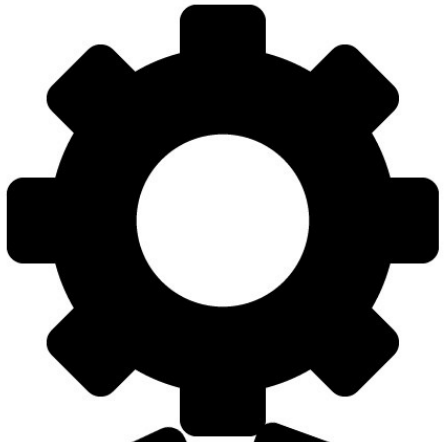
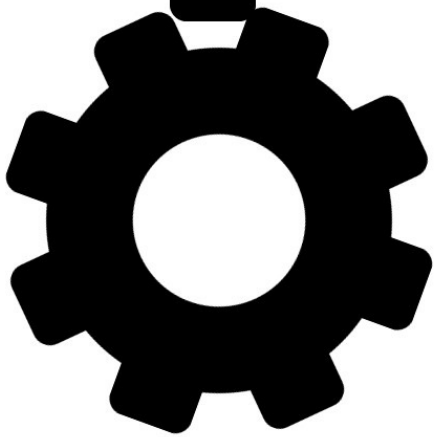
C  mpiler

C  nstruction

LA

Simone Campanoni
simone.campanoni@northwestern.edu

A compiler

High level programming language

Front-end

- ...
- Values encoding
- Checking memory accesses
- Basic block generation

IR

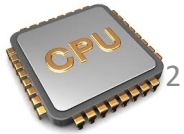
Middle-end

IR

Back-end

- Explicit control flow and data types
- Legalizer
- Instruction selection
- Register allocation
- Peephole optimizations
- Instruction scheduling

Machine code



Outline

- LA
- Encoding values
- Checking array accesses
- Generating basic blocks and variable declarations

LA

```
p      ::= f+
f      ::=      T name ( pars ) { i* }
i      ::= type name | name <- t | name <- t op t |
           label | br label  | br t label label | return | return t
           name <- name([t])+ |  name([t])+ <- t |
           name <- length name t? |
           name( args? ) | name <- name( args? ) |
           name <- new Array(args) | name <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
args   ::= t | t (, t)*
t      ::= name | N
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
pars   ::= type var | type var (, type var)* |
name   ::= [a-zA-Z_][a-zA-Z_0-9]*
label  ::= :name
```

You cannot have invalid memory accesses in the underlying architecture

All values are decoded!

```
void   main      (){
   int64 myRes
   int64 v1
   int64 v2
   myRes <- myF(2)
   v1 <- myRes * 3
   v2 <- myRes + v1
   return v2
}
int64   myF     (int64 p1){
  int64 p1
  int64 p2
  p2 <- p1 + 1
  return p2
}
```

LA standard library

- int64 input (void)
- void print (int64)
- void print (int64[])
- void print (int64[][])
- …

LA standard library

- int64 input (void)
- void print (int64([])*)
- void print (tuple)

No tensor-error
No tuple-error

IR

```
define int64 @test (){
  :entry
  int64 %myRes
  int64 %v1
  int64 %v2
  %myRes <- call @myF(5)
  %v1 <- %myRes * 7
  %v2 <- %myRes + %v1
  return %v2 }
define int64 @myF (int64 %p1){
  :myLabel
  int64 %p2
  %p2 <- %p1 + 3
  return %p2
}
```

LA

```
int64 test (){
  int64 myRes
  int64 v1
  int64 v2
  myRes <- myF(2)
  v1 <- myRes * 3
  v2 <- myRes + v1
  return v2
}
int64 myF (int64 p1){
  int64 p2
  p2 <- p1 + 1
  return p2
}
```

LA and invalid memory acccess

```
int64 test (){
    int64[] v
    v <- new Array(2)
    int64 s
    s <- v[100]
    return 0
}
```

LA and variable declarations

- LA variables must be declared before being used

- The scope of LA's variables is the function
  - They can be declared anywhere in the function
  - There is no semantic difference between declaring a variable in the middle of the function and declaring it at the beginning of that function

Final notes

- Implicit return instruction if missing

```
int64 test (){
    int64 v
    print(v)
}
```

- LA integer variables are implicitly initialized to zero of the LA language
  - One in IR, L3, L2, L1, x86_64

