Outline

• IR

• Linearize the CFG: tracing

• Linearize the data types: data layout
A compiler

High level programming language

Front-end

IR

Middle-end

IR

IR

Back-end

Today: tracing and data layout

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.
define @main (){  
  %myRes <- call @myF(5)  
  %v1 <- %myRes * 4  
  %v2 <- %myRes + %v1  
  return  
}
define @myF (%p1){  
  %p2 <- %p1 + 1  
  return %p2  
}
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
  • 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
p ::= f+
f ::= define | ( 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 | tuple-error | 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

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

\[
\begin{align*}
\text{define int64 } & @\text{myF } (\text{int64 } %p1)\{ \\
& :\text{myLabel} \\
& \text{int64 } %v1 \\
& %v1 \leftarrow 1 \\
& \text{int64}[][ ] \%\text{myMatrix} \\
& \text{int64}[][][][][][][][][][][][] %\text{myTensor} \\
& \text{tuple } %\text{myHeterogeneousArray} \\
& \text{code } %\text{myFunctionPointer} \\
& %\text{myFunctionPointer} \leftarrow @\text{myF} \\
& \text{return } %v1
\end{align*}
\]
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

\[
\begin{align*}
p & ::= f^+ \\
f & ::= \text{define } T \mid (\text{pars}) \{ \text{bb}^+ \} \\
\text{bb} & ::= \text{label } i \ast \text{te} \\
\text{te} & ::= \text{br label} \mid \text{br t label label} \mid \text{return} \mid \text{return } t \\
i & ::= \text{type var} \mid \text{var } \leftarrow s \mid \text{var } \leftarrow t \text{ op } t \\
& \quad \text{var } \leftarrow \text{var}([t])^+ \mid \text{var}([t])^+ \leftarrow s \mid \text{var } \leftarrow \text{length var } t \mid \text{var } \leftarrow \text{length var} \\
& \quad \text{call callee (args?)} \mid \text{var } \leftarrow \text{call callee (args?)} \\
& \quad \text{var } \leftarrow \text{new Array(args)} \mid \text{var } \leftarrow \text{new Tuple(t)} \\
T & ::= \text{type} \mid \text{void} \\
\text{type} & ::= \text{int64([])}^* \mid \text{tuple} \mid \text{code} \\
\text{callee} & ::= u \mid \text{print} \mid \text{input} \mid \text{tuple-error} \mid \text{tensor-error} \\
\text{pars} & ::= \text{type var} \mid \text{type var (, type var)*} \\
\text{args} & ::= t \mid t (, t)^* \\
\text{s} & ::= t \mid l \\
\text{t} & ::= \text{var} \mid N \\
\text{u} & ::= \text{var} \mid l \\
N & ::= (+|-)? [0-9]^+ \\
\text{op} & ::= + \mid - \mid * \mid \& \mid << \mid >> \mid < \mid \leq \mid = \mid \geq \mid > \\
l & ::= @\text{name} \\
\text{label} & ::= :\text{name} \\
\text{var} & ::= %\text{name} \\
\text{name} & ::= \text{sequence of chars matching [a-zA-Z][a-zA-Z_0-9]*}
\end{align*}
\]
Control Flow Graph (CFG)

- A CFG is a graph $G = \langle \text{Nodes}, \text{Edges} \rangle$
- Nodes: Basic blocks
- Edges: $(x,y) \in \text{Edges}$ 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
define int64 @myF (int64 %p1)
{
    :myLabel
    int64 %c
    %c <- %p1 >= 3
    br %c :true :false

    :true
    return 1
    :false
    return 0
}

p ::= f+ 
f ::= define T ! (pars) { bb+ } 
bb ::= label l* 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 | tuple-error | tensor-error 
pars ::= type var | type var (, type var)* |
args ::= t | t (, t)* 
s ::= t | 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]*
IR makes the CFG explicit

