Programming, 5 points).

Implement some optimization. As a reminder, some optimizations we discussed in class are:

- Removing unneccessary jumps
- Dead code elimination
- Constant folding
- Propagating copies of variables that are only written once.

You can implement one of these or get creative (there are more optimizations described in Section 8.5 and Chapter 9 of PDB, though some of these require advanced analyses we didn't learn about).

Remember, we're not using SSA format, so you need to be a little careful about doing optimizations[3].
Task 2.4 (Written, 2 points).

Briefly describe the optimization you implemented in the previous task.

2.1 Extra Credit: So You Think You Can Optimize

If you're done with the assignment and really want to try your hand at optimizing code, we thought we'd encourage this with a friendly competition. In the examples folder, theres a file called `opt.cmm`. What does it do? That's for us to know and you to figure out if you want. But it takes one argument, `arg`. When we compile it with our solution (with no optimizations), the TAC that results is 52 instructions, not counting labels, and executes 105 instructions when `arg`=10.

Let's see how much better you can do.

Make your best effort at building optimizations into your compiler, then submit the code generated with your compiler using this form:

`https://forms.gle/pb527MPyZNxdqHxM7`

Submissions will appear on the leaderboard:

`http://cs.iit.edu/~cs440/leader.html`

(you can submit using your real name or not, but whatever name you give will show up on the leaderboard). You can submit multiple times as you keep improving your optimizations: only your best submission will be displayed. Submissions are ranked by the number of instructions executed when `arg`=10. (The number of non-label instructions is shown too, but this doesn't count for the ranking.) As extra incentive, the students in first, second and third place on the leaderboard at 11:59pm Monday, May 3 will get 3, 2 and 1 bonus points respectively. (If you take 0 or 1 late days on this assignment, you can continue to work on your optimizations and submit to the leaderboard without counting against your late days, just don't resubmit on Blackboard).

Think you've got the hang of optimizing TAC code but don't have the time or OCaml skills to build the optimizations into the compiler? There's also a "hand-optimized" division! Submit your best hand-optimized version of `opt.tac` using the same form (just say "No" when it asks if the code was generated by your compiler). For reference, our best hand-optimized version of the code has 8 non-label instructions and executes 25 instructions when `arg`=10. We will be extra impressed if you can beat that! (Note that you're unlikely to get the program that optimized by just applying standard optimizations to the generated code; you'd probably have to figure out what the code does and write an optimized version of it from scratch.) The rules are the same as for the compiler-generated division, including the prizes. To spread the wealth, you can only win a prize in one division.

Notes:

1. Keep display names respectful, please.
2. The leaderboard is very low-tech and updates infrequently. Be patient and check the "last updated" text at the bottom.

[3] If you want to implement a pass that turns the code into SSA, does fancier optimizations on it, and then converts the code back into valid TAC, we won't stop you, but this question is only worth 5 points.

3. If the leaderboard was updated since your last submission, and you don't see your submission, it's probably because either a) it didn't parse as valid TAC (we use the same parser as `./tac`, so if it works for you on your computer it should be fine) or b) your code doesn't produce the same answer as the original program on some test inputs.

Good luck!

3 Standard Written Questions

Task 3.1 (Written, 0 points).

How long (approximately, in hours/minutes of actual working time) did you spend on this homework, total? Your honest feedback will help us with future homeworks.

Task 3.2 (Written, 0 points).

Who, if anyone, did you collaborate with (and in what way), and what outside sources, if any, did you consult in working on this homework?