MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Lecture Notes – Nov. 25, 1997

MACHINE ARCHITECTURE

We have seen how to implement versions and variations of SCHEME in SCHEME. But how does this map down to a real machine. If we can implement the code for just the meta-circular evaluator, which was carefully written to not require higher order procedures, then we will have on our machine a version of SCHEME which does support higher order procedures.

To get all the way down to actual machines we need to understand a number of issues:

1. The general sort of architecture of modern computers, at a level which we will implement in hardware in the term project in `6.004`.
2. How to write programs in machine code, or assembler language.
3. How to implement the abstractions for `cons`, `car`, and `cdr` on a general machine, since these are not usually supported at the hardware level (whereas integer `+`, `-`, `<`, etc., usually are).
4. How to translate by hand the code for the meta-circular evaluator into machine code[1].

1. Computer Architecture

We will consider a fictional but reasonably generic machine in order to understand what a modern computer looks like. We will not examine all the details of the architecture of machines but look at just enough to understand how the full power of a high level language can be implemented.

The central element of a computer includes a number of registers and one or more arithmetic units. The registers consist of some small fixed number of general registers (perhaps 16 or 32 of them); let's call them `r0`, `r1`, etc. There are usually some additional special registers which we will discuss later.

The general registers are some fixed width, or number of bits. For the last several years most machines have used 32 bits for a register. That is a general register can store a 32 bit number, which if treated as just positive integers means they lie in the range 0 to $4,294,967,295$. By treating them as two's complement numbers they can be considered to lie in the range $-2,147,483,648$ to $2,147,483,647$.

The arithmetic units can do such things as add two numbers from two registers, subtract one from another, multiply them, etc., and put the result in a register as a result.

[1]It is fairly easy to write a *compiler* for code that does not have higher order procedures (e.g., standard `C` code), and chapter 5 of the SICP text shows how to write a compiler for non-HOP scheme (it uses the evaluator itself to handle the HOP parts). We will not cover compilers explicitly in the course this term, but it is not very hard going if you want to read section 5.5 yourself. BTW it is much harder to write a compiler which produces really good machine level code.

Instructions

There is a clock ticking (the speed of the clock makes it a 200MHz machine or whatever) and at some regular[2] number of clock ticks an *instruction is executed.* An instruction might be something like:

```
add     r0, r1
```

which says that the contents of registers `r0` and `r1` should be added and the result placed in register `r0`. For our imaginary machine we will have instructions that look like:

```
op      dest, source
```

where the result ends up in the `dest` register. We will allow the source to be another register, or perhaps a constant, such as:

```
sub     r1, #3
```

which says to subtract the constant 3 from register `r1`.

A typical machine will have a number of other simple "arithmetic" instructions, such as `logand` which bitwise takes the logical conjunction of the two registers, `logior` and `logxor` for inclusive and exclusive logical or, etc. Machines also usually have shift instructions which shift registers a bit to the left, or a bit to the right, etc.

Sequential execution

Instructions are executed sequentially. They are stored in memory, or RAM. Memory is organized as a series of 8 bit bytes, numbered 0 through some large number[3]. In our machine we will assume that each consecutive 4 bytes of RAM represent a single instruction. We will discuss the encoding of the instructions as 32 bit quantities a little later.

A special register, `pc` or *program counter*, keeps the address of the current instruction to be executed. At the beginning of every instruction it is incremented by 4, to point to the next instruction. Thus we can have a sequence of instructions such as:

```
add     r0, #7
sub     r1, r0
add     r2, r0
```

which will get executed one after the other.

The `jump` instruction unconditionally adds some positive or negative offset to the `pc` after letting it be incremented by 4 at the beginning of an instruction. In our *assembler* language description of our machine code we use symbolic names for instructions, but these must be resolved into bits

[2] In fact in many architectures like the x86, it is not a regular number of ticks but depends on the instruction being executed; more modern architectures try to execute instructions on a regular schedule, ultimately once every clock tick—making this happen requires many instructions to be in partial parallel execution.

[3] For a 32 bit machine with byte addressing the maximum size of memory is 4 gigabytes. Until recently it was preposterous to think of having such a large physical RAM, and so only virtual RAMs (really stored on disk but made to appear to be in RAM) could be that large. Now it is getting plausible to have 4 gigabytes of RAM, adding another push to 64 bit machines rather than 32 bit machines.

before they are loaded into the actual machine—a symbolic program called an assembler does this work for us.

```
        add     r0, #7
        jump    place
        sub     r1, r0
place:  add     r2, r0
```

In the above code register `r1` is not affected as the subtraction from it is jumped over by the `jump` instruction.

