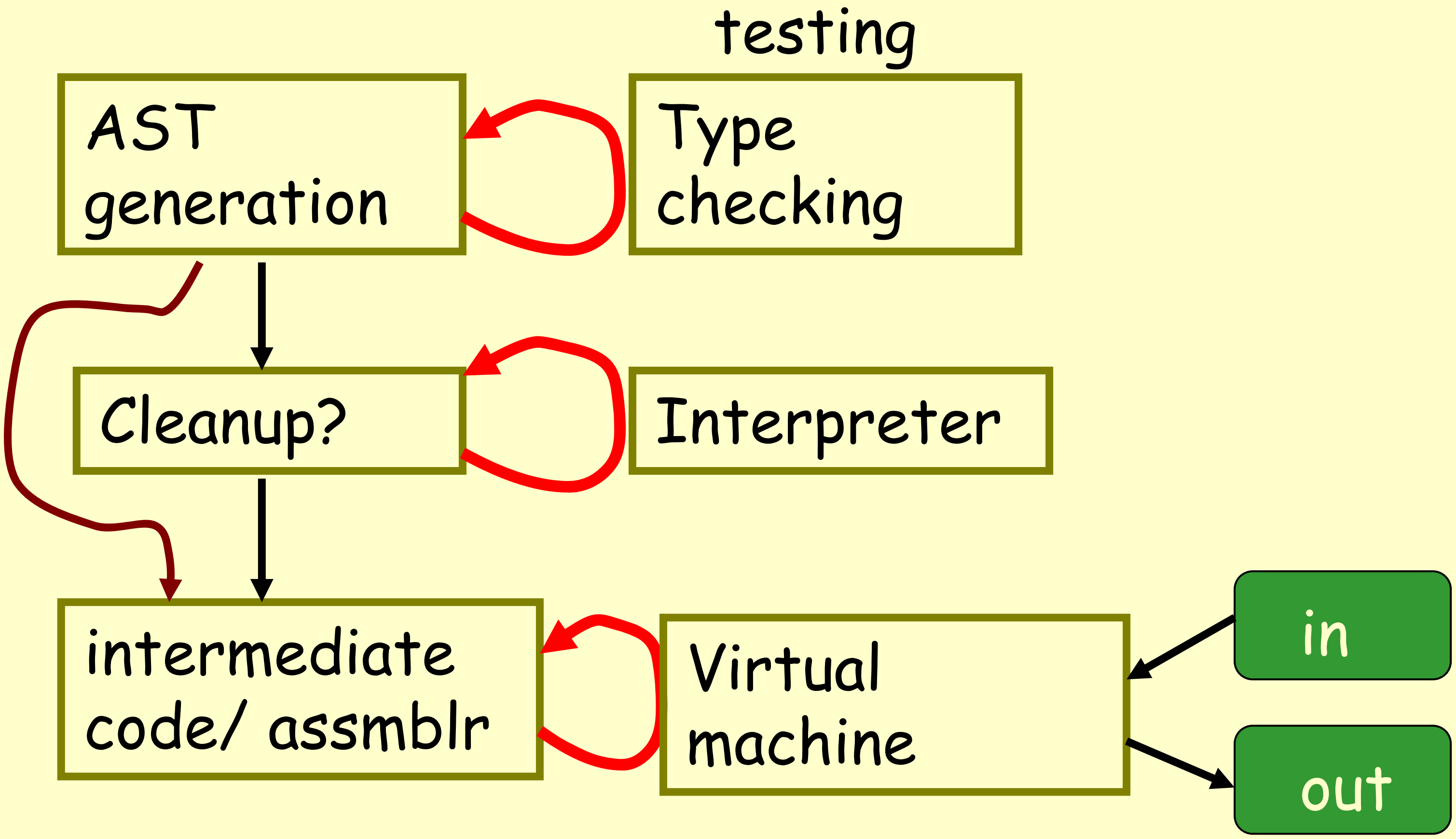
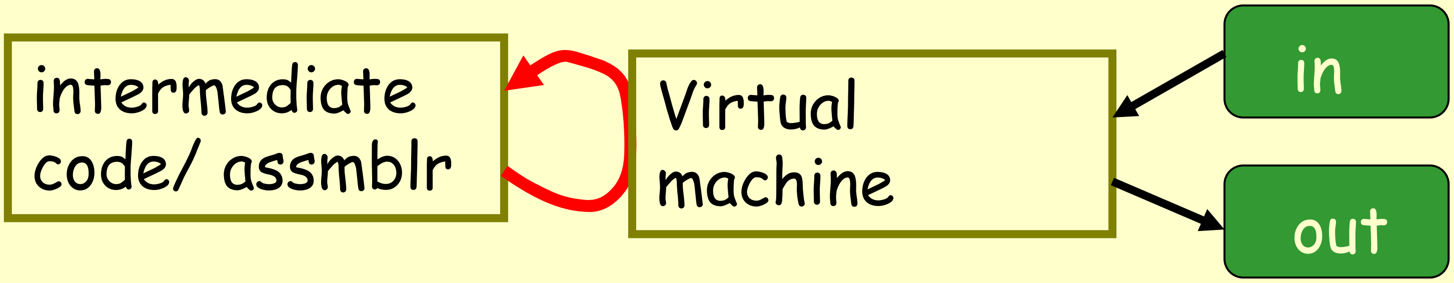
Introduction to Intermediate Code, virtual machine implementation

