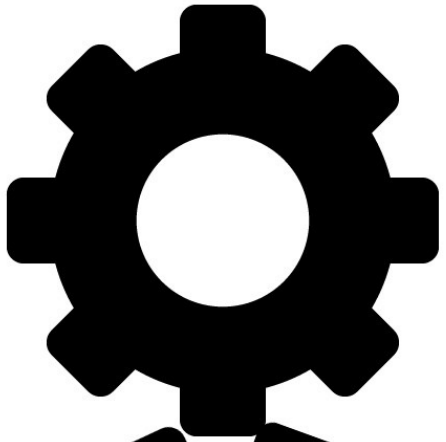
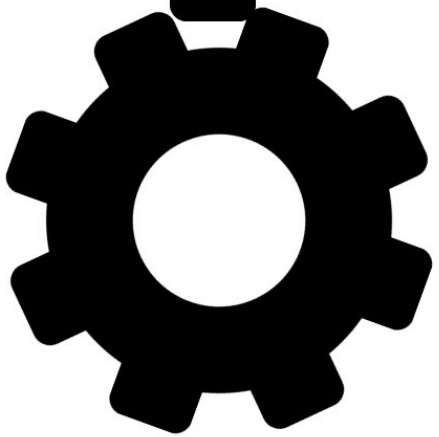
C  mpiler

C  nstruction

LB

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- LB
- Scope
- Control structures

A compiler

High level programming language

Front-end

- Name binding
- Control structure translation
- Values encoding
- Checking memory accesses
- Basic block generation

IR

Middle-end

IR

Back-end

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

Machine code

LA

```
int64 myF (int64 p1){
    int64 v1
    int64 v2
    v1 <- 1
    v2 <- 2
    print(v2)
    print(v1)
    int64 c
    c <- v1 = p1
    br c :true :false
    :true
    return 1
    :false
    return 2
}
```

LB

```
int64 myF (int64 p1){
    int64 v1, c
    v1 <- 1
    {
    int64 v1
    v1 <- 2
    print(v1)
    }
    print(v1)
    if (v1 = p1) :true :false
    :true
    return 1
    false:
    return 2
}
```

LB

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

LB example: scope

```
void main ( ){
   return
}
```

LB

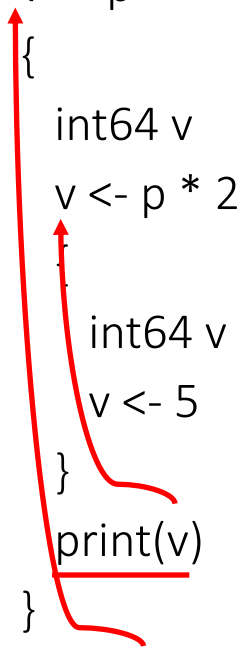
```
p       ::= f+
f       ::= T name ( pars ) scope
scope ::= { i* }
i       ::= type names | name <- t | name <- t op t |
            label | if (cond) label label | goto label | return t? |
           while (cond) label label | continue | break |
            name <- name([t])+ |  name([t])+ <- t |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

LB example: scopes

```
void myF (int64 p){
  {
    {
      return
    }
  }
}
```

LB example: variable's scope

```
void myF (int64 p){
  int64 v
  v <- p
  {
    int64 v
    v <- p * 2
    {
      int64 v
      v <- 5
    }
    print(v)
  }
  print(v)
  return
}
```



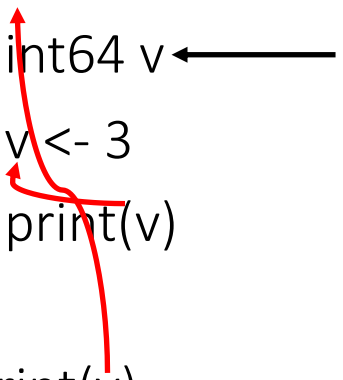
Assuming p is 21, the output is:

42

21

LB example: variable's scope (2)

```
void main (){
  int64 v
  v <- 1
  {
    v <- 2
    int64 v
    v <- 3
    print(v)
  }
  print(v)
  return
}
```



A variable declaration in LB affects only the code within its scope that comes after it

The output is:

3

2

LB

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

LB example: declaring variables