Now we introduce conditional jumps. The `cmp` instruction subtracts the source from the destination but does not change the destination then stores in a small register, the `cc` or *condition code* register, whether the result was negative, zero, or positive. A subsequent conditional jump instruction only jumps if certain conditions are met. E.g.,

```
        cmp     r0, r1
        jump.ge bigger
        sub     r0, #1
bigger: add     r2, r0
```

skips the subtraction of 1 from `r0` if the contents of `r0` were greater than or equal to the contents of `r1`. The condition codes we allow are `eq`, `ne`, `ge`, `gt`, `le`, `lt`, with the obvious interpretations.

We will need one other instruction, the `load` instruction, which again has a destination register, and whose source is simply another register or a constant—there is no arithmetic done on it on the way to the destination. E.g.,

```
        load    r0, #27
        load    r1, r0
```

puts 27 in both registers `r0` and `r1`.

Representation of instructions

Our instructions need to be represented as 32 bit quantities. Usually there is some fixed size subfield of bits specifying the operation (e.g., `add`, `load`, `jump.lt`, etc.).

For arithmetic operations (including `cmp`, and `load`) another field represents the destination register (if we had 32 general purpose registers then this would need to be a 5 bit subfield). Lastly there will be a field representing the source. It would dedicate one bit to saying whether the rest of the field was a constant or not. If not then the rest of the field would specify a register.

For `jump` instructions the part of the instruction that was not the opcode would be a signed displacement to be added to the `pc`.

As an example, suppose we use an 8 bit opcode field (giving a total of 256 possible operations), then we might have the following code representations:

```
                                oooooooodddddsssssssssssssssssss
        cmp     r0, r1          01000000000000000000000000000001
        jump.ge bigger          10000011000000000000000000001000
        sub     r0, #1          01000010000001000000000000000001
        load    r1, #27         01100000000011000000000000011011
bigger: add     r2, r5          01000001000100000000000000000101
                                oooooooodddddsssssssssssssssssss
```

This is just meant to be illustrative. We will not consider the numerical representations of instructions any further.

Stacks

Another important aspect of machines is the stack. There is often a special register, the *stack pointer* or `sp`, which contains an address in memory, often a multiple of four, along with instructions `push` and `pop`, which both mention a single register, which is either copied from or to, respectively, from or to the stack.

```
push    register
```

increments `sp` by four, then stores the contents of the register at that location in memory.

```
pop     register
```

Gets the data in the location pointed to by `sp`, and then decrements `sp` by four[4].

Subroutines

There are special instructions that use the stack that make it easier for our machine to implement prodecure calls:

```
addit:  add     r0, #3
        sub     r0, r1
        rts
start:  load    r0, #27
        load    r1, #19
        jsr     addit
        load    r1, #13
        jsr     addit
end:
```

In the above code, if we start at the label `start`, then by the time we get to label `end`, the contents of register `r0` will be 1.

The `jsr` instruciton places the address of the next instruction on the stack and then jumps to the label instruction. The `rts` instruction pops an address off the stack and jumps to there.

2. Writing Machine Code

Now that we know what machine code looks like we can translate small scheme programs to machine code by hand.

```
(define (max a b)
  (if (> a b)
      a
      b))
```

[4] On different machines the stack grows and shrinks in different directions. There is also a question of whether the first byte pointed to by a multiple of four address is the high order byte of a thirty two bit quantity or the low order byte. Different machines use different conventions.

Could be translated to:

```
start:  cmp     r0, r1
        jump.gt abig
        load    r2, r1
        jump    end
abig:   load    r2, r0
end:    rts
```

where we adopt the convention that the arguments `a`, and `b`, start off in registers `r0`, and `r1`, and we want the result to be in register `r2`. We could make the code a little shorter (and never less efficient, but sometimes more efficient) by rewriting it as:

```
start:  load    r2, r0
        cmp     r0, r1
        jump.gt end
        load    r2, r1
end:    rts
```

If we are to translate bigger scheme programs by hand, let us make a convention amongst ourselves. Let's suppose that the arguments will always be in registers `r0`, `r1`, etc., and that the result should be in `r0`, and that the stack height should be just the same in entry as on exit.

```
(define (triang n)
  (if (= n 0)
      0
      (+ n (triang (- n 1)))))
```

Let's assume that we call this by `jsr`ing to the label `triang`.

```
triang: cmp     r0, #0
        jump.eq ret0
        push    r0         ;need to save for deferred add
        sub     r0, #1
        jsr     triang
        pop     r1
        add     r0, r1
        jump    end
ret0:   load    r0, #0
end:    rts
```

We have optimized movement amongst registers by popping the saved value for `n`, which was in register `r0`, back to another register, `r1`.

Notice that this computation uses $2n$ stack locations.