- LA code variables are initialized to 0
  - Reading a 0 in a code variable is a bug

- Having return instructions implicit (if needed), all variables are implicitly initialized, and having made invalid memory accesses impossible, leaves integer overflow as the **only** undefined behavior of LA

Now that you know LA, rewrite your IR programs in LA

Outline

- LA
- Encoding values
- Checking array accesses
- Generating basic blocks and variable declarations

Value encoding

- Values in LA **are not** encoded
- Values in IR
  (beside array/tuple indices, and the 2nd parameter of length)
  are encoded
- The LA compiler needs to encode the values
  - Solution 1: (this class)
    - Everything is encoded
    - When needed, decode just before uses
  - Solution 2: (advanced)
    - Everything is decoded
    - Encode just before invoking the runtime

Value encoding example

LA

```
int64 v1
…
br v1 :L1 :L2
…
```

IR

```
in64 %v1
int64 %v1_new
%v1 <- 1
…
%v1_new <- v1 >> 1
br %v1_new :L1 :L2
…
```

Integer variables are initialized to “zero”: encoded is 1

Value encoding example 2

LA

```
Int64[] v1
…
br v1 :L1 :L2
…
```

IR

```
In64[] %v1
…
br %v1 :L1 :L2
…
```

Value encoding example 3

LA

```
int64 v1
int64 v2
int64 v3
…
v3 <- v1 + v2
…
```

IR

```
int64 %v1
int64 %v2
int64 %v3
int64 %v1_new
int64 %v2_new
…
%v1_new <- %v1 >> 1
%v2_new <- %v2 >> 1
%v3 <- %v1_new + %v2_new
%v3 <- %v3 << 1
%v3 <- %v3 + 1
…
```

Value encoding: toDecode(i)

1. toDecode(br t label label) = t
2. toDecode(var1 <- length var2 var3) = var3
3. toDecode(var1 <- var2([vari])$^{+}$) = ([vari])$^{+}$
4. toDecode(var1([vari])$^{+}$ <- s) = ([vari])$^{+}$
5. toDecode(var <- t1 op t2) = t1, t2
6. toDecode(everything else) =

Value encoding: toEncode(i)

1. toEncode(var <- t op t) = var
2. toEncode(everything else) =

Value encoding algorithm

1. Encode all constants not used for array/tuple indices and 2nd parameter of length
2. For each instruction *i*
   A. For every variable or number *t* in toDecode(i)
      I. Create a new variable *v'* and place its declaration to the first basic block
      II. Store the decoded value of *t* in *v'*
      III. Change *i* to use *v'* instead of *t*
   B. For every variable *v* in toEncode(i)
      I. Encode *v* just after *i*

Outline

- LA
- Encoding values
- Checking array accesses
- Generating basic blocks and variable declarations

LA

```
p      ::= f+
f      ::=      T name ( pars ) { i* }
i      ::= type name | name <- t | name <- t op t |
           label | br label | br t label label | return | return t
           name <- name([t])+ | name([t])+ <- t |
           name <- length name t? |
           name( args? ) | name <- name( args? ) |
           name <- new Array(args) | name <- new Tuple(t)
T      ::= type | void
type   ::= int64([])* | tuple | code
args   ::= t | t (, t)*
t      ::= name | N
N      ::= (+|-)? [0-9]+
op     ::= + | - | * | & | << | >> | < | <= | = | >= | >
pars   ::= type var | type var (, type var)* |
name   ::= [a-zA-Z_][a-zA-Z_0-9]*
label  ::= :name
```