IR also makes complex data types explicit
• Explicit data types
• Explicit allocations
IR

```
p ::= f*  
f ::= define T | (pars) { bb* }  
bb ::= label i* te  
te ::= br label | br t label 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 | tuple-error | tensor-error

pars ::= type var | type var (, type var)* |  
args ::= t | t (, t)*

s ::= t | 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
  %v[1] <- 1
  return 0
}
```
\[
\text{define int64 @myF (int64 %p1)}\
\begin{array}{l}
:\text{myLabel} \\
\text{int64[][] %v} \\
\text{%v <- new Array(7,7)} \\
\text{%v[0][3] <- 1} \\
\text{return 0}
\end{array}
\]
define int64 @myF (int64 %p1) {
    :myLabel
    int64[][] %v
    %v <- new Array(7,7,7)
    %v[0][1][3] <- 1
    return 0
}
Implicit initialization to “1”
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

```
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

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
\[ p ::= f^+ \]
\[ f ::= \text{define } T \mid (\text{pars}) \{ \text{bb}^+ \} \]
\[ \text{bb} ::= \text{label } \ast \text{ te} \]
\[ \text{te} ::= \text{br } \text{ label } \mid \text{br } \text{ t } \text{ label } \text{ label } \mid \text{return } \mid \text{return } t \]
\[ i ::= \text{type } \text{ var } \mid \text{var } \leftarrow s \mid \text{var } \leftarrow \text{t op } t \mid \]
\[ \text{var } \leftarrow \text{var}([t])^* \mid \text{var}([t])^* \leftarrow s \mid \text{var } \leftarrow \text{length } \text{ var } t \mid \text{var } \leftarrow \text{length } \text{ var } \mid \]
\[ \text{call callee (args?) } \mid \text{var } \leftarrow \text{call callee (args?) } \mid \]
\[ \text{var } \leftarrow \text{new } \text{Array(} \text{args} \text{)} \mid \text{var } \leftarrow \text{new } \text{Tuple(t)} \]
\[ T ::= \text{type } \mid \text{void} \]
\[ \text{type} ::= \text{int64}([\text{*}])^* \mid \text{tuple } \mid \text{code} \]
\[ \text{callee} ::= \text{u } \mid \text{print } \mid \text{input } \mid \text{tuple-error } \mid \text{tensor-error} \]
\[ \text{pars} ::= \text{type } \text{ var } \mid \text{type var} (, \text{type var})^* \mid \]
\[ \text{args} ::= t \mid t (, t)^* \]
\[ s ::= t \mid l \]
\[ t ::= \text{var } \mid N \]
\[ u ::= \text{var } \mid l \]
\[ N ::= (\text{+ } \text{-})? [0-9]^+ \]
\[ \text{op} ::= + \mid - \mid * \mid \& \mid \&\& \mid >> \mid >> = \mid <= \mid = \mid >= \mid > \]
\[ l ::= @\text{name} \]
\[ \text{label} ::= :\text{name} \]
\[ \text{var} ::= %\text{name} \]
\[ \text{name} ::= \text{sequence of chars matching [a-zA-Z_]a-zA-Z_0-9}^* \]
Indices of array and tuple accesses are not encoded

```plaintext
int64[] %vec
int64 %e
%vec <- new Array(7)
%vec[0] <- 3
%vec[2] <- 7
%e <- %vec[0]
call print(%e)
%l <- length %vec 0
```
Indices and dimension# in length are not encoded

- Accessing length of a dimension
  \( %l \gets \text{length} \ %\text{ar} \ %\text{dimID} \)

- Accessing length of a tuple
  \( %l \gets \text{length} \ %\text{tuplePtr} \)

- Accessing array element
  \( \%\text{ar}[%e1][%e2] \gets %v1 \)
  \( %v2 \gets \%\text{ar}[%e1][%e2] \)

- Allocating an array
  \( \%\text{ar} \gets \text{new Array}(\%\text{dim1}, \%\text{dim2}) \)
Variable definition

• The code must define (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.
Final notes on IR

Same undefined behaviors as for L3
but without the one related to the last instruction of a function

L3

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

IR

```assembly
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: tracing
- Linearize the data types: data layout
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

\[
\text{br \%cond :TRUE :FALSE}
\] 

\[
\text{br :FALSE}
\] 