```
void myF (int64 p){
   int64 v1, v2, v3, v4
   return
}
```

LB

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

LB example: if statement

```
void myF (int64 v3){
  if (v3 > 1) :true :false

 :true
   print(v3)
   br :exit

 :false
   print(1)
 :exit
 return
}
```

Output if v3 is 2:
2

LB example: if statement

```
void myF (int64 v3){
   if (v3 > 1) :true :false


  :true
    print(v3)
    br :exit


  :false
    print(1)
  :exit
  return
}
```

Output if v3 is 2:

2

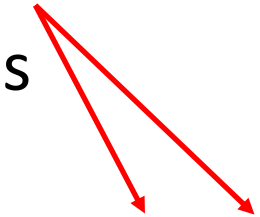
1

LB

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

LB while loops

- Different while loops must have different labels
  - Each loop has its own set of unique labels



while (cond) label label

- Exit label of a while loop must appear after both the beginning label and the while instruction

LB example: while statement with continue

```
void myF (int64 v) {
  int64 c
  c <- 0
  while (c < v) :body :exit
    :body
    print(c)
    c <- c + 1
    continue
  :exit
  return
}
```

LB example: while statement w/o continue

```
void myF (int64 v) {
  int64 c
  c <- 0
  while (c < v) :body :exit
    :body
    print(c)
    c <- c + 1

  :exit
  return
}
```

LB example 2: while statement with scope

```
void myF (int64 v) {
  int64 c
  c <- 0
  while (c < v) :body :exit
  {
    :body
    print(c)
    c <- c + 1
    continue
  }
  :exit
  return
}
```

LB example 3: while statement

```
void myF (int64 v) {
  int64 c
  c <- 0
  {
    :body
    print(c)
    c <- c + 1
  }  while (c < v) :body :exit
  :exit
  return
}
```

LB example 4: while statement

```
void myF (int64[] v) {
  int64 l, index
  l <- length v 0
  index <- 0
  while (index < l) :body :exit
    :body
    print(index)
    index <- index + 1
    continue
  :exit
  return
}
```

LB example 5: while statement with break

