Compiler Construction

IR

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- IR

- Linearize the CFG

- Linearize the data types

A compiler

High level programming language

↓

Front-end

↓

IR

↓

Middle-end

Code analysis *and* transformation

↓

IR

↓

Back-end

Today: translating explicit control flow and data types

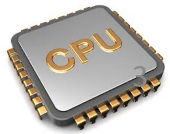
Instruction selection

Register allocation

Assembly generation

↓

Machine code



For simple data types (e.g., int64)
and simple control flows (e.g., straight line code),
L3 and IR are very similar

L3

```
define @main (){
  %myRes <- call @myF(5)
  %v1 <- %myRes * 4
  %v2 <- %myRes + %v1
  return
}
define @myF (%p1){
  %p2 <- %p1 + 1
  return %p2
}
```

IR

```
define void @main (){
  :entry
  int64 %myRes
  int64 %v1
  int64 %v2
  %myRes <- call @myF(5)
  %v1 <- %myRes * 4
  %v2 <- %myRes + %v1
  return
}
define int64 @myF (int64 %p1){
  :myLabel
  int64 %v0
  %v0 <- %p1 + 1
  return %v0
}
```

For simple data types (e.g., int64)
and simple control flows (e.g., straight line code),
L3 and IR are very similar

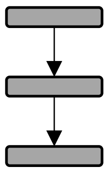
L3 and IR start to differ when we use
more complex data types (e.g., multi-dimensional arrays)
and more complex control flows (e.g., conditional branches)

IR features

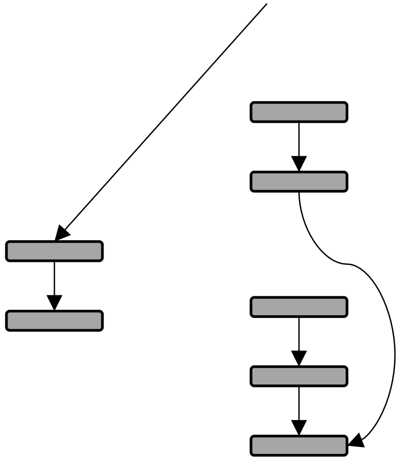
- The IR is a language designed for
  - The rich set of code analyses and transformations included in the middle-end of a compiler
  - Decoupling between source language-specific aspects and architecture-specific aspects
    - It does not imply portability
- Motivation
  - The middle-end job: **analyze, analyze, analyze**, and transform
  - To help analyzing the IR: explicit control flow
  - Liveness analysis is an example of what the middle-end does
  - Your liveness analysis had to "learn" who were the successors of an instruction
  - Successor/predecessor of an instruction: control flows
  - If I have 1000 code analyses, do they all have to "learn" the control flows?
  - Control flows need to be explicit in the code to simplify the middle-end
  - Solution: the IR language encodes computation as a graph called Control Flow Graph (CFG)

Representing the control flow of the program

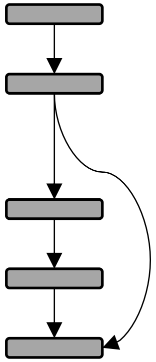
- Most instructions



- Jump instructions



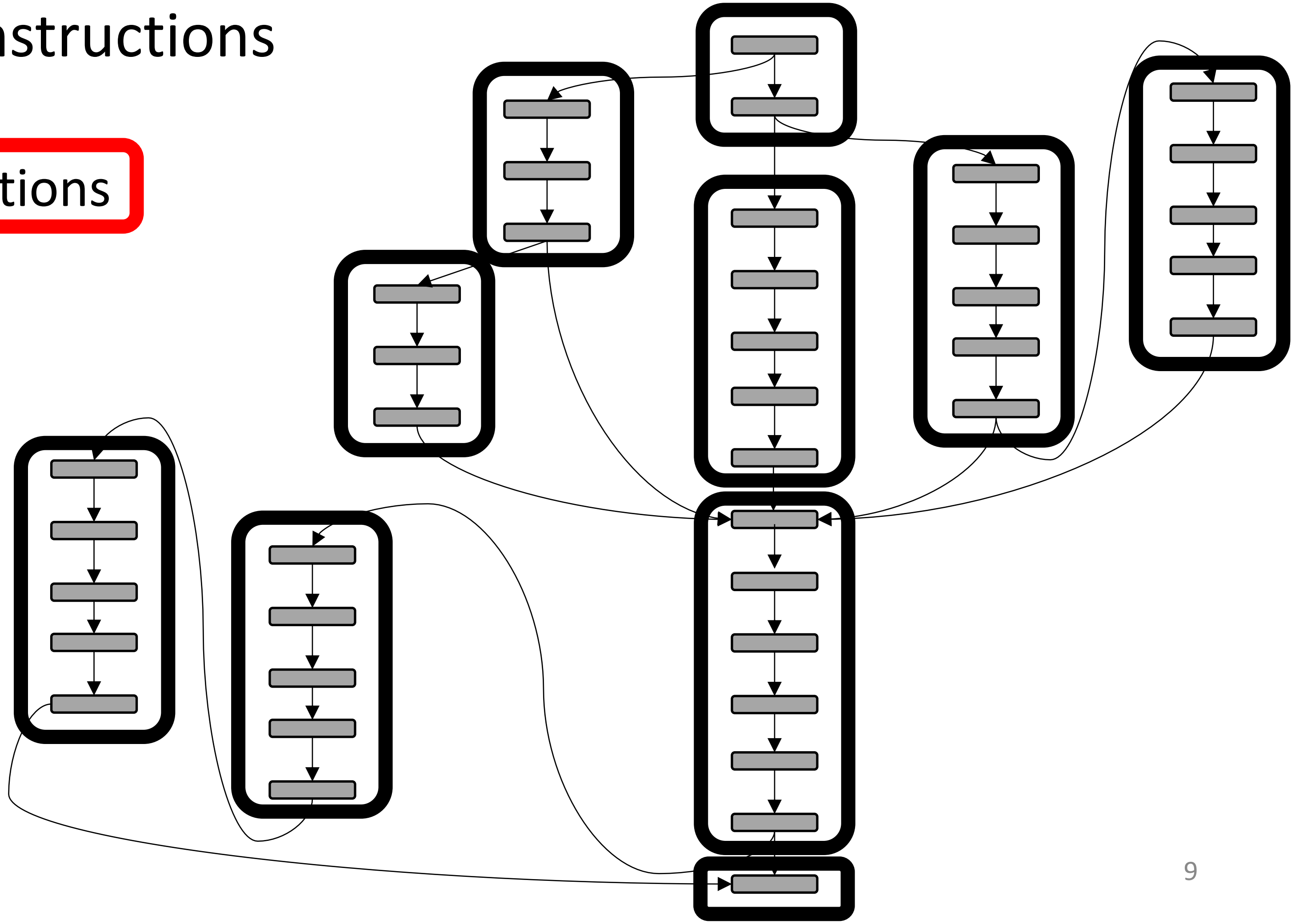
- Branch instructions



Representing the control flow of the program

A graph where nodes are instructions

- Very large
- Lot of straight-line connections
- Can we simplify it?



Basic block

Sequence of instructions that is always entered at the beginning and exited at the end

Basic blocks

A basic block is a maximal sequence of instructions such that

- Only the first one can be reached
  from outside this basic block

- All instructions within are executed consecutively
  if the first one get executed
  - Only the last instruction can be a branch/jump
  - Only the first instruction can be a label

- The storing sequence = execution order in a basic block

Basic blocks in compilers

- Automatically identified
  - Algorithm:
  - Code changes trigger the re-identification
  - Increase the compilation time
- Enforced by design
  - Instruction exists only within the context of its basic block
  - To define a function:
    - you define its basic blocks first
    - Then you define the instructions of each basic block

L3

```
p      ::= f+
f      ::= define l ( vars ) { i+ }
i      ::= var <- s | var <- t op t | var <- t cmp t |
           var <- load var | store var <- s |
           return | return t| label | br label | br t label |
           call callee ( args ) | var <- call callee ( args )
callee ::= u | print | allocate | input | tensor-error
vars   ::= | var | var (, var)*
args   ::= | t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
op     ::= + | - | * | & | << | >>
cmp    ::= < | <= | = | >= | >
N      ::= (+|-)? [0-9]+
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

IR

```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
           var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
           var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define int64 @myF (int64 %p1){
  :myLabel
  int64 %v1
  %v1 <- 1
  int64[][] %myMatrix
  int64[][][][][][][] %myTensor
  tuple %myHeterogeneousArray
  code %myFunctionPointer
  %myFunctionPointer <- @myF
  return %v1
}
```

IR

```
p      ::= f+
f      ::= define T I ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
           var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
           var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | I
t      ::= var | N
u      ::= var | I
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
I      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define void @main (){
  :entry
  call @myF(i, 2)
  return
}