Lecture 19

Where do we go from here?



Details of MJ → what the VM must support
Details of the VM → what IC to generate



What is Intermediate Code? No single answer…

- Any encoding further from the source and closer to the machine. Possibilities:
  - Same except remove line/column numbers!
  - Change all parameter or local NAMES to stack offset counts
  - Change all global references (vars, methods) to vtable counts
  - Possible spew out Lisp as an IC. [My favorite: map every language you encounter into Common Lisp. Macro defs make it possible to define an intermediate level language "especially suited to IC" in Lisp.]
- Not (usually) machine code itself
- Usually some extra "abstraction"
  - Imagine you have arbitrary numbers of registers for calculations.
  - Imagine you have "macro" instructions like x=a[i,j]

What is Intermediate Code? Advantages

- Generally relatively portable between machines
- One IC may support several languages (a typical set might be C, C++, Pascal, Fortran) where the resources needed are similar. (If you can support C, the rest of them are pretty easy.)
- Languages with widely-differing semantics will not fit in this restricted set and may require an extended IC.
- E.g. Java IC supporting Scheme is hard because Scheme has 1st-class functions. Scheme IC supporting Java is plausible structurally but might not be as efficient.

Forms of Intermediate Code

- Virtual stack machine
  - simplified “macro” machine
- 3-address code
  - A :=B op C
- Register machine models
  - with unlimited number of registers, mapped to real registers later
- Tree form (= lisp program, more or less)

Using Intermediate Code

- We need to make some progress towards the machine we have in mind
  - Subject to manipulation, “optimization”
  - Perhaps several passes
  - Or, we can generate assembler
  - Or, we could plop down in absolute memory locations the binary instructions needed to execute the program "compile-and-go" system. This was a common "student compiler" technique for Pascal and Fortran.

Reminder of what we are doing.. From a tree representation like our AST..

- Instead of typechecking or interpreting the code, we are traversing it and putting together a list of instructions… generating IC … that if run on a machine -- would execute the program.
- In other words instead of interpreting or typechecking a loop,or going through a loop executing it, we write it out in assembler.

The simplest example

Compiling the MJ program segment … 3+4…

The string `"3+4"` [too simple to have line/column numbers]

parses to `(Plus (IntegerLiteral 3) (IntegerLiteral 4))`

typechecker approves and says it is of type int.

A program translating to lisp produces

```
(+ 3 4)
```

Which could be executed…
But we don't really want Lisp, we want machine code. We could start from the Lisp or from the AST, i.e. …. (Plus …)…

Just a simple stack machine (oversimplified)

Compiling

```
(+ 3 4)

(some-kind-of-compiler '(+ 3 4)) ➔ ;;  I made-up name ☺
((pushi 3) (pushi 4) (+))
```

Result is a list of assembly language instructions

```
   push immediate constant 3 on stack
   push immediate constant 4 on stack
   +  = take 2 args off stack and push sum on stack
```

Conventions: result is left on top of stack.
Location of these instructions unspecified.
Lengthy notes in simple-machine file describe virtual machine.

