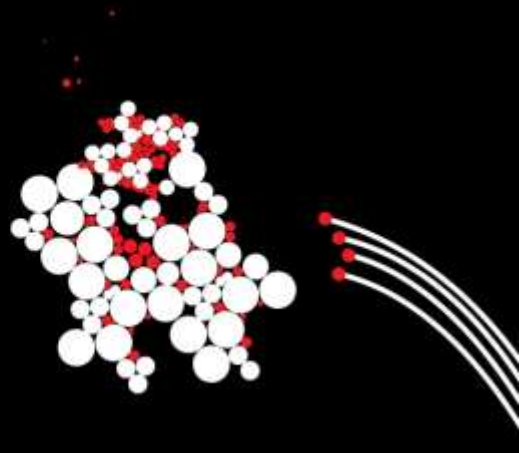
UNIVERSITY OF TWENTE.

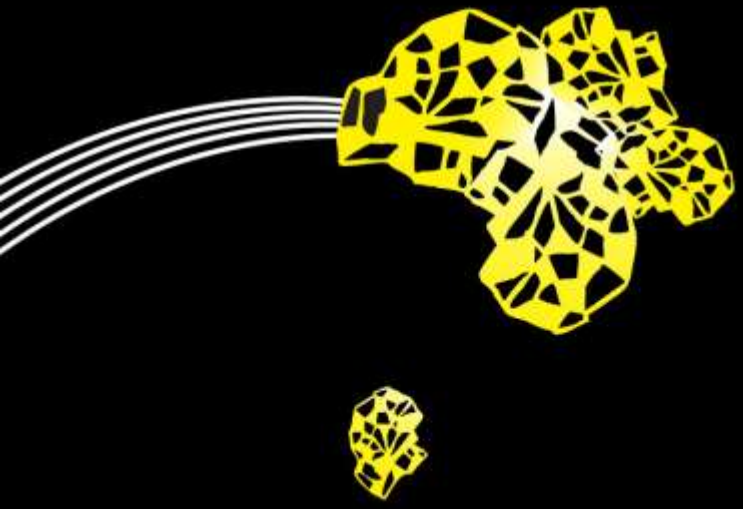


COMPILER CONSTRUCTION:

CC 4-2: LINEAR IR

17 MAY 2019

MODULE 8: PROGRAMMING PARADIGMS



LINEAR INTERMEDIATE REPRESENTATIONS

- “Linear” as opposed to “graph-based”
- Essentially: sequence of low-level instructions
  - Not necessarily target machine instructions
  - Abstract from available #registers
  - Abstract from precise memory layout
- Different linear IRs for different "abstract" machines

1-ADDRESS, 2-ADDRES AND 3-ADDRESS CODE

a=3
b=9

- Consider expression $a - (2 \times b)$
- One-address code
  - Stack-based
  - See final project

```
push  a
push  2
push  b
mult
sub
```

9 (b)
2
3 (a)
stack

- Two-address code
  - Destructive register-based
  - No longer en vogue

```
load  2   => r1
load  b   => r2
mult  r1  => r2
load  a   => r1
sub   r2  => r1
```

r2 := r1 * r2
r1 := r1 - r2

- Three-address code
  - Non-destructive register-based
  - Popular since RISC (Reduced Instruction Set Computers)
  - Used in EC ("ILOC")

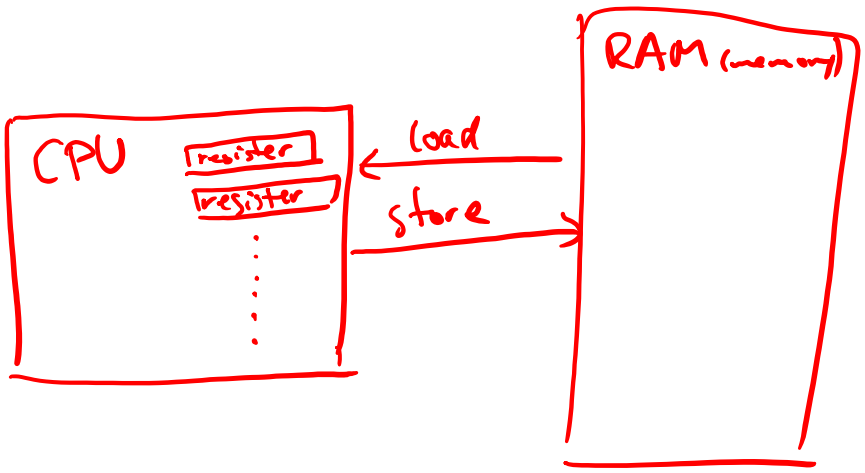
```
loadI   2          => r1
loadAI  r_arp, @b  => r2
mult    r1, r2     => r3
loadAI  r_arp, @a  => r4
sub     r4, r3     => r5
```

r3 := r1 * r2

variant: SSA (LLVM)