define int64 @myF (int64 %p1,
int64 %p2){
  :entry
  int64 %v1
  %v1 = %p1 + %p2
  return %v1
}
```

IR

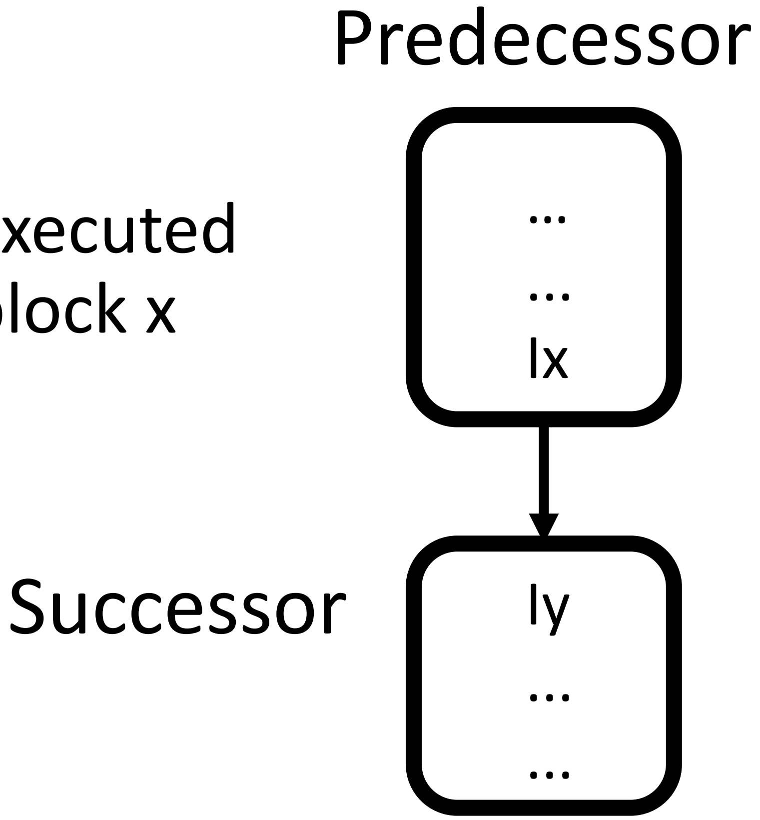
```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
           var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
           var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define void @main (){
  call print(1)
  return
}
```

Control Flow Graph (CFG)

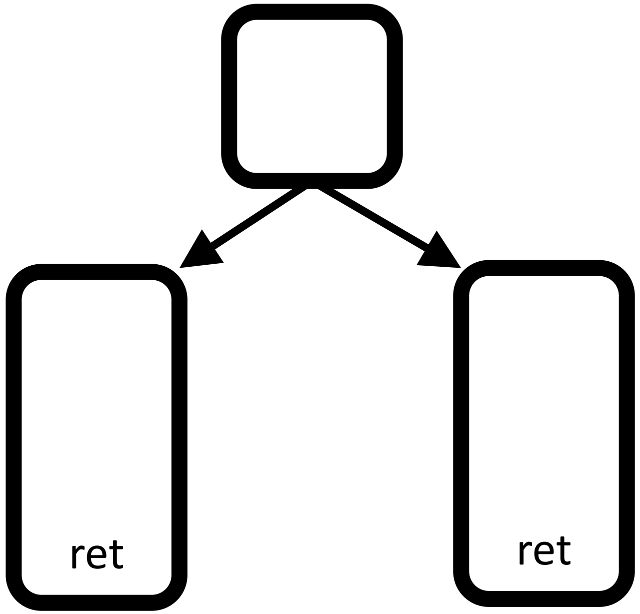
- A CFG is a graph G = <Nodes, Edges>
- Nodes: Basic blocks
- Edges: (x,y) ϵ Edges iff

  first instruction in basic block y **might** be executed just after the last instruction of the basic block x



Control Flow Graph (CFG)

- Entry node: block with the first instruction of the function
- Exit nodes: blocks with the return instruction
- All basic blocks beside the first can be stored in any order



IR function is a CFG

IR encodes the CFG explicitly by

- Enforcing (in the grammar)
  explicit predecessor-successor relations between basic blocks
- Enforcing the first basic block is the entry point of the function

IR

```
p       ::= f+
f       ::= define T l ( (type var)* ) { bb+ }
bb      ::= label i * te
te      ::= br label | br t label label | return | return t
i       ::= type var | var <- s | var <- t op t |
            var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
            call callee ( args? ) | var <- call callee ( args? ) |
            var <- new Array(args) | var <- new Tuple(t)
T       ::= type | void
type    ::= int64([])* | tuple | code
callee  ::= u | print | input | tensor-error
args    ::= t | t (, t)*
s       ::= t | label | l
t       ::= var | N
u       ::= var | l
N       ::= (+|-)? [0-9]+
op      ::= + | - | * | & | << | >> | < | <= | = | >= | >
l       ::= @name
label   ::= :name
var     ::= %name
name    ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define int64 @myF (int64 %p1){
  :myLabel
  int64 %c
  %c <- %p1 >= 3
  br %c :true :false

  :true
   return 1

  :false
  return 0
}
```

IR makes the CFG explicit

IR also makes complex data types explicit

- Explicit data types
- Explicit allocations

IR

```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
           var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
           var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

Implicit initialization to "1"

```
define int64 @myF (int64 %p1){
  :myLabel
  int64[] %v
  %v <- new Array(7)
  %v[1] <- 1
  return 0
}
```

IR

```
p       ::= f+
f       ::= define T l ( (type var)* ) { bb+ }
bb      ::= label i * te
te      ::= br label  | br t label label | return | return t
i       ::= type var | var <- s | var <- t op t |
            var <- var([t])+ |  var([t])+ <- s | var <- length var t | var <- length var |
            call callee ( args? ) | var <- call callee ( args? ) |
            var <- new Array(args) | var <- new Tuple(t)
T       ::= type | void
type    ::= int64([])* | tuple | code
callee  ::= u | print | input | tensor-error
args    ::= t | t (, t)*
s       ::= t | label | l
t       ::= var | N
u       ::= var | l
N       ::= (+|-)? [0-9]+
op      ::= + | - | * | & | << | >> | < | <= | = | >= | >
l       ::= @name
label   ::= :name
var     ::= %name
name    ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

Implicit initialization to "1"

```
define int64 @myF (int64 %p1){
  :myLabel
  int64[][] %v
  %v <- new Array(7,7)
  %v[0][3] <- 1
  return 0
}
```

IR

```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
           var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
           var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

Implicit initialization to "1"

```
define int64 @myF (int64 %p1){
  :myLabel
  int64[][][] %v
  %v <- new Array(7,7,7)
  %v[0][1][3] <- 1
  return 0
}
```

IR

```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
             var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
             var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

Implicit initialization to "1"

```
define int64 @myF (int64 %p1){
  :myLabel
  tuple %t
  %t <- new Tuple(7)
  %t[0] <- 3
  return 0
}
```

IR makes the CFG explicit

IR also makes complex data types explicit

- Explicit data types
- Explicit allocations
- Explicit access to size objects
    - E.g., length of an array

IR

```
p       ::= f+
f       ::= define T l ( (type var)* ) { bb+ }
bb      ::= label i * te
te      ::= br label | br t label label | return | return t
i       ::= type var | var <- s | var <- t op t |
            var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
            call callee ( args? ) | var <- call callee ( args? ) |
            var <- new Array(args) | var <- new Tuple(t)
T       ::= type | void
type    ::= int64([])* | tuple | code
callee  ::= u | print | input | tensor-error
args    ::= t | t (, t)*
s       ::= t | label | l
t       ::= var | N
u       ::= var | l
N       ::= (+|-)? [0-9]+
op      ::= + | - | * | & | << | >> | < | <= | = | >= | >
l       ::= @name
label   ::= :name
var     ::= %name
name    ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define int64 @myF (int64 %p1){
  :myLabel
  int64[][][] %v
  int64 %l_0
  int64 %l_1
  int64 %l_2
  %v <- new Array(7,7,7)
  %l_0 <- length %v 0
  %l_1 <- length %v 1
  %l_2 <- length %v 2
  return 0
}
```

IR

```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
           var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
           call callee ( args? ) | var <- call callee ( args? ) |
           var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define int64 @myF (int64 %p1){
  :myLabel
  tuple %v
  int64 %l
  %v <- new Tuple(7)
  %l <- length %v
  return 0
}
```

IR makes the CFG explicit

IR also makes complex data types explicit

- Explicit data types
- Explicit allocations
- Explicit access to size objects
    - E.g., length of an array
- Object accesses only possible in simple assignments

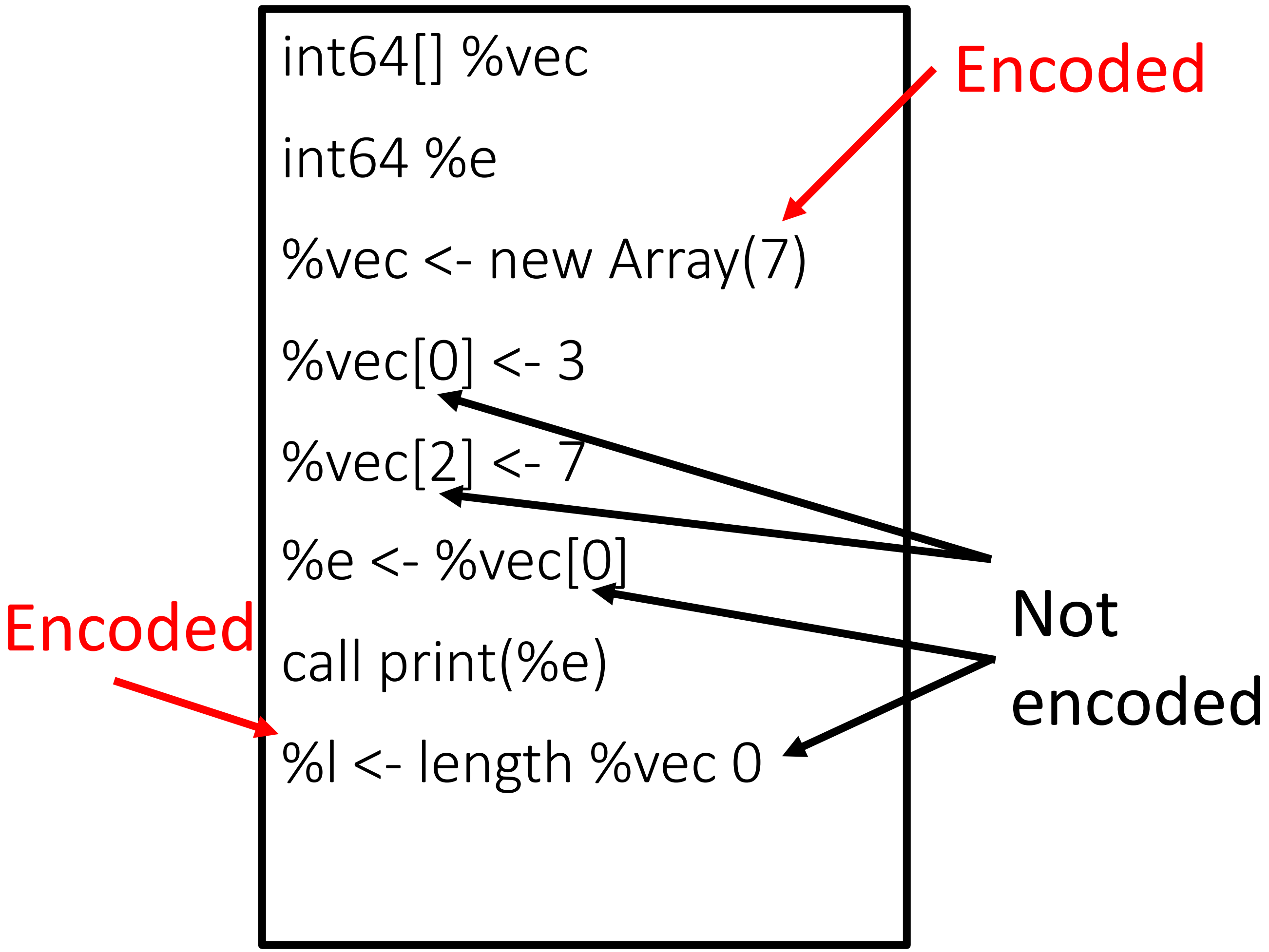
IR