It doesn't have to look like lisp if we write a printing program

```
(pprint-code compiled-vector) ;; prints out…
        pushi 3
        pushi 4
        +
```

Consider the file Simple-compiler.fasl

Load this file and you have functions for compiling methods, exps, statements. You can trace them.

```
mj-c-method  compiles one method
mj-c-exp compiles one expression
mj-c-statement compiles one statement

Each program calls "emit" to add to the generated
program some sequence of instructions.
 Typically these are consed on to the front of a list
intended to become the "body" of a method. This list
which is then reversed, embroidered with other
instructions, and is ready to assemble.
```

Some FAQ. 1. Redundant work?

- Q: Going back to the AST for compiling, it seems to me that I am re-doing things I already did (or did 95%) for type-checking.
- A. You are right. Next time you write a compiler (hehe) you will remember and maybe you will save the information some way. E.g. save the environment / inheritance hierarchy, offsets for variables, type data used to determine assembler instructions, etc.

FAQ 2. Where do the programs live?

- You might ask this question; how are the methods placed in memory?
- For now we do not have to say, but we could let a "loader" determine where to put each code segment in memory. Each method can refer to instructions in its body by a relative address; a call sets the program counter (PC) to the top of the method's code.
- In a "real" program, a loader would resolve the references to methods /classes/ etc defined elsewhere. MJ has no such problems. (discuss why?)
- Or we could let all this live in some world where there is a symbol table (like Lisp) and lets us just grab the definition when we want to get it. (Dynamic loading, too)

FAQ 3: Too many layers?

- Q. There are too many pieces up in the air. Where do I start?
- A. Yes, we are faced with a multi-level target for understanding.
- That's why we took it in steps up to here. Two more levels, closely tied together.
- We produce code for the Assembler
  - Understand the assembler input / output
  - Requires understanding VM
- The VM determines what instructions make sense to generate to accomplish tasks (esp. call/return, get/set data
- The programming language definition determines what we need. E.g. Where does "this" object come from?
  - Look at output of translator
  - You see what to generate (or equivalent…)

The 2-pass assembler… (50 lines of code?)

What does the assembler program do with a list of symbolic instructions – the reversed list mentioned previously?
1. Turn a list of instructions in a function into a vector.

```
;; count up the instructions and keep track of the labels
;; make a note of where the labels are (program counter
;; relative to start of function).
;; Create a vector of instructions.
;; The transformed "assembled" program can support fast
;; jumps forward and back.

(multiple-value-bind (length labels)
   ;extract 2 items from 1st pass
    (asm-first-pass (fn-code fn))
  (setf (fn-code fn) ;; put vector back instead of list
        (asm-second-pass (fn-code fn)
                         length labels))
  fn))
```

What does the assembler do?

- Transforms a list of symbolic instructions and labels into a vector.
  - Turn a <u>list</u> of instructions in a function into a <u>vector</u>.
  - When there is a jump <label>, replace with a jump <integer> where the <integer> is the location of that <label>
- First pass merely counts up the instructions and keeps track of the labels. Produces a lookup-table (e.g. assoc. list) for label → integer mapping.
- Second pass creates the vector with substitutions

The 1st pass counts up locations

```
(defun asm-first-pass (code)
  "Return the label assoc list"
  (let ((length 0)
        (labels nil))
    (dolist (inst code)
      (if (label-p inst)
          (push (cons inst length) labels)
          (incf length)))
    labels))
;; could return (values length labels) ☺
```

The 2nd pass resolves labels, makes vector

```
(defun asm-second-pass (code labels)
  "Put code into code-vector, adjusting for labels."
  (coerce
   (map 'list
        #'(lambda (inst)
            (if (member (car inst) '(jump jumpn jumpz pusha call))
                `(,(car inst)
                  ,(cdr (assoc (second inst) labels))
                  ,@(cddr inst))
                inst))
        (remove-if #'label-p code))
   'vector))
```

