CS 42—Stored-program computers, part 2

Thursday, September 27, 2018

Summary

Today, we complete our focus on stored-program computers, by discussing how function calls might be implemented on such machines. Our discussion will take the form of a thought experiment: assuming we can't change the architecture of the machine (i.e., the kinds of memory it has—registers and RAM—and the basics of the fetch / execute cycle), how can we extend the Hmmm programming language so that we can write programs that have functions?

Terminology

Instruction set: the commands supported by Hmmm (e.g., `read r1`)

Syntax: the characters we type when writing a program (e.g., `read r1`)

Semantics: what the syntax means (e.g., get a number from the user and store it in register `r1`)

Argument: when calling a function, an argument is a value that we pass to the function

Parameter: part of a function definition that describes an argument it can take (e.g., a name for the argument value, the range of values it's allowed to take, etc.)

Caller: a piece of code that calls a function

Callee: a function that is called

Stack: a part of memory that we treat in a special way—each value we read from the stack must be the most recently non-read value that we've written. In other words, the last value we put in the stack is the first value we take out of it.

LIFO: stands for "**L**ast **I**n **F**irst **O**ut", the order we write and read things from the stack.

Recall: Hmmm so far

Our Harvey Mudd Miniature Machine (Hmmm) has 16 registers and 256 locations of random-access memory (RAM). Hmmm programs are stored in RAM, starting at location 0. Additionally, Hmmm has a program counter (a register whose value is the *location* of the next line of code to execute) and an instruction register (whose value is the currently executing line of code).

Instruction	Description
halt	stop!
read rX	Place user input in register rX (input can be from -32768 to +32767)
write rX	Print contents of register rX
nop	Do nothing
setn rX N	Set register rX equal to the integer N (-128 to +127)
addn rX N	Add integer N (-128 to 127) to register rX
copy rX rY	Set rX = rY
add rX rY rZ	Set rX = rY + rZ
sub rX rY rZ	Set rX = rY - rZ
neg rX rY	Set rX = -rY
mul rX rY rZ	Set rX = rY * rZ
div rX rY rZ	Set rX = rY / rZ (integer division; no remainder)
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)
jumpn N	Set program counter to line N
jeqzn rX N	If rX == 0, then jump to line N
jnezn rX N	If rX != 0, then jump to line N
jgtzn rX N	If rX > 0, then jump to line N
jltzn rX N	If rX < 0, then jump to line N

Hmmm conventions

Write lots of comments / documentation. Clearer is better than shorter (but shorter *can* be clearer).

`r0` always contains the value 0. All other registers should be written before they're read.

`r13` is reserved for the **return value** (i.e., *what* is the result?)

`r14` is reserved for the **return location** (i.e., *where* should the program go next?)

`r15` is reserved for the **stack pointer** (the location where the next thing will be pushed)

New instructions for function calls

Instruction	Description
jumpr rX	Set program counter to the line number whose value is in rX
calln rX N	Copy the next address into rX, then jump to line N
pushr rX rY	Store value of rX into memory location rY; increment rY
popr rX rY	Decrement rY; load contents of memory location rY into rX;

Function-call discipline

If your program has functions, initialize the stack pointer first. (Remember that the code and the stack share the RAM).

When writing a function, declare which registers should serve as parameters and describe any assumptions you make about their values.

Inside a function:

- assume the argument values are in the registers you declared
- assume every register (except `r0`) is available for writing
- `jumpr r14` means "return"

When calling a function:

1. Save—by pushing onto the stack—all register values that you'll need when the call returns. Remember that you might need `r13` and `r14`.
2. Prepare the arguments by writing to the appropriate registers.
3. Call the function with `calln r14` *N* (where *N* is the first line of the function).
4. Restore—by popping from the stack—all the register values you need to finish the function. Pop the values in the reverse order you pushed them.

Next time: Functional programming in Racket