```
void myF (int64 v) {
  int64 c
  c <- 0
  {
    :body
    print(c)
    if (c = 42) :WOW : WOW2
    :WOW
    break
    :WOW2
    c <- c + 1
  }  while (c < v) :body :exit
  :exit
  return
}
```

Final notes on LB

- Same standard library of LA
  - int64 input (void)
  - void print (int64([])*)
  - void print (tuple)

- As in LA, LB variables are implicitly initialized to zero

Now that you know LB

- Rewrite all your LA programs and
- write a new LB program

Outline

- LB

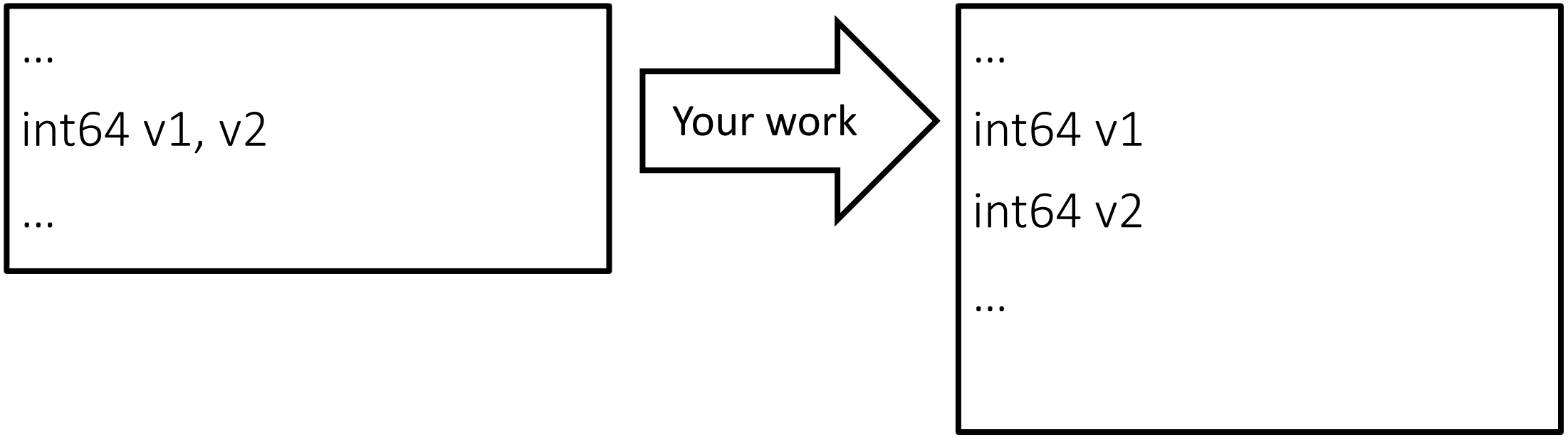
- Scope

- Control structures

To translate an LB function to LA

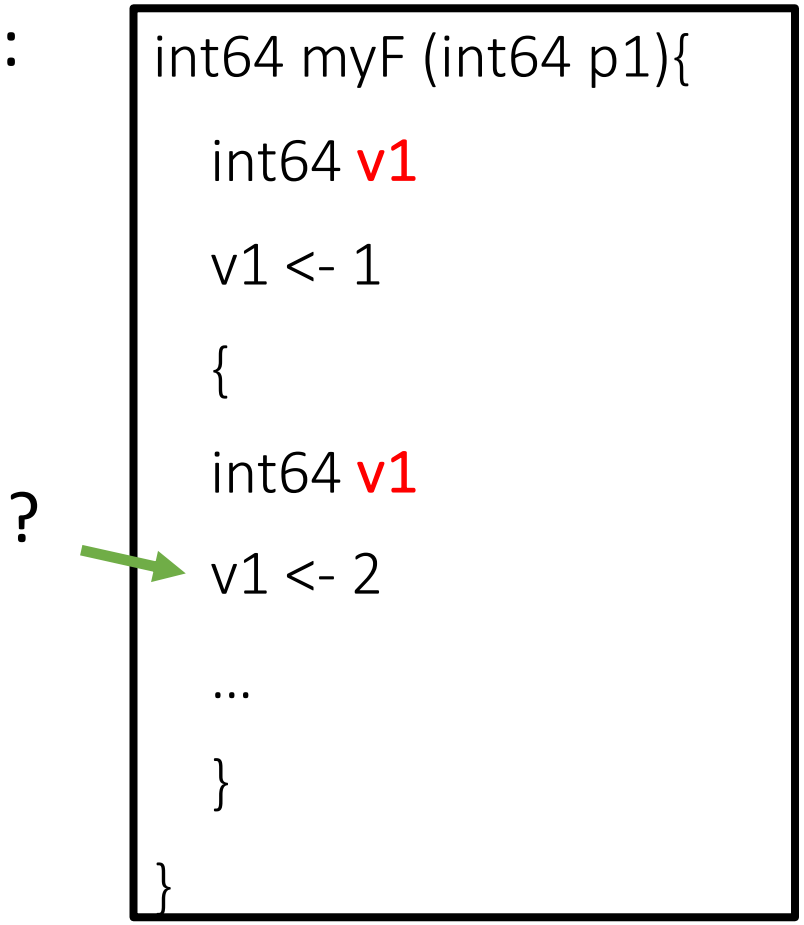
- Translate multiple variable declarations
- Flat the nested scopes
- Translate if and while statements

Translate multiple declarations



Name binding

- Association of entities (e.g., variables) with identifiers
- For example:



Scope

- Determines which names bind to which entities (e.g., variables)