ILOC: EC APPENDIX A

- Intermediate Language for an Optimizing Compiler
  - Assembly language for a simple RISC machine
  - Three-address code
- Abstract machine for ILOC:
  - Unlimited number of registers
  - Separate main memory, load and store
  - Arithmetic and control-flow operations work with registers only



ILOC: EC APPENDIX A

- Intermediate Language for an Optimizing Compiler
  - Assembly language for a simple RISC machine
  - Three-address code
- General format (in EBNF)

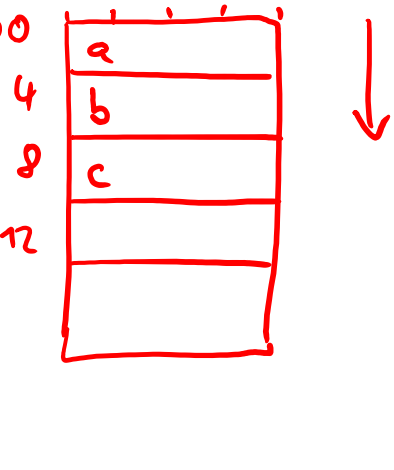
```
instr  → ( label ':' )? opcode sources '=>' targets
       | ( label ':' )? opcode sources '->' targets
```

  - `label`: optional symbolic label, used as jump target
  - `opcode`: symbolic name of the instruction
  - `sources`: zero, one or two source operands
  - '`=>`' for value manipulation instructions
  - '`->`' for jump instructions
  - `targets`: one or two target operands
- Operands
  - *Constants*: either concrete numbers or symbolic names (preceded by @)
  - *Registers*: initially unbounded number, restricted on concrete machine
  - *Labels*: in case of jump instructions

EXAMPLE COMPUTATION (FROM BLOCK 1)

- Assignment:

```
function bla(int a, int b, int c) {
    a <- a * 2 * b * c * d
    ...
}
```



register containing base memory address
(arp = “activation record pointer”)

offset of variable a

ILOC:

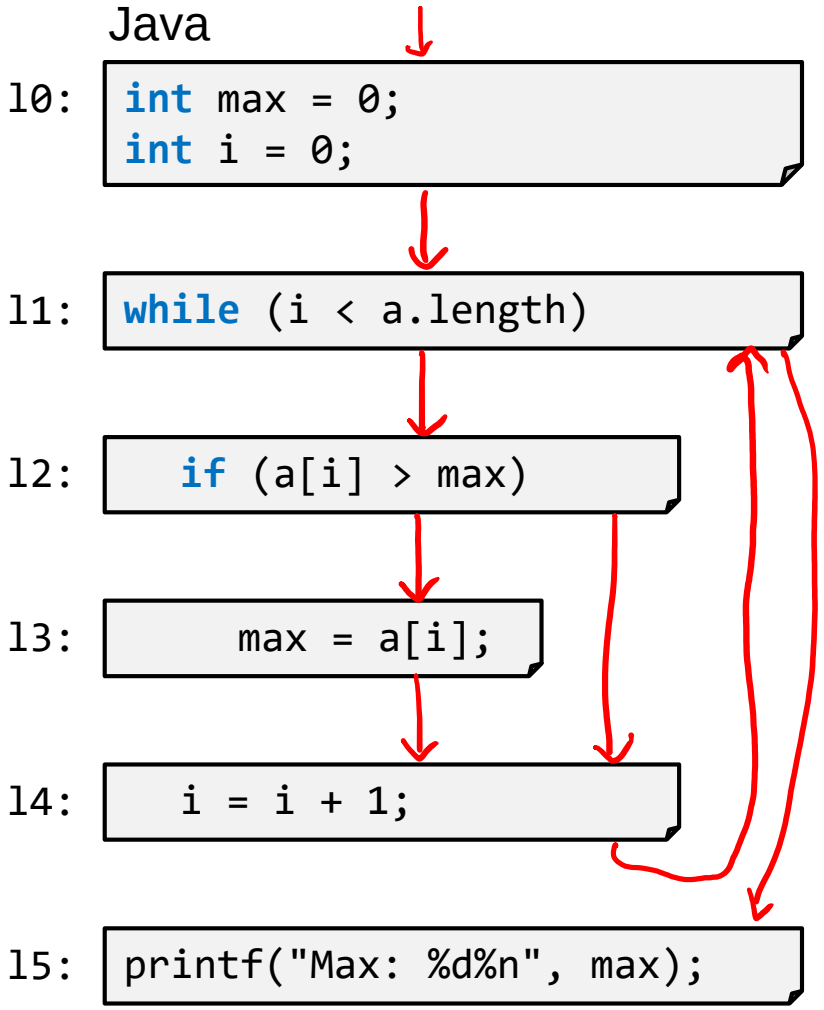
```
loadAI  r_arp,@a  => r_a      // load 'a'
loadI   2         => r_2      // constant 2 into r_2
loadAI  r_arp,@b  => r_b      // load 'b'
loadAI  r_arp,@c  => r_c      // load 'c'
loadAI  r_arp,@d  => r_d      // load 'd'
mult    r_a,r_2   => r_a      // r_a <- a * 2
mult    r_a,r_b   => r_a      // r_a <- (a * 2) * b
mult    r_a,r_c   => r_a      // r_a <- (a * 2 * b) * c
mult    r_a,r_d   => r_a      // r_a <- (a * 2 * b * c) * d
storeAI r_a       => r_arp,@a // write r_a back to 'a'
```

