NTNU – Trondheim
Norwegian University of
Science and Technology

Problem set 6

Wrapping up

- The skeleton this time can already compile programs that don't do anything fancy
- What it's missing:
  - Local variables
  - Function calls
  - Conditionals
  - Loops
  - Continue

Timing, once more

- This is such a direct continuation of PS5 that it may be simpler to just complete your own design rather than adapt to mine
  - Feel free to do that
- If PS5 is an uphill, it may be simpler to just submit however far you get, and pick up from where this week's skeleton starts
  - Feel free to do that as well
- The overlap in time is not by design, it is an unfortunate accident
  - Please, choose the path of least resistance
  - I just want everyone to have the chance to touch all the parts of the generator
  - If you can't find the time to complete things yourself, jump ahead
  - *That's what the skeleton codes are for in the first place*
- The goal is that you get to grips with how it works, I can see that from PS6
  - It is not so important for me to accumulate a shining collection of half-complete generators
  - Think of PS5 as a status update, it will not be judged harshly
  - It was originally meant to introduce code generation in stages, so that nobody gets stuck with the whole nine yards on the very last evening.

Local variables

- Unlike global variables, the names of these are not the mechanism by which we access them
  - They go on the run-time stack
- Their sequence number can be used to find their offset from the base pointer
- Open a function by making space for them on the stack
  - They were counted as we generated the symbol table
  - (mind the 16-byte alignment)
- Otherwise, these can go into expressions in the same manner as global variables

Function calls

- These also appear in expressions
- Generating them is a matter of following the same calling convention by which we receive them (PS5)
- That is,
  - Put the 6 first args in their designated registers (names in a constant array in the skeleton code)
  - Spill any additional args. on the stack
  - Call the function
  - Restore the stack, find the result in %rax (as with any subexpression)

Conditionals (IF & relations)

- The relation can be computed in the same manner as arithmetic expressions:
  - Recursively generate code to evaluate the left expression, leaving result in %rax
  - Put the result away on stack
  - Recursively generate code to evaluate the right expression
  - Get former result from stack
  - Compare, and jump as appropriate to the comparison operator

Where to jump?

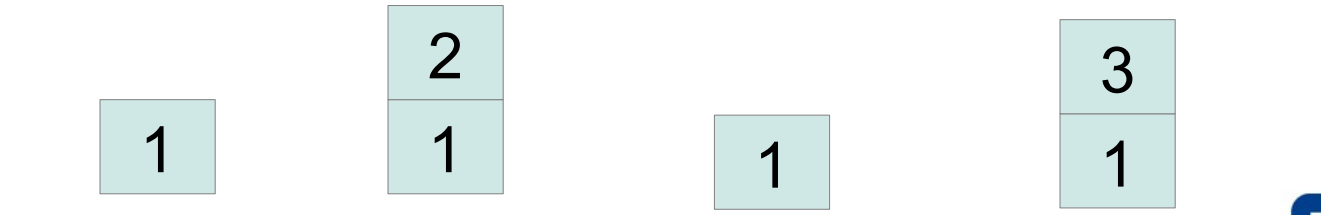
if (a = b) then A else B

can be turned into

```
evaluate a
evaluate b
compare
jump-not-equal ELSE
A
jump ENDIF
ELSE:
B
ENDIF:
(rest of program...)
```

Those labels need a numbering scheme

- This can just be a counter suffix on the labels; ELSE1, ENDIF1, ELSE2, ENDIF2, *etc.*
- Mind that conditional statements can *nest*, though:
  - We'll need another stack to push/pop the counter values on
  - This way, the top of the stack tracks the innermost IF we're inside:
    if_1 (a) { if_2 (b) then c; if_3 (d) then e; }
    results in



Loops

- Loops are a lot like conditionals:
  while (c) { A }
  becomes

```
WHILELOOP:
evaluate c
jump-false ENDWHILE
A
jump WHILELOOP
ENDWHILE:
(rest of program...)
```

Those labels need a numbering scheme too

- It's a nested construct, same story as for IF-statements
- It's not a bad idea to give them their own counter and stack, though, because...

Continue

- Continue-statements skip directly to the condition-evaluation of the enclosing while loop
- With a shared counting scheme, the enclosing construct might be an IF
- With separate stacks for ifs and whiles, the index of the enclosing while loop is on top of the while-stack
  - You can surely hatch a different counting scheme if you so desire, this is just my suggestion

When all that is done...

...the VSL compiler is finished!

Give yourself a pat on the back, and some kind of reward. A compiler is a tough thing to make.

FAQ

- Why did we split the work in so many little parts?
  - Our compiler gets some of its structure from the progress of the course. It could be designed much better.
- Why didn't we do this in C++, like modern people?
  - For the sake of transparency to the machine level. Even if you never see VSL again (good thing, too), you can make an educated guess at how a C compiler works at this point, and that is valuable.
- Why didn't we do this in $LANGUAGE, like $LANGUAGE_ADVOCATE people?
  - Because you can now pick your favorite language/toolkit and do that over the summer.
  - Don't go forth and write compilers from scratch in C, we did it to get a feeling for precisely what better tools do on your behalf
- Can I design FooLang12000 and be famous?
  - Probably not – most languages which survive do so because they add something to solve a real-world problem. Language design for the sport of it can be Great Fun[1], but they very rarely change the world.

[1] http://esolangs.org/wiki/Language_list