```
p      ::= f+
f      ::= define T l ( (type var)* ) { bb+ }
bb     ::= label i * te
te     ::= br label | br t label label | return | return t
i      ::= type var | var <- s | var <- t op t |
             var <- var([t])+ | var([t])+ <- s | var <- length var t | var <- length var |
             call callee ( args? ) | var <- call callee ( args? ) |
             var <- new Array(args) | var <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
callee ::= u | print | input | tensor-error
args   ::= t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
l      ::= @name
label  ::= :name
var    ::= %name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
define int64 @myF (){
  :myLabel
  int64[] %ar
  int64 %v0
  int64 %v1
  int64 %t
  %ar <- new Array(7)
  %v0 <- %ar[0]
  %v1 <- %ar[1]
  %t <- %v0 + %v1
  return 0
}
```

```
define int64 @myF (){
  :myLabel
  int64[] %ar
  int64 %t
  %ar <- new Array(7)
  %t <- %ar[0] + %ar[1]
  return 0
}
```

Indices of array and tuple accesses are not encoded



Indices and dimension# in length are not encoded

- Accessing length of a dimension

%l <- length %ar %dimID

- Accessing length of a tuple

%l <- length %tuplePtr

- Accessing array element

%ar[%e1][%e2] <- %v1

%v2 <- %ar[%e1][%e2]

- Allocating an array

%ar <- new Array(%dim1, %dim2)

Encoded

Not encoded

Variable definition

- The code must defines (statically) a variable before using it
- All variable definitions must appear in the function before all of its uses and at the entry point basic block

```
int64 %d
%d <- 5
```

```
%d <- 5
int64 %d
```

```
:E
br :S
:S
int64 %d
%d <- 5
return
```

Final notes on IR

Same undefined behaviors as for L3

but without the one related to the last instruction of a function

L3

```
define @myF (%p1){
  %p2 <- %p1 + 1
}
```

IR

```
define void @myF (int64 %p1){
  int64 %p2
  %p2 <- %p1 + 1
  return
}
```

Now that you know the IR language

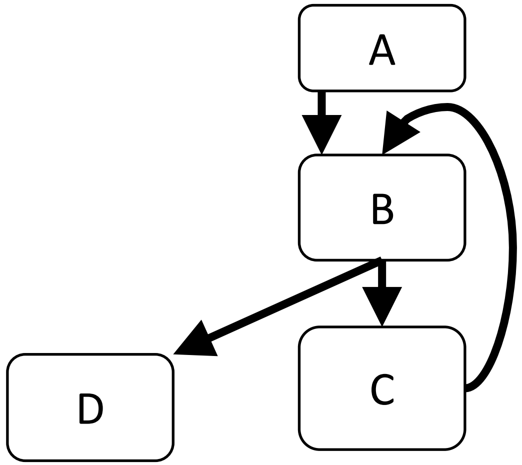
Rewrite all of your L3 programs in IR

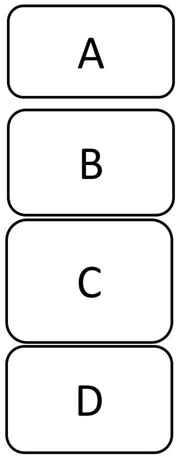
Outline

- IR
- Linearize the CFG
- Linearize the data types

From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3
- Any order will preserve the original semantics
  as long as the entry point BB is the first one (property of the CFG)





Naïve solution (not ok for your homework)

- Ignore the problem

- In other words:
  the sequence of basic blocks described in the IR program file
  is going to be the sequence chosen

- Translate a two labels IR branch into 2 branches in L3

```
br %cond :TRUE :FALSE
```

Your work →

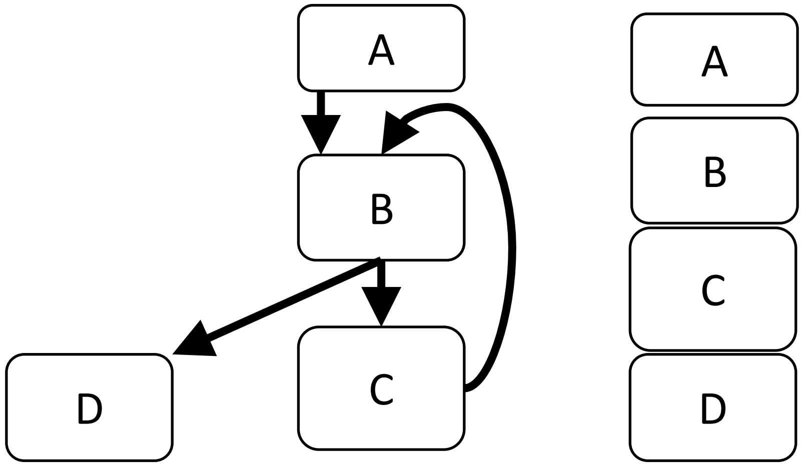
```
br %cond :TRUE

br :FALSE
```

From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3
- Any order will preserve the original semantics
  as long as the entry point BB is the first one (property of the CFG)
- Different orders will have a different #branches
- We want to select the one with the lowest #branches
  - Run-time vs. compile-time

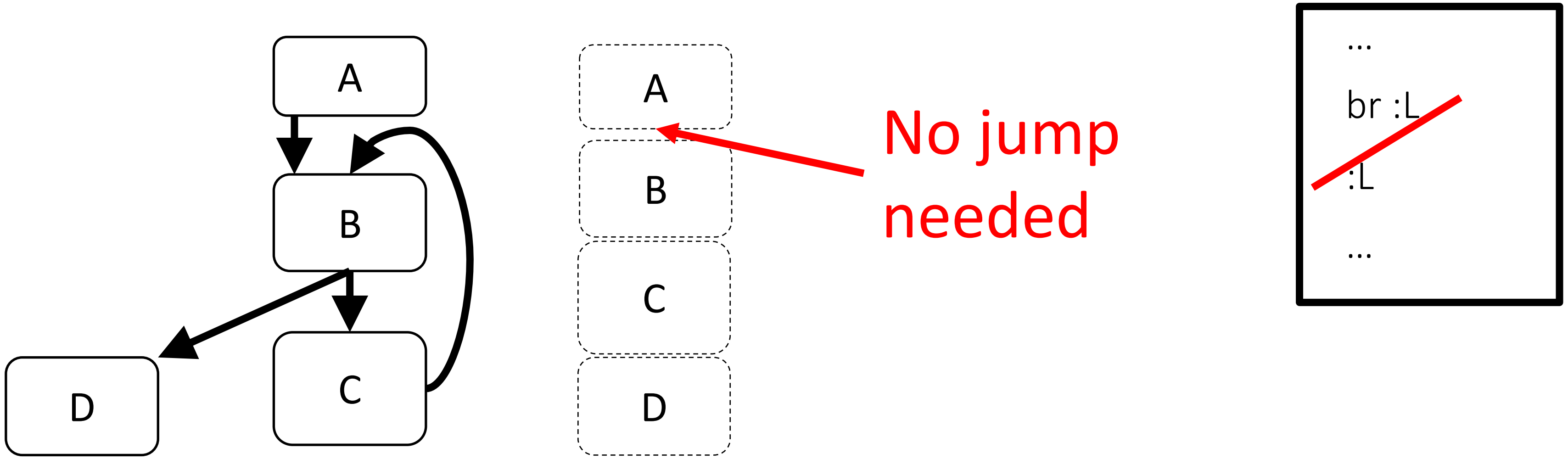
From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3



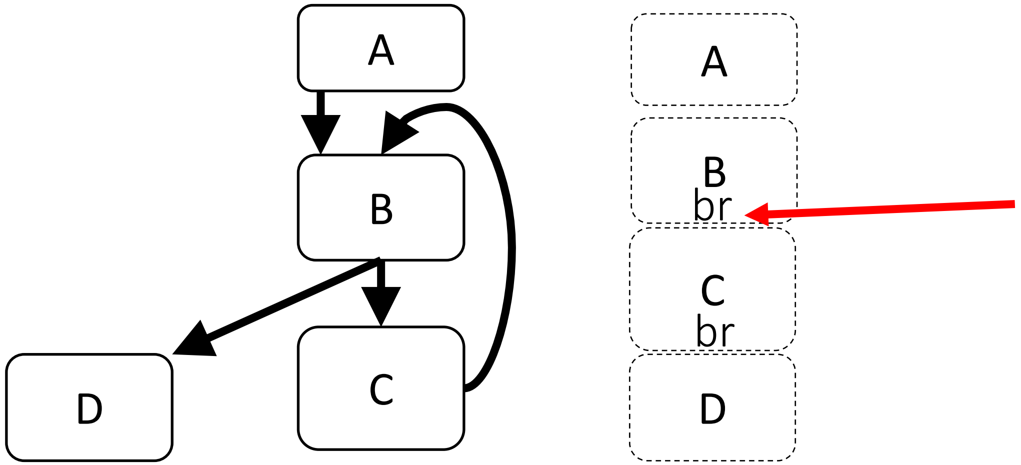
From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3



From CFG to a sequence of instructions

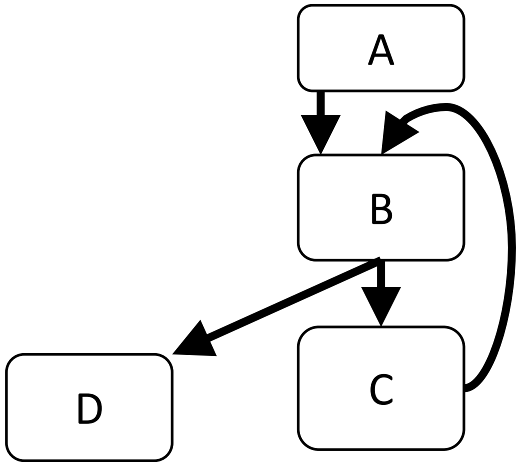
- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3



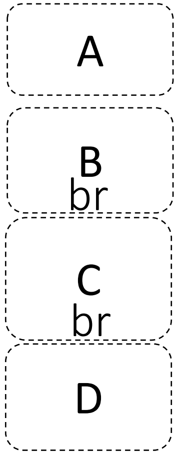