MAX COMPUTATION IN JAVA AND ILOC

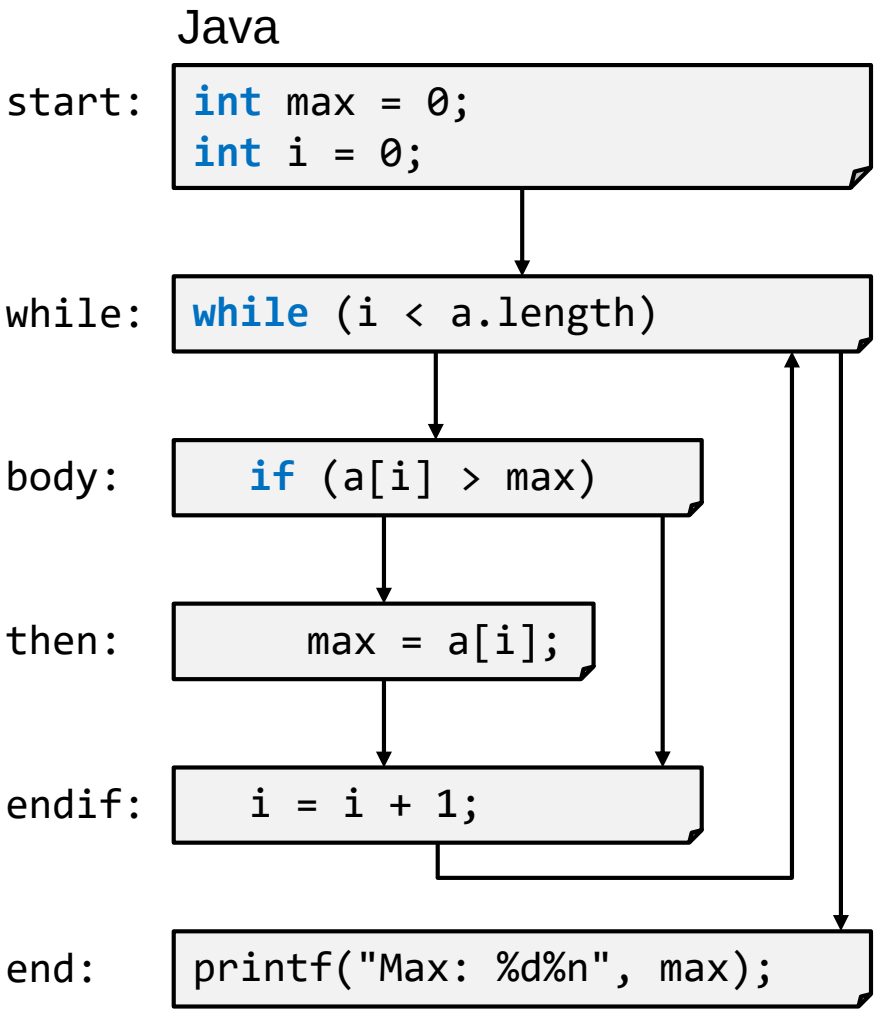
Java

```
int max = 0;
int i = 0;
while (i < a.length) {
   if (a[i] > max) {
      max = a[i];
   }
   i = i + 1;
}
printf("Max: %d%n", max);
```