```
int64 myF (int64 p1){
   int64 v1
   v1 <- 1
   {
   int64 v1
   v1 <- 2
   print(v1)
   }
   print(v1)
}
```

Binding time

- Static (**LB variables**, C variables, C++ variables, Java variables)
- Dynamic (C++ virtual methods, Java object methods)

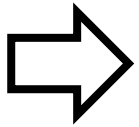
```
class  A{
   virtual void myF ();
}
class B : A{
     void myF () override;
}
void anotherF (A *obj) {
      obj->myF();
}
```

Translating an LB function

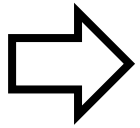
1. For each variable *v* declared by instruction *i* in a scope *s*
   - Rename *v* (only) in *i* to a new and unique name *z*
   - Remember the mapping *v* -> *z* (e.g., *s.map[v] = z*)
2. For each instruction *i* of a function
   - For each variable *v* referenced by *i*
     - Find the innermost scope *s* (starting from *i*) that declares *v* before *i* in the original code
     - If *s* exists, Function parameters
       then change the reference *v* of *i* from *v* to *s.map[v]*
3. Remove all nested scopes

Example

```
int64 myF (int64 p1){
    int64 v1
    v1 <- 1
    {
    int64 v1
    v1 <- 2
    …
    }
}
```



```
int64 myF (int64 p1){
    int64 v1_0
    v1 <- 1
    {
    int64 v1_1
    v1 <- 2
    …
    }
}
```

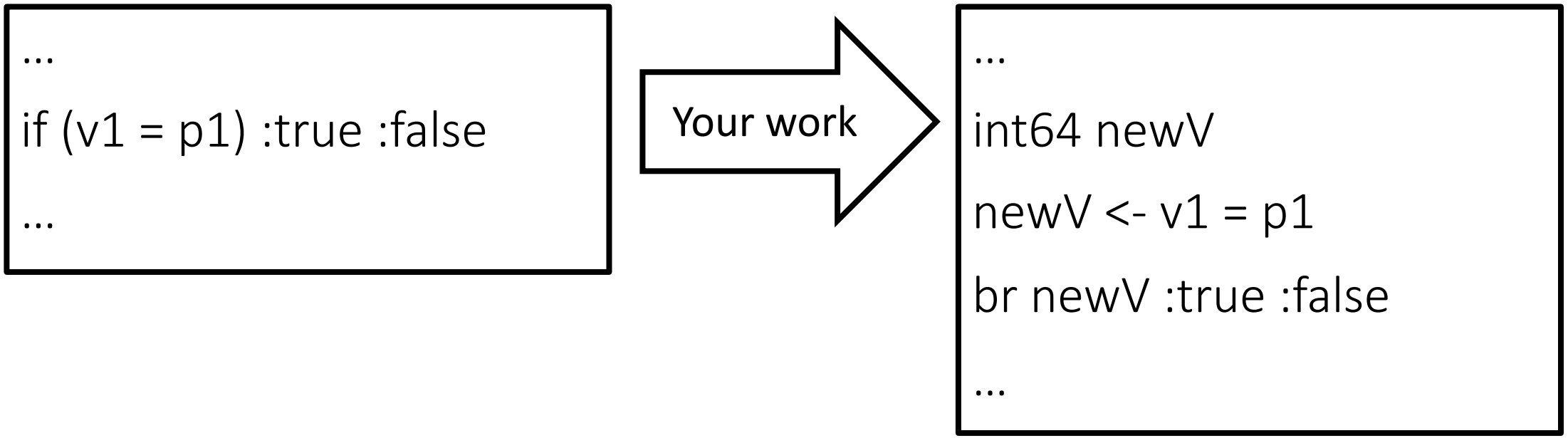