The MJ Virtual Machine

The easy parts

It is a stack-based architecture simulated by a Lisp program.

```
   How does this differ from the MIPS architecture in CS61c?
1.  No registers for values of variables
2.  (there are some implicit registers fp, sp, pc)
```

It is a stack-based architecture simulated by a Lisp program

- Some differences from CS61c MIPS/SPIM architecture
  - Registers not used for passing arguments
  - No "caller saved" or "callee saved" registers
  - Only implicit registers (e.g. program counter, frame pointer, stack pointer)
  - Debugging in the VM
  - I/O much different
  - No interrupts
  - No jump delay slot
  - No floating point co-processor
  - Undoubtedly more differences

We set up an update-program-counter/ execute loop

```
(defun run-vm (vm)
  ;; set up small utilities
        (loop
         (vm-fetch vm)                          ; Fetch instruction / update PC
         (case (car (vm-state-inst vm))
           ;; Variable/stack manipulation instructions
           (lvar  (vpush (frame (a1))))
           (lset  (set-frame (a1) (vpop)))
           ;;; etc etc
           (pushi (vpush (a1)))
           (pusha (vpush (a1)))
           ;; Branching instructions:
           (jump  (set-pc (a1)))
           ;;; etc
           ;; Function call/return instructions:
         ;; …..
             ;; Arithmetic and logical operations:
         ;; …..
             ;; Other:
             ;;;….
             (t     (error "Unknown opcode: ~a" (vm-state-inst vm))))))
```

The outer exception handling controller

```
(defun run-vm (vm)
  ;; set up small utilities
 (handler-case

        (loop
         ;;process stuff
          )
        (error (pe)
          (format t "~%Caught error: ~a" pe)
          (if (vm-state-crash-hook vm)
              (funcall (vm-state-crash-hook vm) vm)
              (vm-state-print vm)))))
```

List of Opcodes (I)

DETAILS in simple-machine.lisp

```
;; the EASY ones

;; +, *, -, <, and - Operations on two variables (e.g. pop a, b, push a-b)
;; not            - Operates on one variable
;;
;; print  - Pops an integer and prints it
;; read   - Reads an integer and pushes it
;; exit s - Exits with status code s
;;
;; debug  - Ignored by the machine; may be used for debugging hooks
;; break  - Aborts execution; may be useful for debugging
```

List of Opcodes (II)

```
;; lvar i  - Gets the ith variable from the stack frame
;; lset i  - Sets the ith variable from the stack frame
;; pop     - Pops the stack
;; swap    - Swaps the top two elements
;; dup i   - Duplicates the ith entry from top of stack
;; addi i  - Adds an immediate value to the top of stack
;; alloc   - Pops a size from the stack; allocates a new array of that size
;; alen    - Pops an array, pushes the length
;; mem     - Pops index and array; pushes array[index].
;; smem    - Pops value, index, and array; sets array[index] = value.
;; pushi i - Push an immediate value
;; pusha i - Push an address
;;
;; jump a  - Jumps to a
;; jumpz a - Pops val, jumps to a if it's zero
;; jumpn a - Pops val, jumps to a if it isn't zero
;; jumpi   - Pops an address and jumps to it
;;
;; call f n - Calls function f with n arguments.
;; calli n  - Calls a function with n arguments.  Address popped from stack.
;; frame m  - Push zeros onto the stack until there are m slots in the frame.
;; return   - Returns from a call.  Stack should contain just the return val.
```

Architecture for arithmetic is based on underlying lisp arithmetic, e.g. + ≡ #'+

`Machine arithmetic can be defined as anything we wish`
`(arbitrary precision?  16 bit, 32 bit, 64 bit? )`

Machine + assembler in simple-machine.lisp

250 lines of code, including comments ☺

```
(defun run (exp) ;;exp is (cleaned-up-perhaps ast)
  "compile and run MJ code"
  (machine (assemble (mj-compile exp))))
```

(machine(assemble(mj-compile(cleanup(mj-parse filename))))) is equivalent to run-mj, though we should check for semantic errors in there, too.

Details, lots, in the on-line stuff; Discussion in sections.