MAX COMPUTATION IN JAVA AND ILOC



MAX COMPUTATION IN JAVA AND ILOC



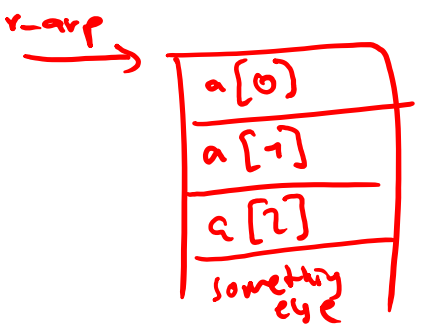
- Need to decide on memory layout
  - Where do we put `max` and `i`?

in registers r_max and r_i

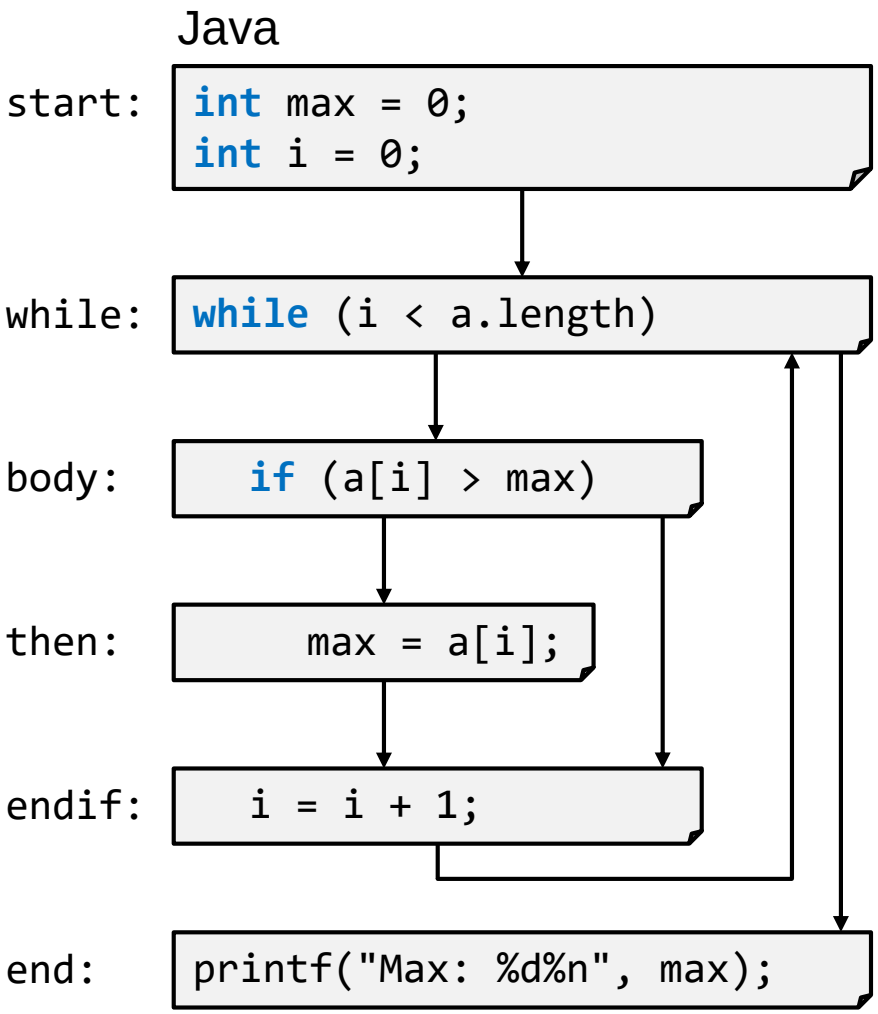
  - Where do we get `a`?
  - Where do we get `a.length`?

at offset @a after value of r_arp in memory

as constant @a.length = 3



MAX COMPUTATION IN JAVA AND ILOC



ILOC

```
start: loadI 0 => r_max
       loadI 0 => r_i
       loadI @alength => r_len

while: cmp_LT r_i,r_len => r_cmp
       cbr r_cmp -> body, end

body:  i2i r_i => r_a
       multI r_a,4 => r_a
       addI r_a,@a => r_a
       loadAO r_arp,r_a => r_ai
       cmp_GT r_ai,r_max => r_cmp
       cbr r_cmp -> then,endif

then:  i2i r_ai => r_max

endif: addI r_i,1 => r_i
       jumpI -> while

end:   out "Max: ", r_max
```