```
int64 myF (int64 p1){
    int64 v1_0
    v1_0 <- 1
    {
    int64 v1_1
    v1_1 <- 2
    …
    }
}
```

Outline

- LB
- Scope
- Control structures

Translating if structures

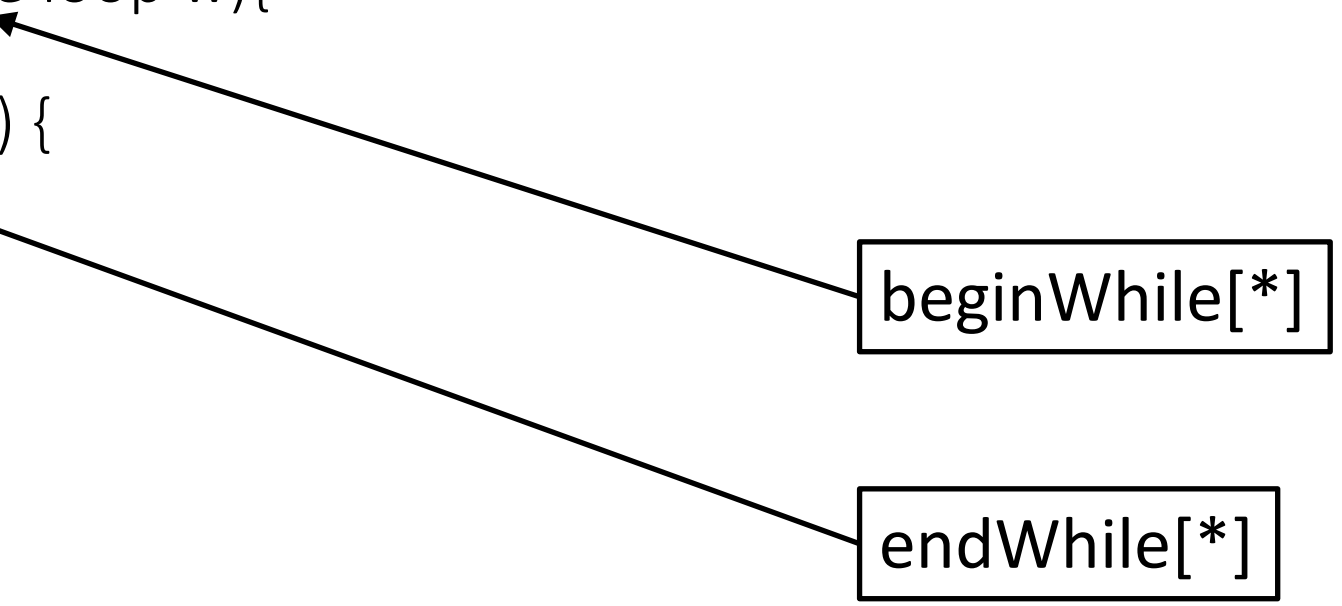


Translating while structures

1. Identify entry and exit point of each while instruction w
   while (%v1 = 3) :body_w :after_w
   - *beginWhile[w] = :body_w*
   - *endWhile[w] = :after_w*
2. Add a new label *l* just before a while instruction *w*
   - *condLabels[w] = l*
3. Map instructions to their more nested loop *w*
4. Translate while instructions
5. Translate continue and break instructions

Mapping instructions to their loops

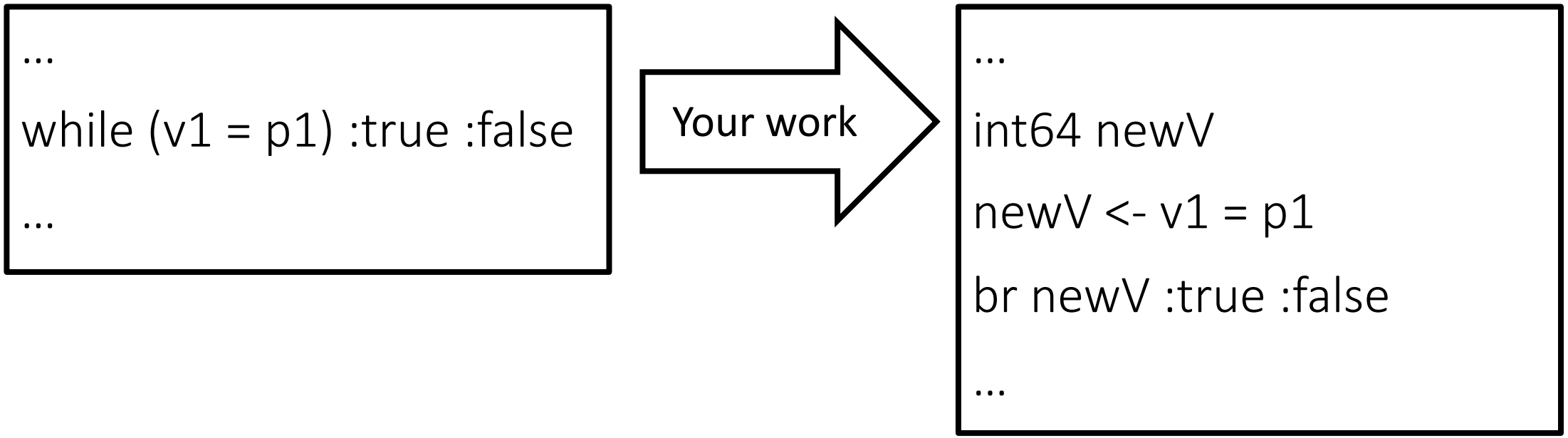
```
i = F.firstInstruction(); loopStack = Stack();
while (i){
  if (loopStack.size() > 0)
      w = loopStack.top(); loop[i] = w;
  if (i is a label){
     if (i is the beginning of a while loop w){
       loopStack.push(w);
     } else if (i is the end of a loop) {