```
...
br %c :LABEL_D
:LABEL_C
...
:LABEL_D
...
```

From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3



2



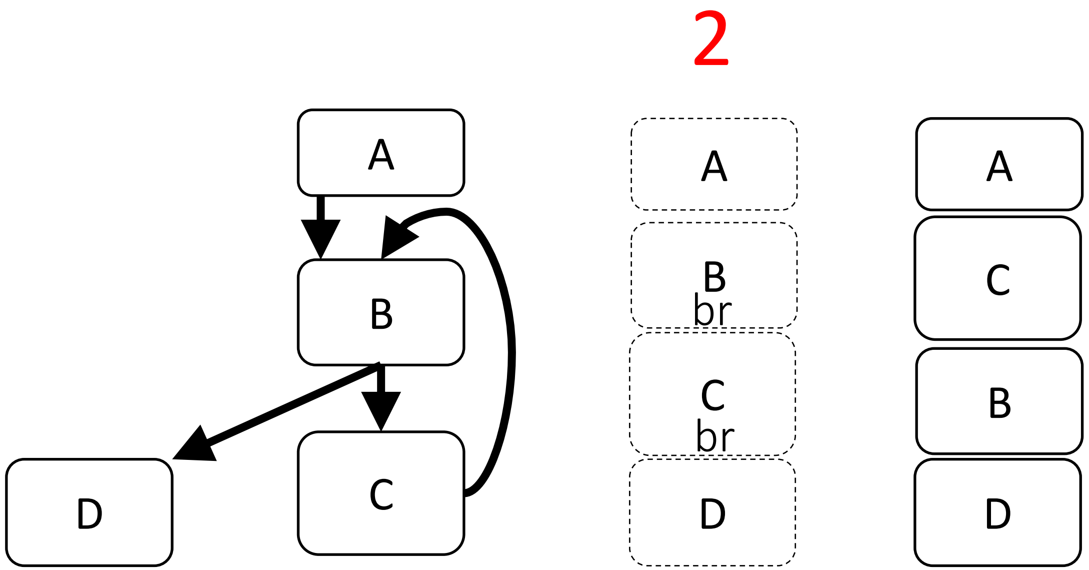
Let us assume the loop B-C is executed many times.

How many branches do we execute per loop iteration?

Is this the best we can do?

From CFG to a sequence of instructions

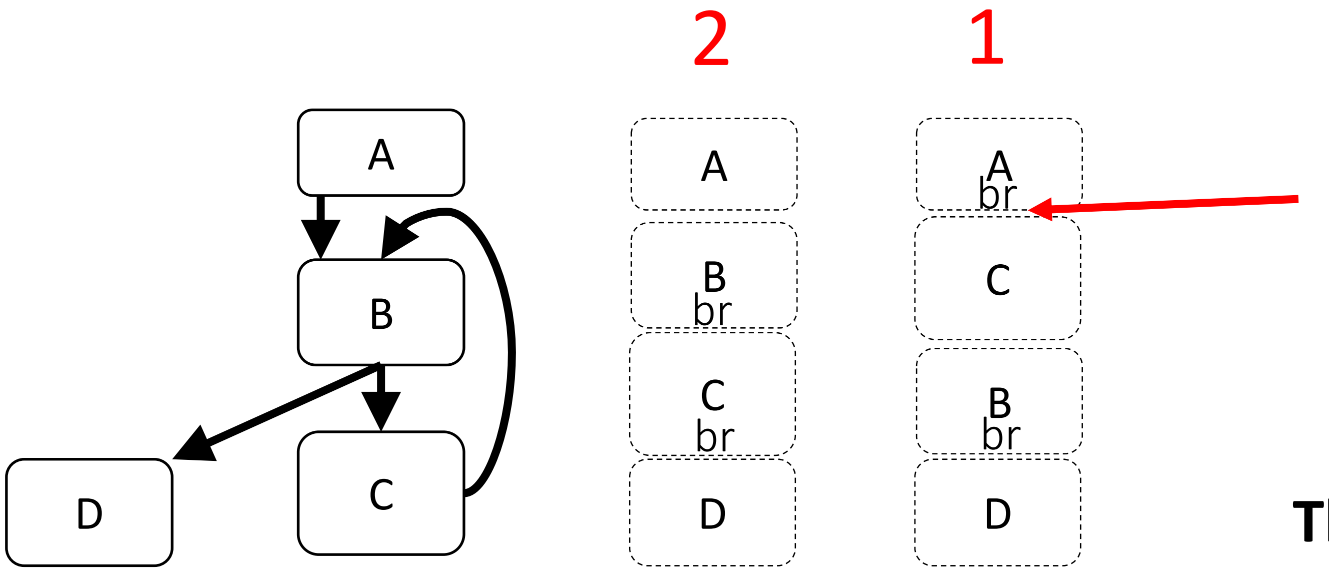
- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3



From CFG to a sequence of instructions

- CFG is a 2-dimension representation
- L3 is a 1-dimension representation
- We need to linearize CFG to generate L3

How many branches do we execute per loop iteration?



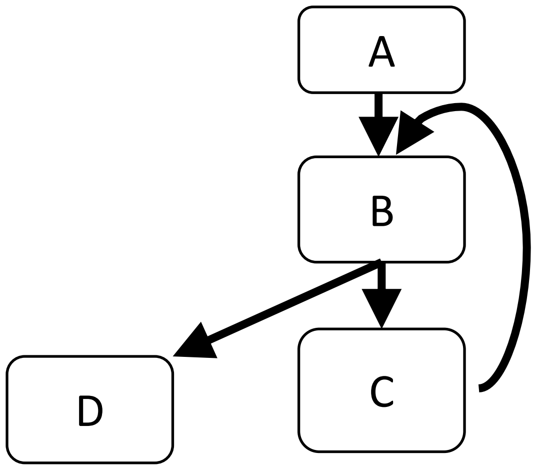
The tracing problem

CFG linearization

- A trace is a sequence of basic blocks (instructions) that could be executed at run time
  - It can include conditional branches
- A program has many overlapping traces
- For our goal:
  - Find a set of traces that cover the whole function without any overlapping
    - Each basic block belongs to exactly 1 trace
  - Remove unconditional branches within the same trace

Finding the not overlapping traces

```
list <- all basic blocks
do{
      tr  = new trace()
      bb = fetch_and_remove(list)
      while (bb is not marked){
            mark bb
            tr.append(bb)
            succs = successors(bb)
            if there is c ∈ succs such that c is unmarked and profitable(bb, c)
                bb = c
      }
} while (list is not empty)
```



Outline

- IR
- Linearize the CFG
- Linearize the data types

IR features

- Basic blocks and control Flow Graph (CFG)

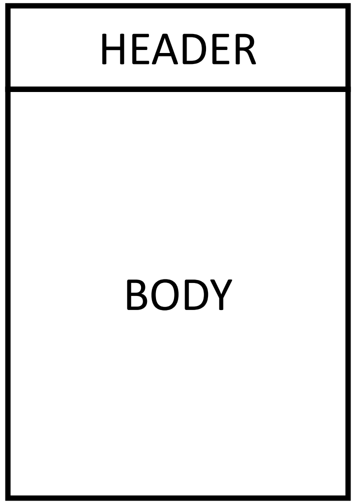
- Data types
    - Multi dimension arrays

```
define int64 @myF (int64 %p1){
  :myLabel
  int64 %p2
  %p2 <- %p1 + 1
  return %p2
}
```

Multi-dimension arrays

- The IR compiler must linearize all arrays
  - Data layout
  - Body

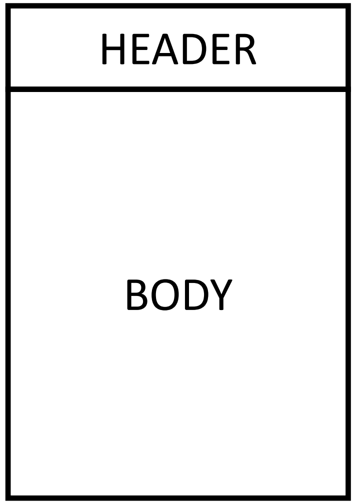
- The IR compiler must store the dimension lengths
  - Data layout
  - Header