MAX COMPUTATION IN JAVA AND ILOC

Java

```
int max = 0;
int i = 0;
while (i < a.length) {
   if (a[i] > max) {
      max = a[i];
   }
   i = i + 1;
}
printf("Max: %d%n", max);
```

ILOC

```
start: loadI 0 => r_max
       loadI 0 => r_i
       loadI @alength => r_len
while: cmp_LT r_i,r_len => r_cmp
       cbr r_cmp -> body, end
body:  i2i r_i => r_a
       multI r_a,4 => r_a
       addI r_a,@a => r_a
       loadAO r_arp,r_a => r_ai
       cmp_GT r_ai,r_max => r_cmp
       cbr r_cmp -> then,endif
then:  i2i r_ai => r_max
endif: addI r_i,1 => r_i
       jumpI -> while
end:   out "Max: ", r_max
```

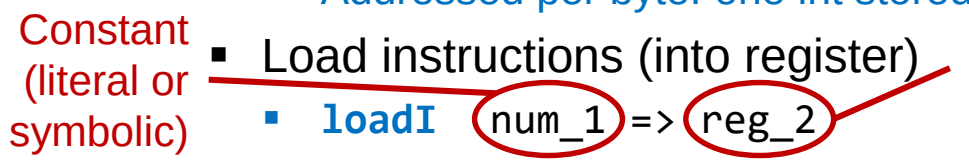
array address calculation

- Choices:
  - `max`, `i` stored in `r_max`, `r_i`
  - `a.length` given as `@alength`
  - array `a` stored in memory, from offset `r_arp + @a`
  - Every `int` takes 4 bytes of memory

This is not official ILOC:
Added by us for ease of debugging

REGISTERS VERSUS MEMORY

- Registers
  - Accessed by name
  - “Close” to processor, fast access
  - Limited number (but unbounded for now)
  - Hold a single integer (which can be a memory address) or Boolean
- Memory
  - Accessed by index: typically base address + offset
  - Slower access, but much more space
  - Addressed per byte: one int stored over four successive bytes (MSB first)
- Load instructions (into register)




```
loadI  num_1 => reg_2          // reg_2 <- num_1
load   reg_1 => reg_2          // reg_2 <- MEMORY(reg_1)
loadAI reg_1, num_2 => reg_3   // reg_3 <- MEMORY(reg_1+num_2)
loadAO reg_1, reg_2 => reg_3   // reg_3 <- MEMORY(reg_1+reg_2)
```

- Analogous for store instructions (from register)

MEMORY MODELS

- Register-to-register
  - Use registers as much as possible, also for (local) variables
  - Start by assigning every temporary value to different register
    - For ease of optimization
  - When translating to real machine (*register allocation*):
    - Reuse registers wherever possible
    - Shift registers to memory
- Memory-to-memory
  - Immediately store all variables in memory
    - Store again upon every change in value
  - Program is “legal” from the start
    - Runs on every machine
  - When translating to real machine (optimization):
    - Shift memory locations to registers
- Register-to-register more suitable for (later) optimization

CONTROL FLOW

- Compare instructions
  - `cmp_LT r_1,r_2 => r_3` `// r_3 <- (r_1 < r_2)`
  - `cmp_LE r_1,r_2 => r_3` `// r_3 <- (r_1 <= r_2)`
  - `cmp_EQ r_1,r_2 => r_3` `// r_3 <- (r_1 = r_2)`
  - etc.
- Jump instructions
  - `jump reg_1` `// PC <- MEMORY(reg_1)` don't use this
  - `jumpI lab_1` `// PC <- lab_1`
  - `cbr reg_1 -> lab_1,lab_2` `// PC <- reg_1 ? lab_1 : lab_2`

label

program counter

FAKE INSTRUCTIONS (NOT TRULY ILOC)

- For simulation purposes
  - `in  string, reg  // prints string, asks for input`
  - `out  string, reg  // prints string and register`

SUPPORT FOR ILOC (PP LAB CODE)

- Provided as part of the lab files:
  - class `Program`
    - Encodes an ILOC program
  - class `Memory`
    - Simulates an array of memory locations
  - class `Machine`
    - Simulates memory + registers + symbolic constants
  - `Assembler`
    - Converts a textual ILOC program to a `Program` object
  - `Simulator`
    - Runs a `Program` object
    - After initialisation of the `Machine`
- Demo