       loopStack.pop();
     }
  }
  i = next(i);
}
```



Translating while structures

1. Identify entry and exit point of each while instruction w
   while (%v1 = 3) :body_w :after_w
   - *beginWhile[w] = :body_w*
   - *endWhile[w] = :after_w*
2. Add a new label *l* just before a while instruction *w*
   - *condLabels[w] = l*
3. Map instructions to their more nested loop *w*
4. Translate while instructions
5. Translate continue and break instructions

Translating while instruction



Translating while structures

1. Identify entry and exit point of each while instruction w
   while (%v1 = 3) :body_w :after_w
   - *beginWhile[w] = :body_w*
   - *endWhile[w] = :after_w*
2. Add a new label *l* just before a while instruction *w*
   - *condLabels[w] = l*
3. Map instructions to their more nested loop *w*
4. Translate while instructions
5. Translate continue and break instructions

Translating continue

- Let *i* be a continue instruction
- Fetch the innermost loop *w* that *i* belongs to:
  w = Loop[i]
- Fetch the label placed just before the condition of *w*:
  l_cond = condLabels[w]
- Generate a jump to the condition code of *w:*
  br l_cond

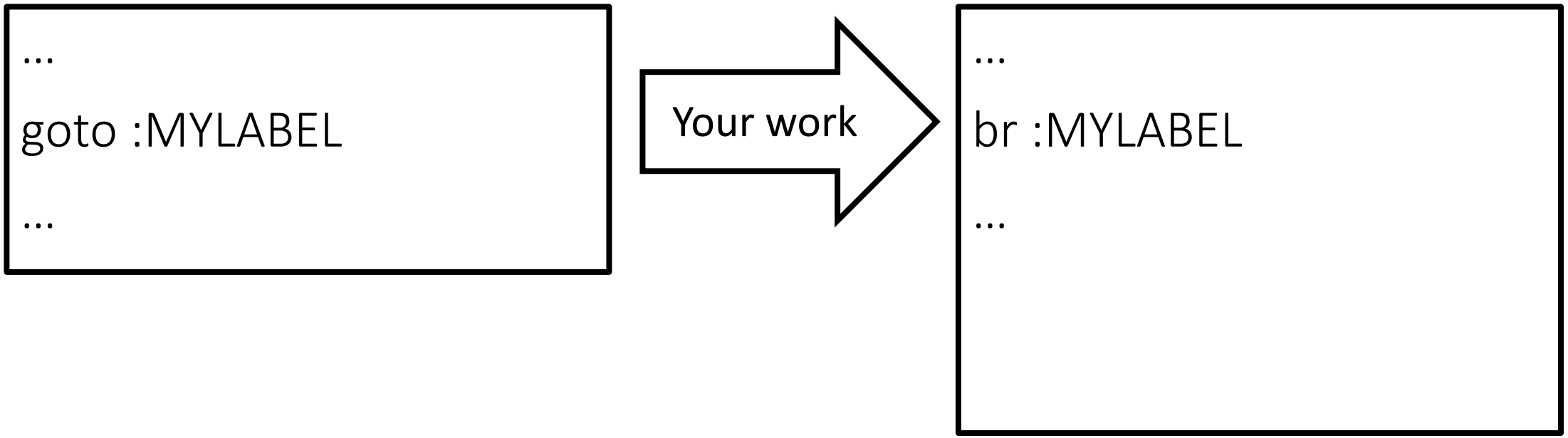
Translating break

- Let *i* be a break instruction
- Fetch the innermost loop *w* that *i* belongs to:
  w = Loop[i]
- Fetch the exit label of *w*:
  l_exit = endWhile[w]
- Generate a jump to leave the loop w:
  br l_exit

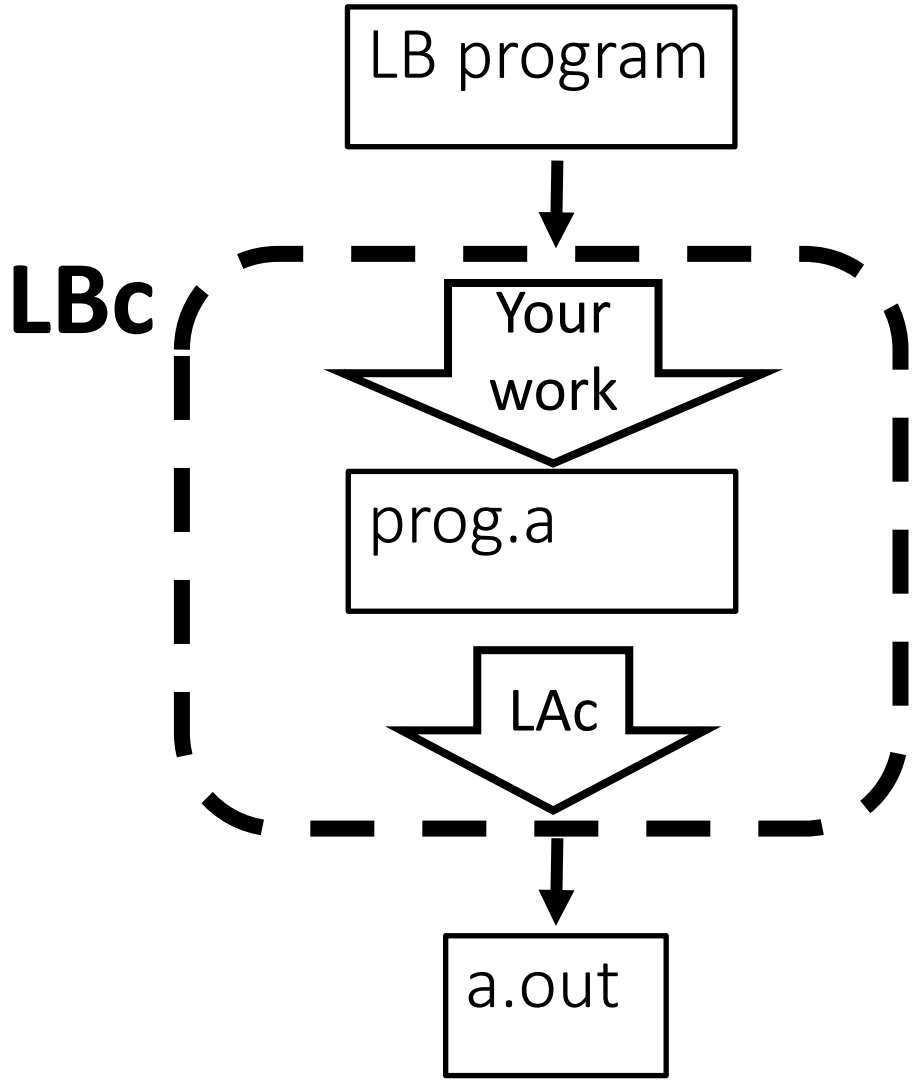
LB

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

Translating goto instruction



The LB compiler (LBc)



- Competition:
  During our last class
- Winner of the competition:
  - Get an A
  - His/her/their name(s) go to the Hall of Fame of the class

Homework #7

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

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

Always have faith in your ability

Success will come your way eventually

Best of luck!