Checking array/tensor/tuple accesses

To avoid invalid memory accesses, we need to generate code to

1. Check if an array, tensor, or tuple has been allocated before their use

2. Check if the indexes used to access an array, tensor, or tuple are within the allowed ranges

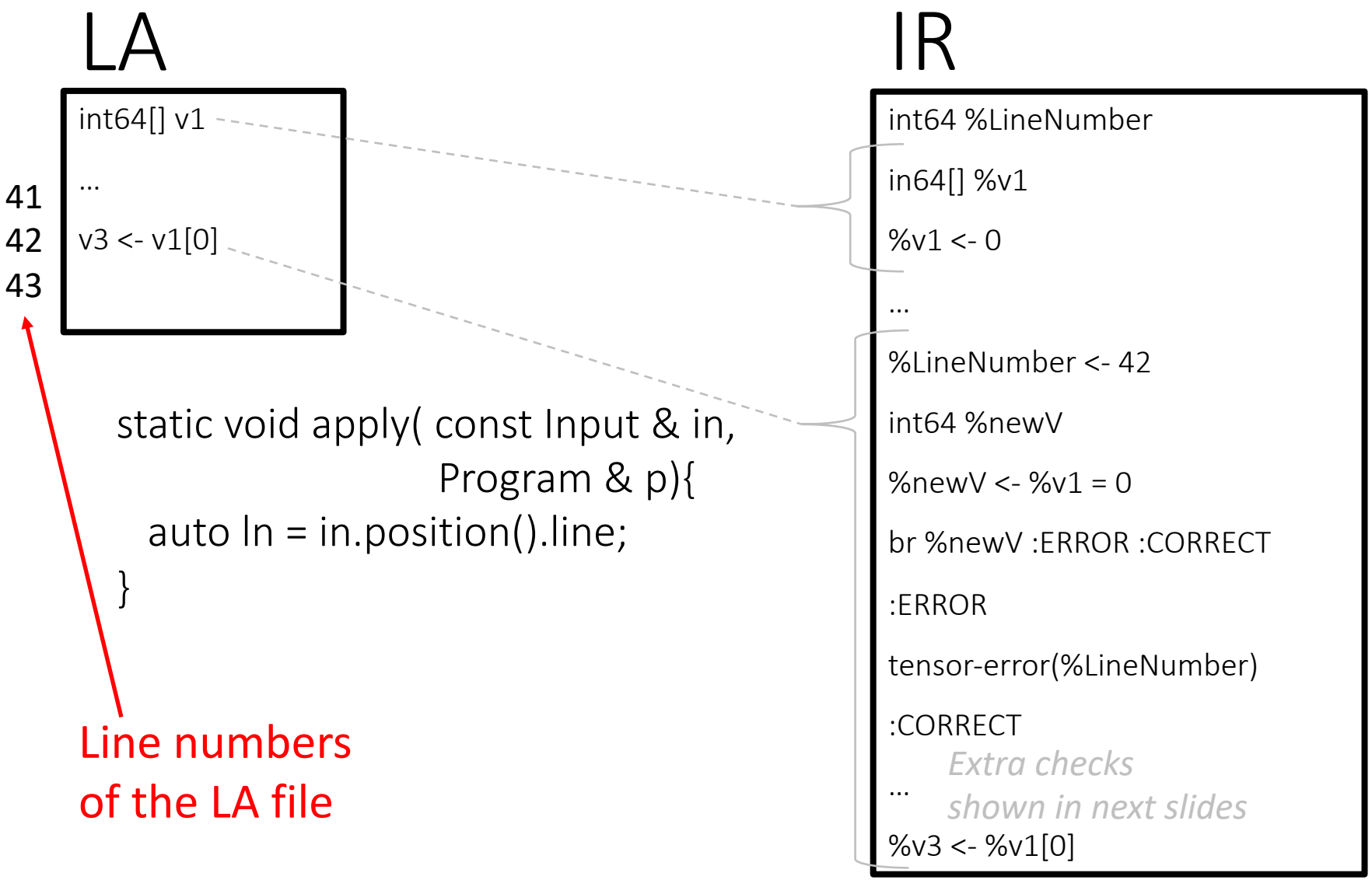
Checking array/tensor/tuple allocation

To check if an array, tensor, or tuple has been allocated
before their use

1. Initialize an array/tensor/tuple variable to 0 at their declaration

2. Before every access of an array/tensor/tuple v:
   I. Check if v is not 0
   II. If it is 0, then call tensor-error(int64 line)
   passing the line number of the LA file
   of the incorrect instruction

Checking allocation example



Checking allocation example

LA

```
41  int64[] v1
42  ...
43  v3 <- v1[0]
```

IR

```
int64 %LineNumber
in64[] %v1
%v1 <- 0
...
%LineNumber <- 42
int64 %newV
%newV <- %v1 = 0
br %newV :F :C
:F
tensor-error(%LineNumber)
:C
...
%v3 <- %v1[0]
```

Checking whether the memory has been allocated or not

Checking allocation example

LA

```
41  Int64[] v1
42  …
43  v3 <- v1[0]
```

IR

```
int64 %LineNumber
in64[] %v1
%v1 <- 0
…
%LineNumber <- 42
int64 %newV
%newV <- %v1 = 0
br %newV :F :C
:F
tensor-error(%LineNumber)
:C
…
%v3 <- %v1[0]
```