Your work
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
From CFG to a sequence of instructions

No jump needed
From CFG to a sequence of instructions

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

Diagram:
- A -> B -> C -> D
- B -> C -> br
- A -> br

```
From CFG to a sequence of instructions

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
From CFG to a sequence of instructions

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)
        bb = select_next(list, bb, succs)
    }
} while (list is not empty)
Outline

• IR

• Linearize the CFG: tracing

• Linearize the data types: data layout
IR features

• Basic blocks and control Flow Graph (CFG)

• Data types
  • Multi dimension arrays

```c
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

```
 HEADER

 BODY

 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

```plaintext
int64[][] %m
%m <- new Array(7,9)
```
Data layout for multi-dimension arrays

```plaintext
int64[][] %m
%m <- new Array(7,9)
```
Data layout for multi-dimension arrays

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

```
? ? ?
????

BODY
```

- **Total number of memory locations**
- **Lengths of dimensions**
Data layout for multi-dimension arrays

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

Total number of memory locations

Lengths of dimensions
Data layout for multi-dimension arrays

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

![Diagram showing data layout for multi-dimension arrays]

- **Lengths of dimensions**
  - BODY
  - m
  - (3 * 4) + 2
  - ?
  - ?
  - ?

```plaintext
14
```
Data layout for multi-dimension arrays

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

```
14
7
?
BODY
(3 * 4) + 2
```

Lengths of dimensions
Data layout for multi-dimension arrays

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

```
(3 * 4) + 2

Lengths of dimensions

BODY
```

$m$
Data layout for multi-dimension arrays

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

Lengths of dimensions

```
BODY
  14
  7
  9
```

\[(3 \times 4) + 2\]
Data layout for multi-dimension arrays

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

```
BODY
```

Lengths of dimensions

\[(3 \times 4) + 2\]
Translating length for arrays

IR

... %l1 <- length %m 1

L3

... %offset <- 8
%offset <- %offset + 8
%address <- %m + %offset
%l1 <- load %address

Your work

m

14
7
9
...
Translating new Array()

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

Computing the number of memory locations

Your work

arrayLength
%p1
%p2
...

Encoding

%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
...

Why 2?
Linearize an array

• m[0][0]
  %o1 ← 8
  %o2 ← 2 * 8
  %o ← %o1 + %o2
  %a ← %m + %o
  store %a ← ...

• m[0][1]?
  • By row, by column

int64[][] %m
%m ← new Array(7,9)
%m[0][0] ← ...
Data layout for this class

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
</tbody>
</table>

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

- **IR:** $L \times M \times N$: \%A[\%k][\%i][\%j] <- 5

- **L3:** Offset = $8 + (3 \times 8) + (k \times M \times N) + (i \times N) + j$ * 8

  - ADDR_M <- A + 16
  - M_ <- load ADDR_M
  - M <- M_ >> 1
  - ADDR_N <- A + 24
  - N_ <- load ADDR_N
  - N <- N_ >> 1
  - newVar1 <- i * N
  - M_N <- M * N
  - newVar2 <- k * M_N
  - newVar3 <- newVar2 + newVar1
  - index <- newVar3 + j
  - offsetInBody <- index * 8
  - offset <- offsetInBody + 32
  - addr <- A + offset
  - store addr <- 5

To fetch M

To fetch N

// newVar1 <- (i * N)
// newVar2 <- (k * M * N)
// newVar3 <- (k * M * N) + (i * N)
// index <- (k * M * N) + (i * N) + j
// 8 + (3 * 8)
Multi-dimension arrays

• No limit to the number of dimensions

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

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

Length is not encoded (like L3 array)
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
...

68
Translating length for tuples

IR

... %l1 <- length %t
...

L3

... %l1 <- load %t
%l1 <- %l1 << 1
%l1 <- %l1 + 1
Encoding
...

Your work
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

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

code %fp
%fp <- @myOtherF
call %fp(2)

Your work

%fp <- @myOtherF
call %fp(2) ...
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!