```
int64[][] %m
int64 %e
int64 %l0
int64 %l1
%m <- new Array(7,7)
%m[0][0] <- 3
%m[2][1] <- 7
%e <- %m[0][0]
call print(%e)
%l0 <- length %m 0
%l1 <- length %m 1
```

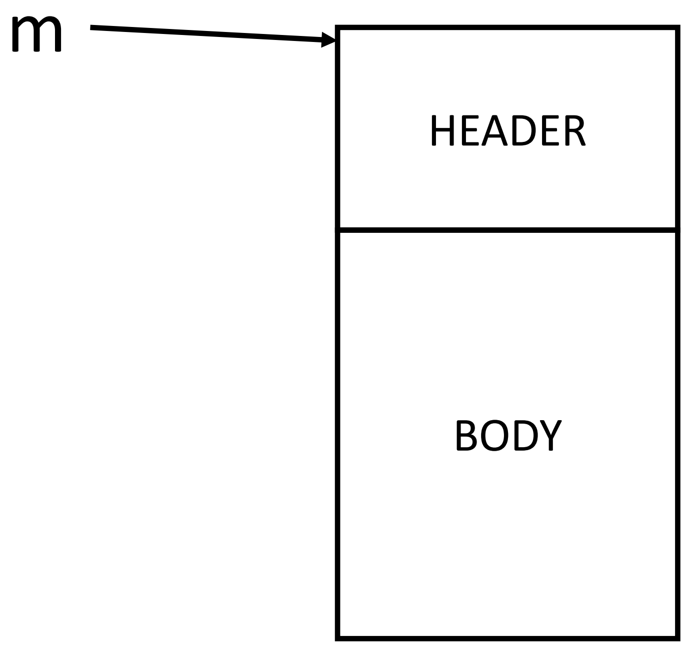
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



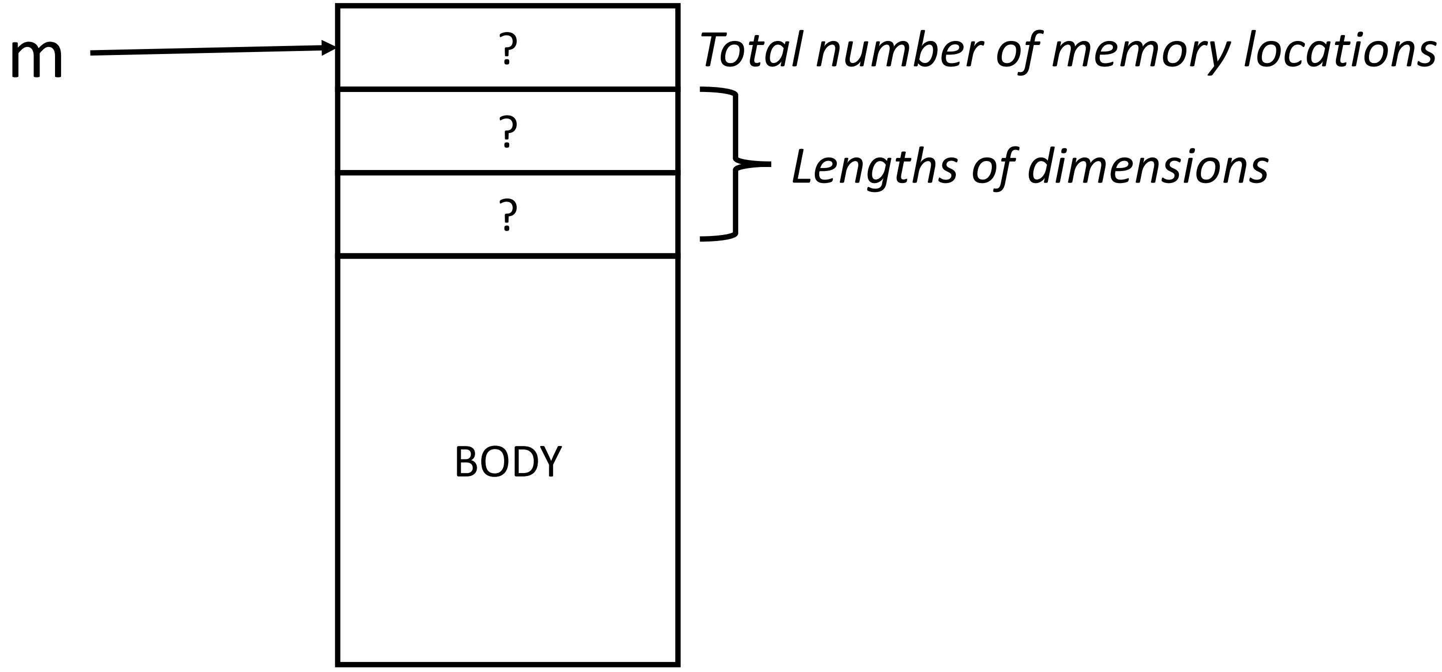
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



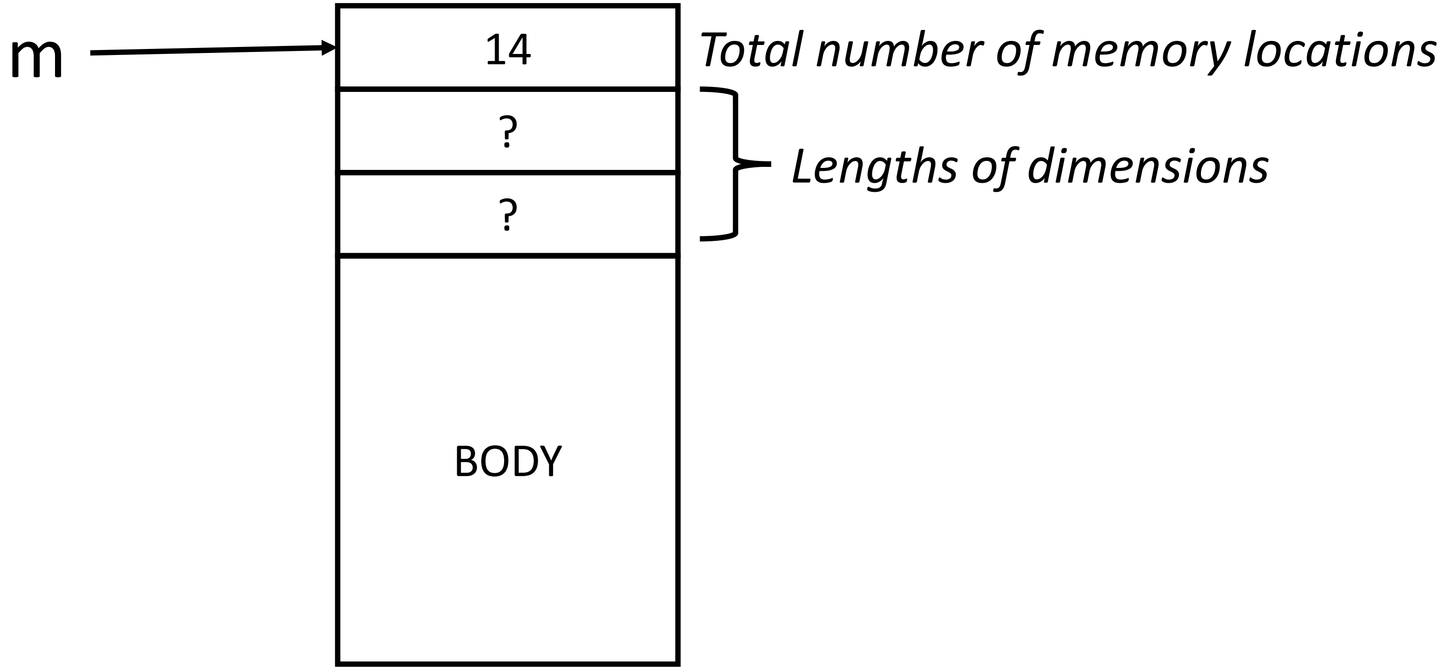
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



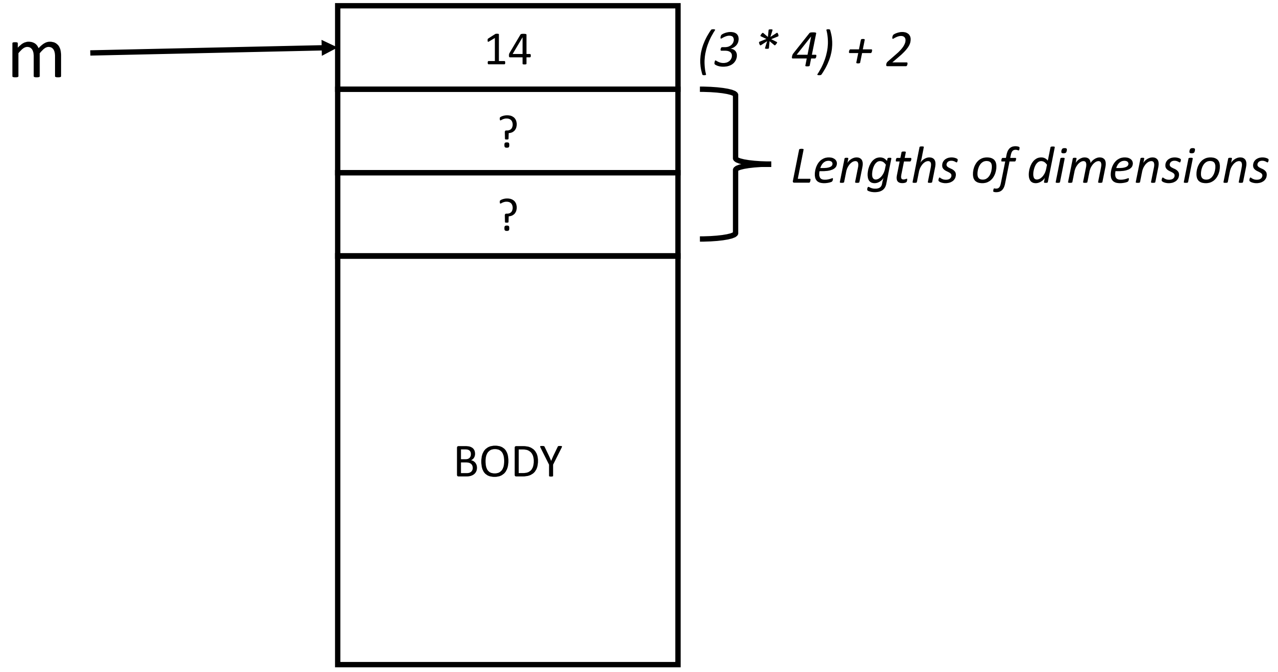
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



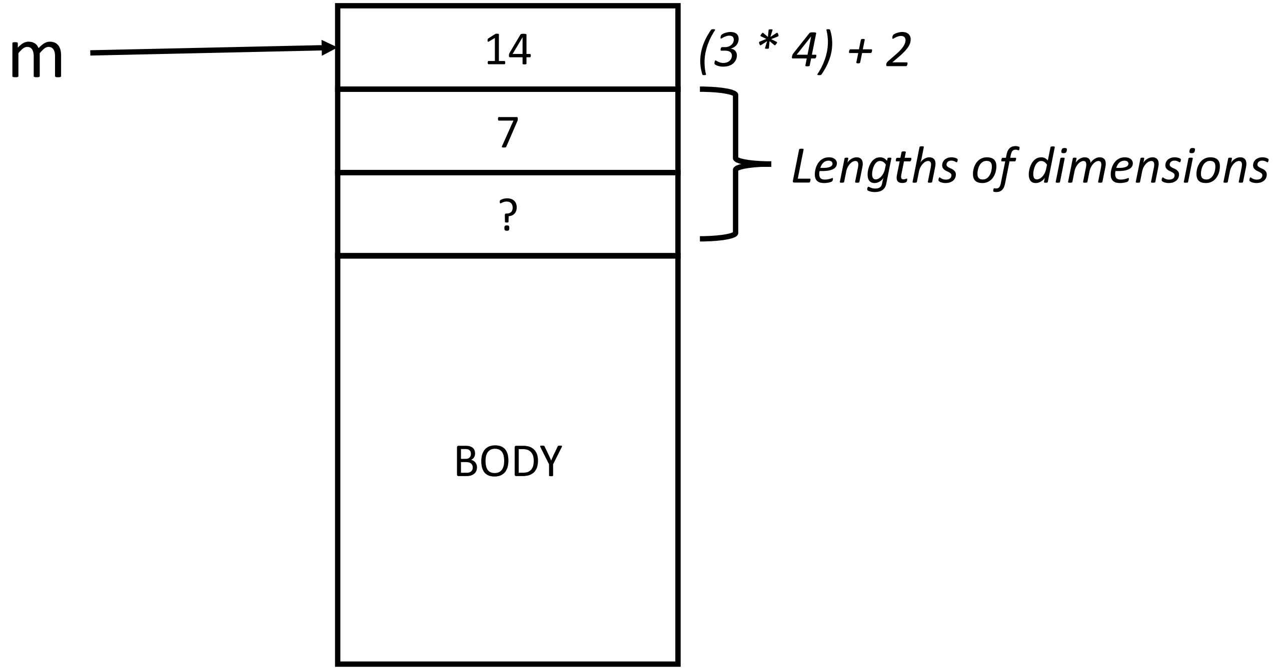
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



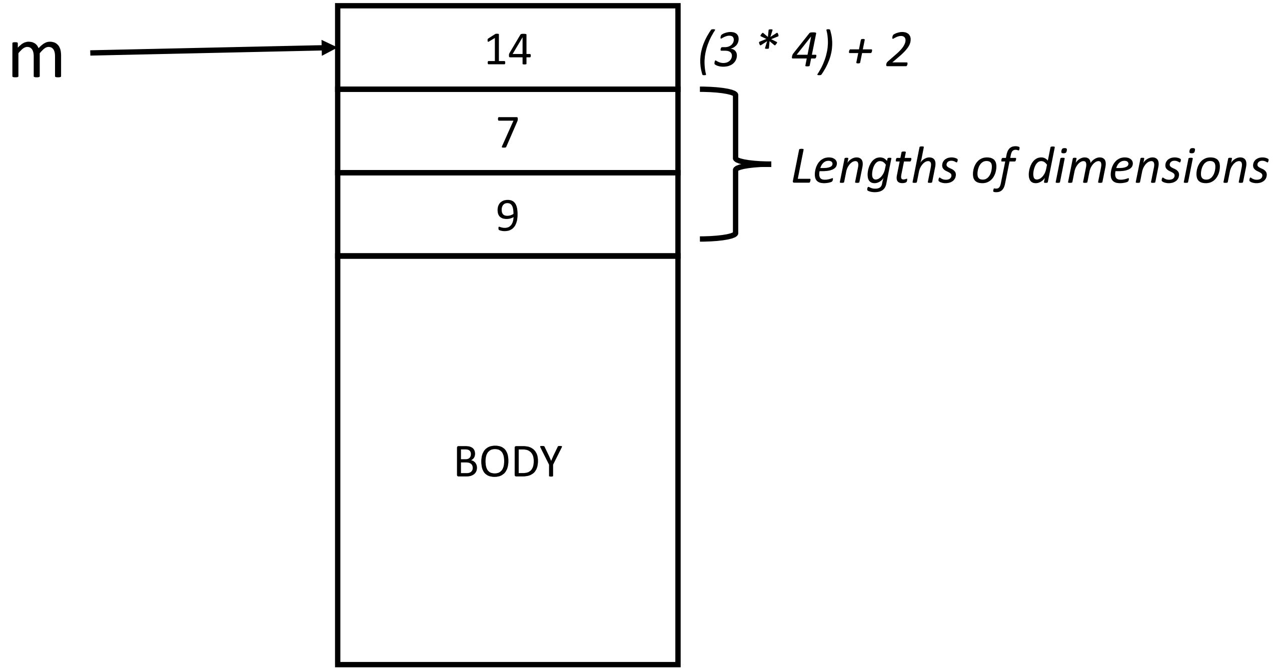
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



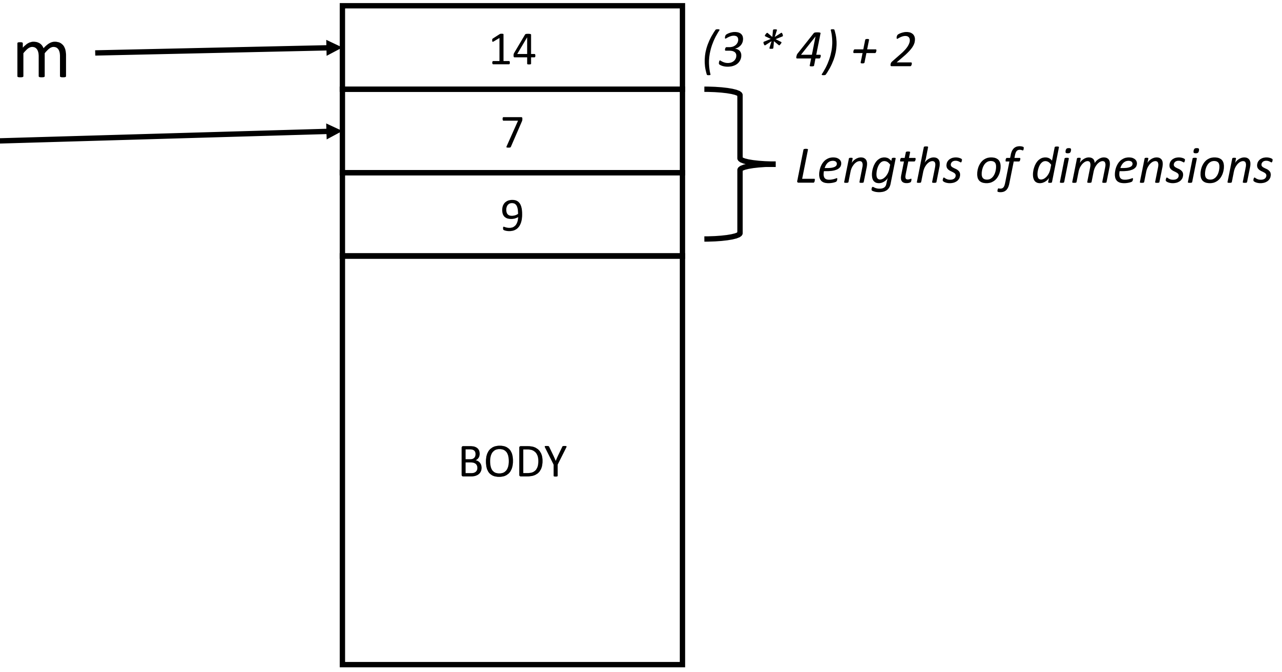
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)
```