Error reporting

Checking whether the memory has been allocated or not

Checking allocation example

LA

```
41  Int64[] v1
42  …
43  v3 <- v1[0]
```

IR

```
int64 %LineNumber
in64[] %v1
%v1 <- 0
…
%LineNumber <- 42
int64 %newV
%newV <- %v1 = 0
br %newV :F :C
:F
tensor-error(%LineNumber)
:C
…
%v3 <- %v1[0]
```

Error reporting

Checking whether the memory has been allocated or not

Checking whether the memory offset is within the object allocated

Checking single-dimension array or tuple accesses

Check if the index used to access a single-dimension array are within the allowed range

To do so, for the index i used (e.g., ar[i]), we need to

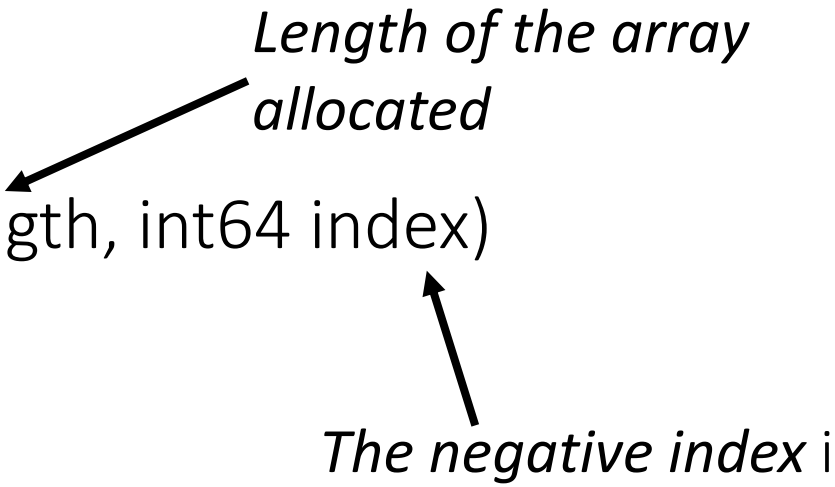
A. Check i is not negative
B. Check i is less than the length of the array

Checking single-dimension array or tuple accesses

A. Check i is not negative

1. Check if i >= 0
2. If it isn't, then call

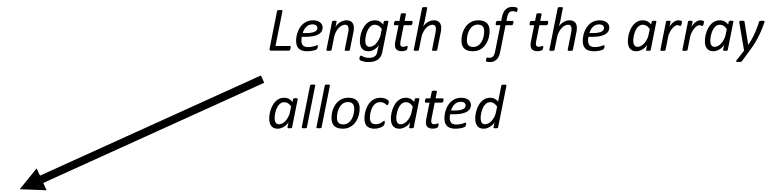
tensor-error(int64 line, int64 length, int64 index)



Checking single-dimension array or tuple accesses

B. Check i is less than the length of the array

1. Load the length of the relative dimension (e.g., l_i <- length ar 0)
2. Check if i is less than that length (e.g., i < l_i)
3. If it isn't, then call tensor-error(int64 line, int64 length, int64 index)



Checking single-dimension array or tuple accesses

Check if the index used to access a single-dimension array are within the allowed range

To do so, for the index i used (e.g., ar[i]), we need to

A. Check i is not negative
B. Check i is less than the length of the array

Checking tensor accesses

Check if the indexes used to access a tensor
are within the allowed ranges

To do so, for every index i (e.g., t[k][i][j]), we need to

A. Check i is not negative
B. Check i is less than
   the length of the dimension it refers to of the tensor

Checking tensor accesses

A. Check i is not negative
1. Check if i >= 0
2. If it isn't, then call
   tensor-error(int64 line, int64 d, int64 length, int64 index)