Now consider an iterative version in scheme—one that uses only tail recursive calls to other procedures:

```
(define (triang n)
  (define (help n sum)
    (if (= n 0)
        sum
        (triang (- n 1) (+ sum n))))
  (help n 0))
```

We make a helper subroutine and assume the code starts by `jsr`ing to the label `triang`.

```
help:   cmp     r0, #0
        jump.eq retsum
        push    r0
        push    r1
        add     r1, r0
        sub     r0, #1
        jsr     help
        load    r2, r0      ;save the result
        pop     r1          ;restore the stack
        pop     r0
        load    r0, r2      ;get the result back to the right location
        rts
retsum: load    r0, r1
         rts
triang: load    r1, #0
push    r0
        push    r1
        jsr     help
        pop     r1
        pop     r0
        rts
```

We soon notice that saving and storing the arguments around the procedure calls is not necessary as they are not used again. This gives us:

```
help:   cmp     r0, #0
        jump.eq retsum
        add     r1, r0
        sub     r0, #1
        jsr     help
        rts
retsum: load    r0, r1
        rts
triang: load    r1, #0
        jsr     help
        rts
```

Now we see that there is nothing to do after the procedure calls, but instead just return to the top location on the stack. If we do a jump instead of a `jsr`, the callees `rts` will do the work for us!

```
help:   cmp     r0, #0
        jump.eq retsum
        add     r1, r0
        sub     r0, #1
        jump    help
retsum: load    r0, r1
        rts
triang: load    r1, #0
        jump    help
```

Not stack is used at all! Constant stack size!

If we re-order just a little we can get rid of another `jump`.

```
triang: load    r1, #0
help:   cmp     r0, #0
        jump.eq retsum
        add     r1, r0
        sub     r0, #1
        jump    help
retsum: load r0, r1
        rts
```

So now we see how tail recursion works. It is just a transfer of control, and lets the thing that is called handle getting the result into the right place, and getting back to the right place.

3. Implementing Pairs and Pointers

Scheme variables can have different types as their values, and we can't tell what they will be necessarily by inspecting the program. It depends what values are passed as arguments, and may vary between different calls.

We will encode Scheme values as 32 bit quantities, but their type will have to be encoded in them. Let us assume that we expect not to have more than 256 megabytes of memory (or 2^{28} bytes). Then we could use the four most significant bits of a 32 bit quantity to specify the type of a Scheme value, and the lower 28 bits to either be the value directly or to be the address in memory of the components that comprise the value[5].

A pair will need to have two 32 bit values as its components, the `car` and `cdr`. Thus a pair 32 bit value will have four bits of type tag, and 28 bits of address into memory where in two successive 4 byte locations will be found the `car` and `cdr`.

Strings will have an address which is a where a series of bytes making up the string are stored. Perhaps there will be a length field there also, or perhaps they will be null terminated.

Symbols will have an address where their components are stored—the only component we need consider is a symbol's print name, which itself will be a string (i.e., a 32 bit quantity with a string tag, pointing off to the string contents).

Numbers will have a four bit tag and then 28 bits of actual numerical value, so they will range from $-134,217,728$ to $134,217,727$.

The emptylist will have its own tag, and the other 28 bits can be zeroes.

Likewise booleans will have special tags, and the 28 bits of data will encode whether they are true or false.

Suppose we choose the following tags, as the most significant four bits for Scheme values:

[5] In a real implementation we can be much more clever than this on a byte addressed machine and encode the type bits in the lower two and half bits of a 32 bit quantity, take no performance hit in dreferencing pointers, and use some of the hardware word alignment testing to do type error checking for us—the Sparc architecture supports a handful of special instructions just so that Lisp can be implemented efficiently on it using these tricks.

```
symbol      0
string      1
pair        2
number      3       ;not a real address
emptylist   4       ;not a real address
boolean     5       ;not a real address
procedure   6       ;not needed in our mceval, but a good
                    ;idea for real
```

If symbols are additionally stored at low addresses then our current machine code will let us put their addresses as constants in the `cmp` instruction's available 18 bits. In a real implementation we might want to do something more general.

What we do need to do is provide ways of getting at the tags, and implementing `cons`, `car`, and `cdr`.

If there is a Scheme value in a register, then the tag is the four most significant bits. We could simply shift the contents 28 bits to the right to get ahold of them. Most machines have such an instruction, but to make our code below easier to read we will write it as:

```
gettag  dest, source
```

which will take a value in a `source` register, not disturb it, but place the four most significant bits in the low order four bits of the `dest` register, padded on the left with zeroes.