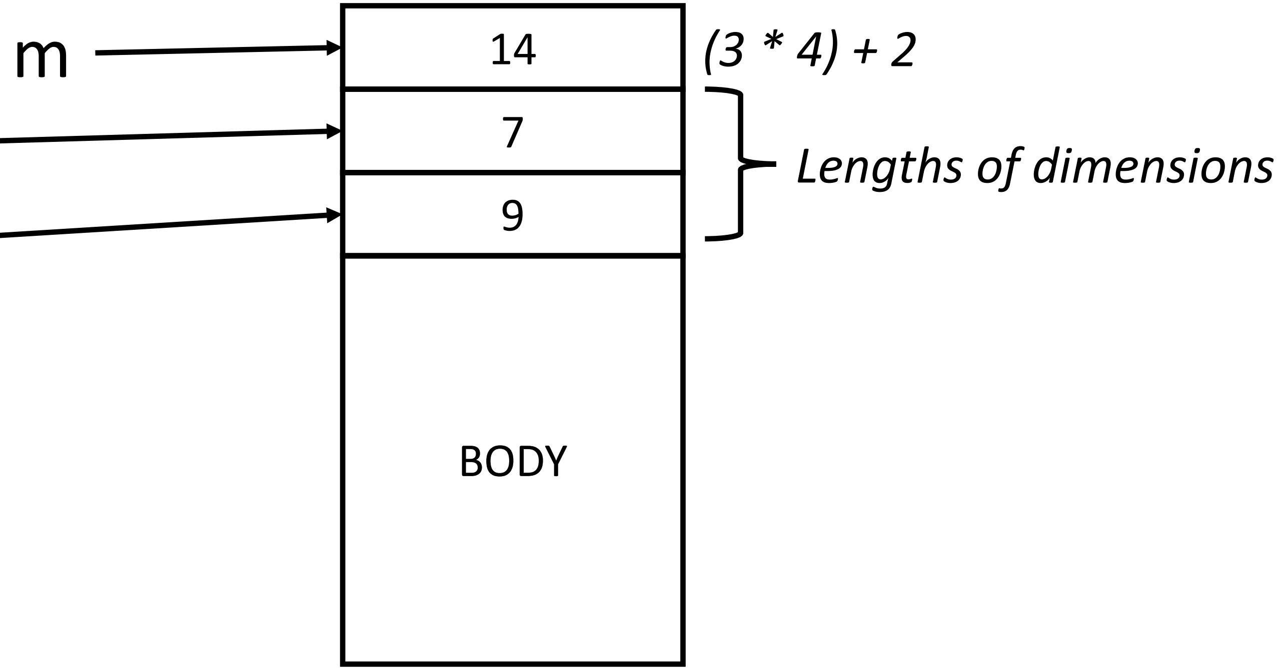
Data layout for multi-dimension arrays

```
int64[][] %m

%m <- new Array(7,9)

... <- length %m 0
```