First dimension (from left to right) of the tensor incorrectly accessed

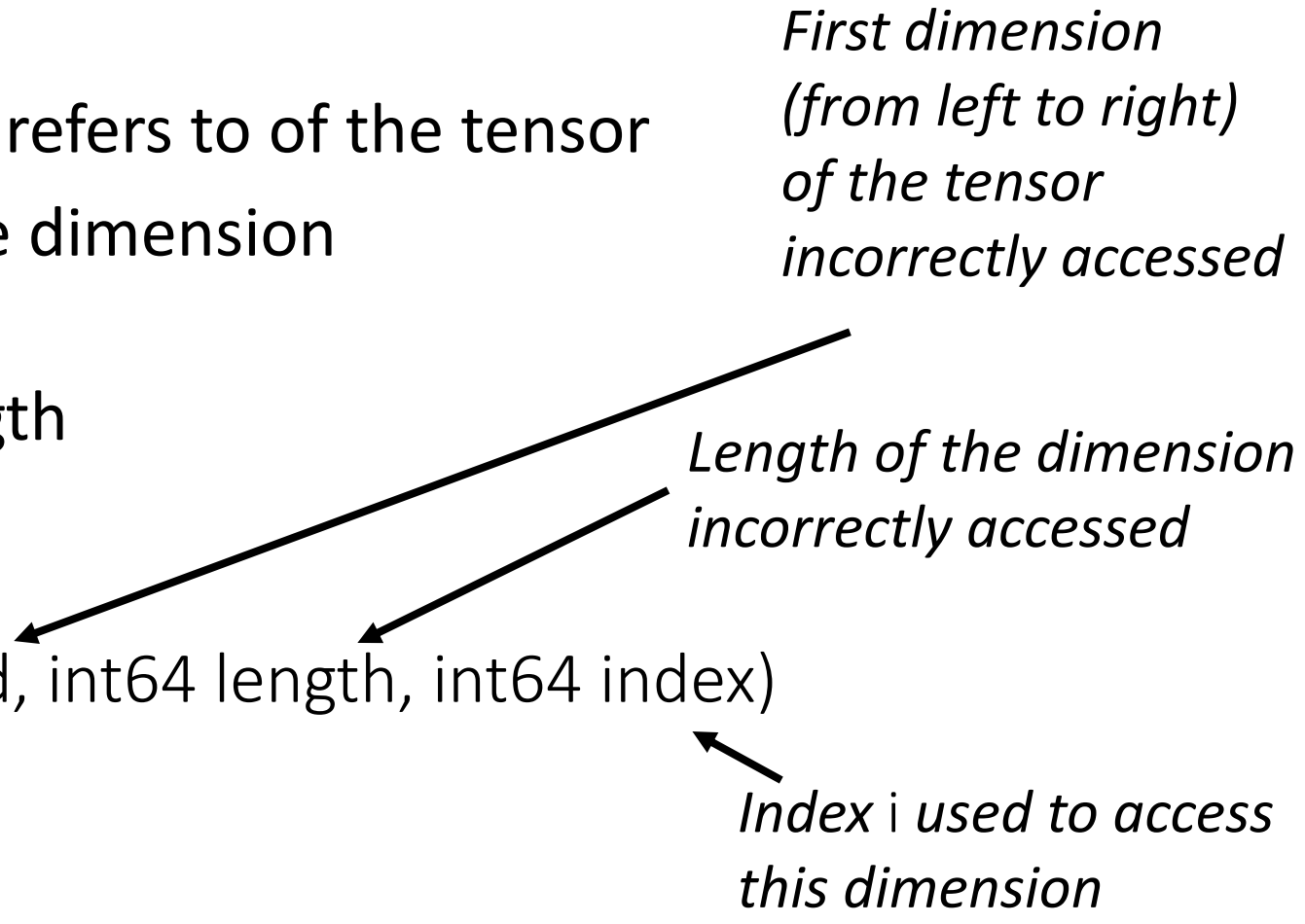
Length of the dimension incorrectly accessed

The negative index i *used to access this dimension*

Checking tensor accesses

B. Check i is less than
the length of the dimension it refers to of the tensor

1. Load the length of the relative dimension
(e.g., l_i <- length t 1)
2. Check if i is less than that length
(e.g., i < l_i)
3. If it isn't, then call
tensor-error(int64 line, int64 d, int64 length, int64 index)



Outline

- LA
- Encoding values
- Checking array accesses
- Generating basic blocks and variable declarations