If we assume a pair is stored as two consecutive four byte quantities then indirect addressing, i.e., using the lower 28 bits of a register as an address to access in memory, possibly with a small offset, will implement `car`, and `cdr`. On a real machine we might need to zero out the four tag bits, but let us assume that:

```
load    dest, 0(reg)   ;works as car
load    dest, 4(reg)   ;works as cdr
```

but to make our code easier to read we will write it as:

```
getcar  dest, reg
getcdr  dest, reg
```

Finally we must deal with `cons`. That will not be a primitive instruction on any realistic machine, and must incorporate garbage collection. We will examine this topic on Decemeber 2nd. So let us assume that there is a procedure, with a beginning label of `cons` that expects arguments in registers `arg1`, and `arg2`, and which returns a new pair (tagged correctly) in the `val` register.

4. Writing the MC Evaluator in Machine Code

We are now in a position to translate our meta-circular evaulator, by hand into machine code.

We will give certain registers symbolic names and use those names in our code rather than `r0`, `r1`, etc. The registers we need, along with their usual contents are:

```
env     the current environment
exp     the expression currently being evaluated
unev    a list of things not yet evaluated
args    a list of evaluated arguments
proc    a procedure to apply
val     the place where all values are returned
tag     temporary register for putting tags
arg1    first argument to cons
arg2    second argument to cons
```

We will write each procedure as a label with the name of the procedure. We assume the following convention:

1. To call a procedure which we are implementing by hand simply do a `jsr` to its label.
2. Each procedure expects its arguments in certain registers.
3. Each procedure returns its result in the `val` register.
4. Each procedure returns to the place that was on top of the stack when it was called.
5. Each procedure is free to destroy all registers—it is the caller's responsibility to save any register it needs.

We will only implement a few of the procedures of the meta-circular evaluator; the others are all equally easy, or difficult, to work through. For the procedures we do implement we will use the following conventions for where the arguments are supplied:

```
mceval   exp env
mcapply  proc args
evlist   unev env
lookup   exp env
cons     arg1 arg2
```

Now we are in a position to implement the meta-circular evaluator. We can translate it piece by piece:

```
(define mc-eval
  (lambda (exp env)
    (cond ((number? exp) exp)                     ;base-case
          ((symbol? exp) (lookup exp env))        ;base case
          ((eq? (car exp) 'quote) (car (cdr exp)));special forms
          ((eq? (car exp) 'cond) (evcond (cdr exp) env))
          ((eq? (car exp) 'begin) (evseq (cdr exp) env))
          ((eq? (car exp) 'lambda) (list 'proc (cdr exp) env))
          ((eq? (car exp) 'define) (evdefine (cdr exp) env))
          (else (mc-apply (mc-eval (car exp) env)
                          (evlist (cdr exp) env))))))
```

```
mceval: gettag  tag, exp
        cmp     tag, #3
        jump.ne notnum
        load    val, exp      ;numbers self evaluate
        rts
notnum: cmp     tag, #0
        jump.eq lookup        ;tail recursive call to lookup
        getcar  val, exp      ;must be a combination
        cmp     val, #quote   ;relies on symbols having a 0 tag
        jump.ne notquot
        getcdr  val, exp
        getcar  val, exp
        rts                   ;return (cadr exp)
noquot: cmp     val, #cond
        jump.ne notcnd
        getcdr  exp, exp
        jump    evcond        ;tail recursive call to evcond
...
notdef: getcar  unev, exp     ;the apply case
        push    unev          ;save unevaled operator
        push    env
        getcdr  unev, exp     ;unevaled arguments
        jsr     evlist        ;non tail recursive -- arg vals
        pop     env
        pop     exp           ;restoring unev to exp!
        push    val           ;save the list of values
        jsr     mceval        ;call self to get proc
        load    proc, val     ;1st argument for apply
        pop     args          ;pop list of vals
        jump    mcapply       ;and tail recursively go apply

(define evlist
  (lambda (lst env)           ;map evaluator over list
    (cond ((null? lst) '())
          (else (cons (mc-eval (car lst) env)
                      (evlist (cdr lst) env))))))

evlist: gettag  tag, unev
        cmp     tag, #4      ;is it empty list?
        jump.ne evbody
        load    val, unev    ;simply return empty list
        rts
evbody: getcar  exp, unev    ;save this for later for mc-eval
        push    exp
        getcdr  unev, unev   ;cdr down list
        push    env          ;and save what else we need
        jsr     evlist       ;call self
        pop     env          ;restore what we saved
        pop     exp
        push    val          ;save tail of evaled elements
        jsr     mceval       ;evaluate first element
        load    arg1, val    ;make it the car argument to cons
        pop     val          ;get previous saved tail
        load    arg2, val    ;make it the cdr argument to cons
        jump    cons         ;tail recursively let cons do the work
```