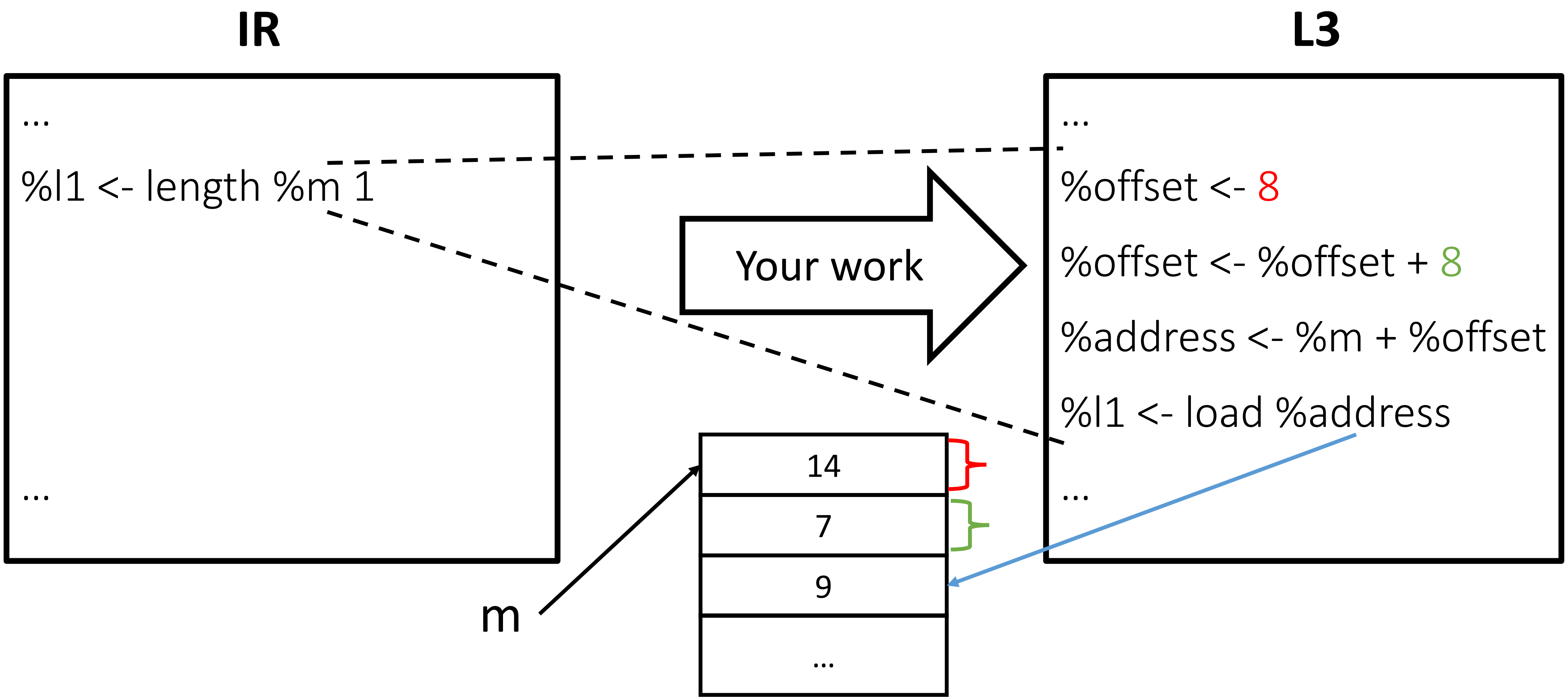


Data layout for multi-dimension arrays

```
int64[][] %m
%m <- new Array(7,9)
… <- length %m 0
… <- length %m 1
```



Translating length for arrays



Translating new Array()

```
...
Int64[][] %a
%a <- new Array(%p1,%p2)
...
```

Your work

```
...
%p1D <- %p1 >> 1
%p2D <- %p2 >> 1
%v0 <- %p1D * %p2D
%v0 <- %v0 + 2
%v0 <- %v0 << 1
%v0 <- %v0 + 1
%a <- call allocate(%v0, 1)
%v2 <- %a + 8
store %v2 <- %p1
%v3 <- %a + 16
store %v3 <- %p2
...
```

Computing the number of memory locations (the first four lines, %p1D through %v0 + 2)

Why 2? (points to `%v0 <- %v0 + 2`)

Encoding (the `<< 1` and `+ 1` lines)

arrayLength
%p1
%p2
…

Linearize an array

- m[0][0]

  %o1 <- 8

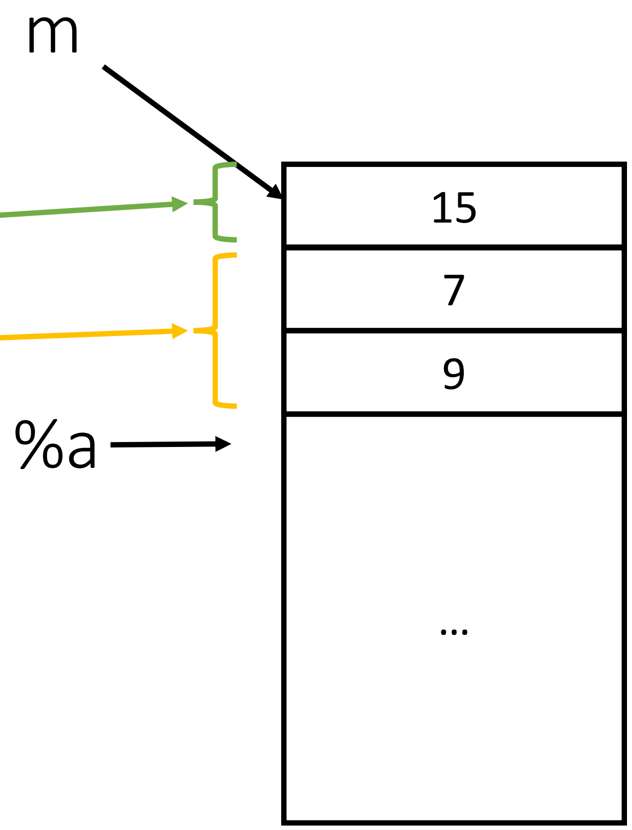
  %o2 <- 2 * 8

  %o <- %o1 + %o2

  %a <- %m + %o

  store %a <- …

```
int64[][] %m

%m <- new Array(7,9)

%m[0][0] <- …
```



- m[0][1]?
  - By row, by column

Data layout for this class

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3

0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3

- Matrix M x N
  - Offset for all: B = 8 + (2 * 8)
  - Offset A[0][1] = B + ((1)) * 8
  - Offset A[0][2] = B + ((2)) * 8
  - Offset A[0][i] = B + ((i)) * 8
  - Offset A[1][0] = B + ((1 * N) + 0) * 8
  - Offset A[i][j] = B + ((i * N) + j) * 8
- Tensor L x M x N: B = 8 + (3 * 8)
  - Offset A[k][i][j] = B + ((k * M * N) + (i * N) + j) * 8

Linearization example (2)

- IR: L x M x N: %A[%k][%i][%j] <- 5
- L3: Offset = 8 + (3 * 8) + ((k * M * N) + (i * N) + j) * 8
  - ADDR_M <- A + 16
  - M_ <- load ADDR_M
  - M <- M_ >> 1

    (To fetch M)
  - ADDR_N <- A + 24
  - N_ <- load ADDR_N
  - N <- N_ >> 1

    (To fetch N)
  - newVar1 <- i * N // newVar1 <- (i * N)
  - M_N <- M * N
  - newVar2 <- k * M_N // newVar2 <- (k * M * N)
  - newVar3 <- newVar2 + newVar1 // newVar3 <- (k * M * N) + (i * N)
  - index <- newVar3 + j // index <- (k * M * N) + (i * N) + j
  - offsetInBody <- index * 8
  - offset <- offsetInBody + 32 // 8 + (3 * 8)
  - addr <- A + offset
  - store addr <- 5

Multi-dimension arrays

- No limit to the number of dimensions

```
int64[][] %m
%m <- new Array(7,9)
Int64[][][][][][][][][][] %crazy
%crazy <- new Array(7,7,7,7,7,7,7,7,7,7)
```

- The data layout follows the scheme of the previous slides

IR features

- Basic blocks and control Flow Graph (CFG)

- Data types
    - Multi dimension arrays
    - Tuples

Tuples

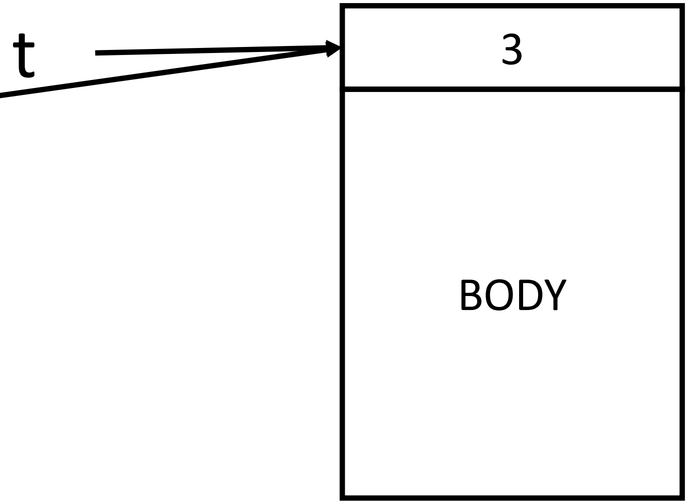
- An IR tuple is an heterogeneous 1-dimension array