Instructions without basic blocks

Instructions organized in basic blocks

```
Inst = F.entryPoint() ; newInsts = new List() ; startBB = true ;
While (Inst){
 if (startBB){   // Next instruction must be a label
    if (Inst is not Label) {
       L = new Label() ; newInsts.append(L) ;
     }
    startBB = false;
 } else if (Inst is Label){  // L1 i+ L2 …
    g = new Goto(Inst); newInsts.append(g) ; // L1 i+ goto L2 …
 }
 newInsts.append(Inst) ;
 if (Inst is Terminator) { // Next instruction must be a label
    startBB = true;
 }
 Inst = F.nextInst(Inst);
}
…
```

See next slide

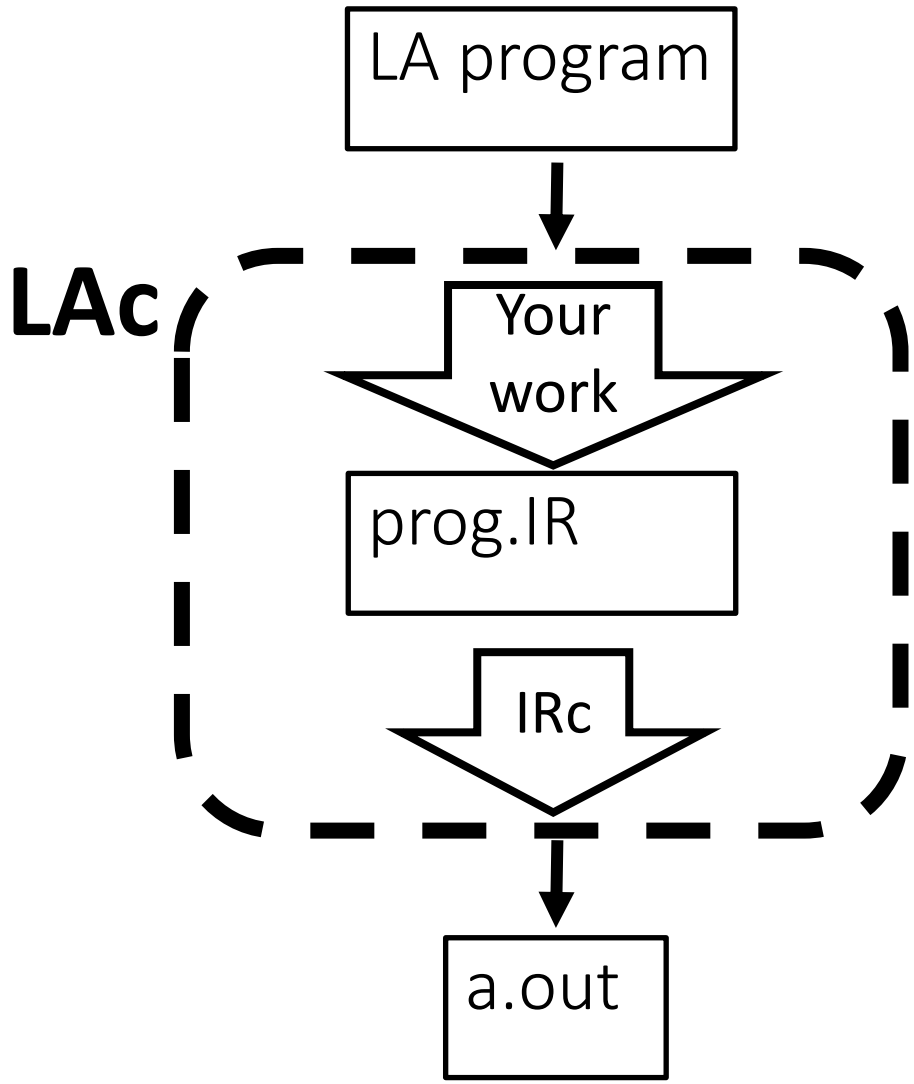
Previous slide

```
...
If (!startBB){
  if (Function's return type is void){
    r = new Return()
  } else {
    r = new Return(0)
  }
  newInsts.append(r)
}
```

Variable declarations

- Collect all LA variable declarations
- Generate IR variable declarations (in any order)
  at the beginning of the first basic block
- Append to the first basic block the initialization of code variables to 0

The LA compiler (LAc)



- To build LAc:
  translate an LA program
  to an equivalent IR one

- We need to
  1. Encode values
  2. Generate code to check memory accesses
  3. Create basic blocks

Homework #6

Write a compiler that translates an LA program (.a) to an IR one

- You need to generate prog.IR
- You need to pass all tests in the framework

Always have faith in your ability

Success will come your way eventually

Best of luck!