- So, an IR tuple is equivalent to an L3 array

```
tuple %t

%t <- new Tuple(7)

%t[0] <- 5

int64[] %a

%a1 <- new Array(3)

%t[1] <- %a1

%a2 <- new Array(5,3)

%t[2] <- %a2
```

Data layout for tuples

```
tuple %t
%t <- new Tuple(7)
… <- length %t
```



Length is not encoded (like L3 array)

- Returned length is encoded
- The translation of “length” needs to encode the value loaded from memory

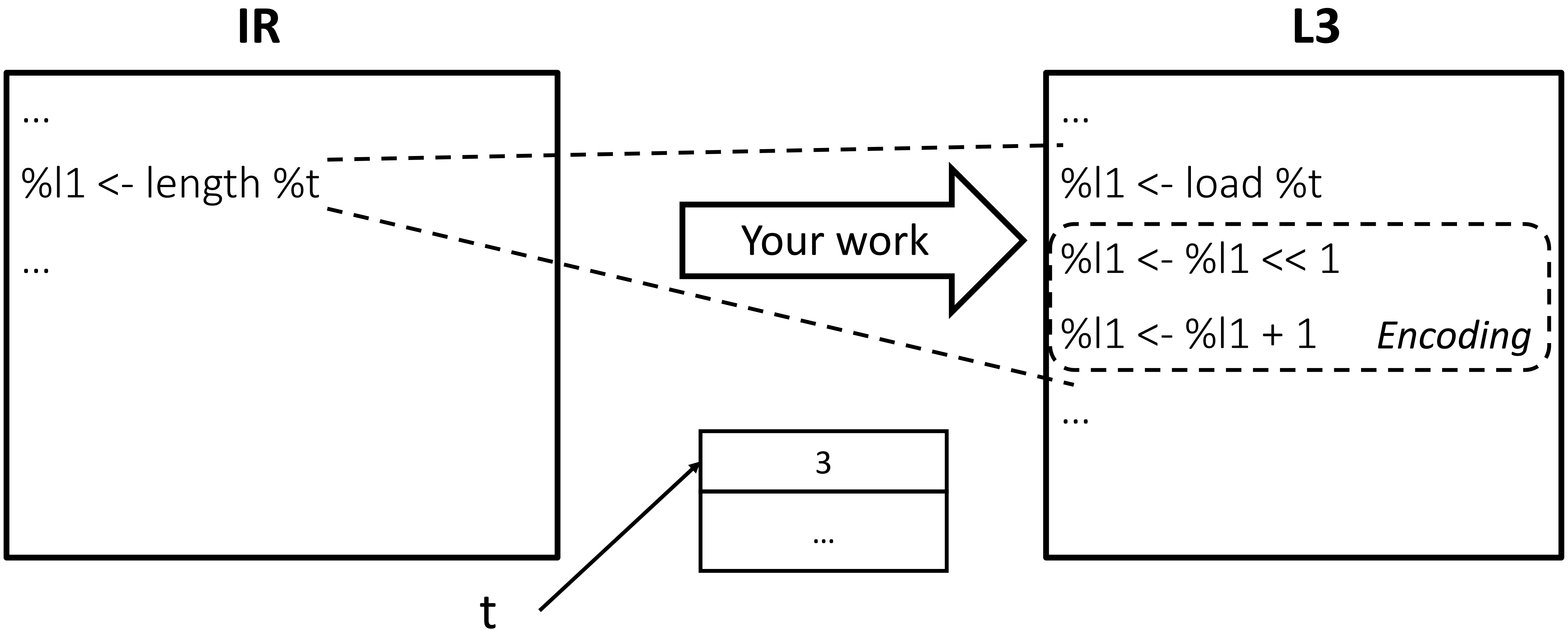
Translating tuples

```
…
tuple %t
%t <- new Tuple(7)
%t[0] <- 5
%v <- %t[0]
…
```

Your work

```
…
%t <- call allocate(7, 1)
%newVar0 <- %t + 8
store %newVar0 <- 5
%newVar1 <- %t + 8
%v <- load %newVar1
…
```

Translating length for tuples



IR features

- Basic blocks and control Flow Graph (CFG)

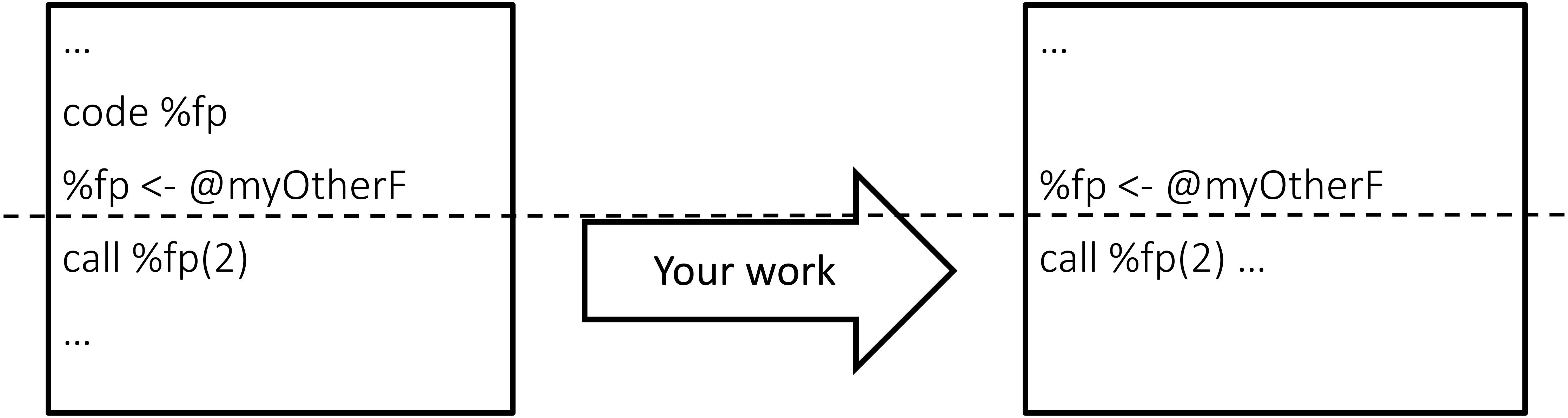
- Data types
  - Multi dimension arrays
  - Tuples
  - Function pointers

Function pointers

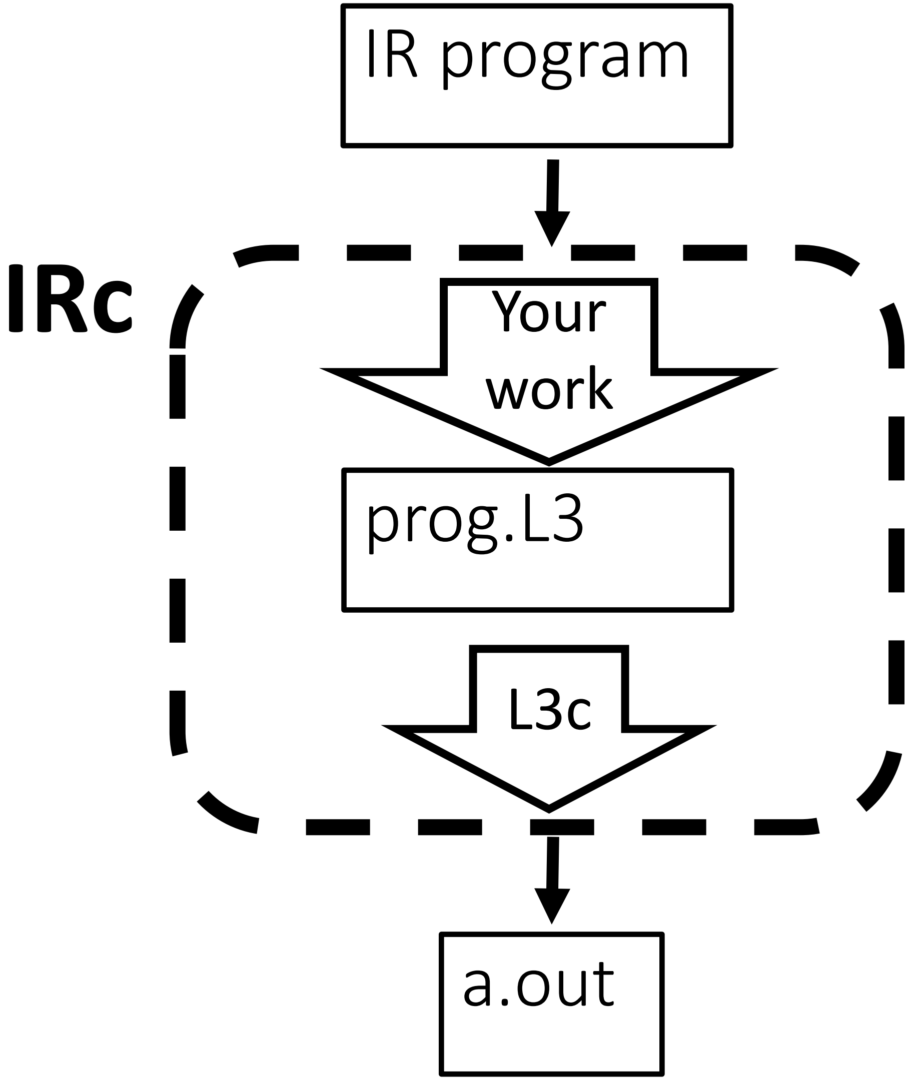
- Instances of type “code”
- Variables of type code can only store function names
- They can be used in call instructions
- They are normal variables
- They can be stored in tuples

```
define code @myF (tuple %t){
  code %fp
  %fp <- @myOtherF
  call %fp (%firstArg,2)
  %t[0] <- %fp
  return %fp
}
```

Translating function pointers



Homework #5: the IR compiler (IRc)



- To build IRc:
  translate an IR program
  to an equivalent L3

- We need to linearize the arrays

- We need to translate
  the other IR instructions

Always have faith in your ability

Success will come your way eventually

Best of luck!