LLVM: Intermediate Representation & Optimization

Seminar *Automation, Compilers, and Code-Generation*

July 6, 2016

Frank Emrich

RWTH AACHEN UNIVERSITY

Compiler Architecture

General three-phase compiler:

Compiler Architecture

General three-phase compiler:

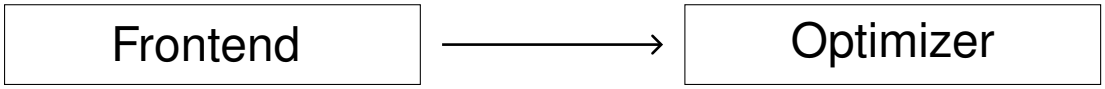
Frontend Check syntax, build Abstract Syntax Tree

Compiler Architecture

General three-phase compiler:

Frontend Check syntax, build Abstract Syntax Tree

Optimizer Optimizes some representation of the input program



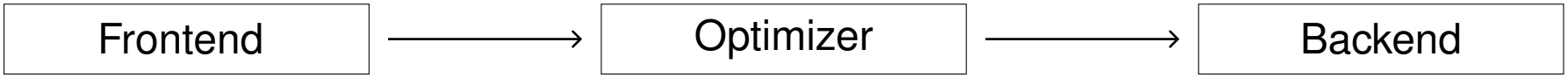
Compiler Architecture

General three-phase compiler:

Frontend Check syntax, build Abstract Syntax Tree

Optimizer Optimizes some representation of the input program

Backend Generates target-specific machine code



Compiler Architecture

General three-phase compiler:

Frontend Check syntax, build Abstract Syntax Tree

Optimizer Optimizes some representation of the input program

Backend Generates target-specific machine code

Realization by LLVM Compiler Infrastructure:

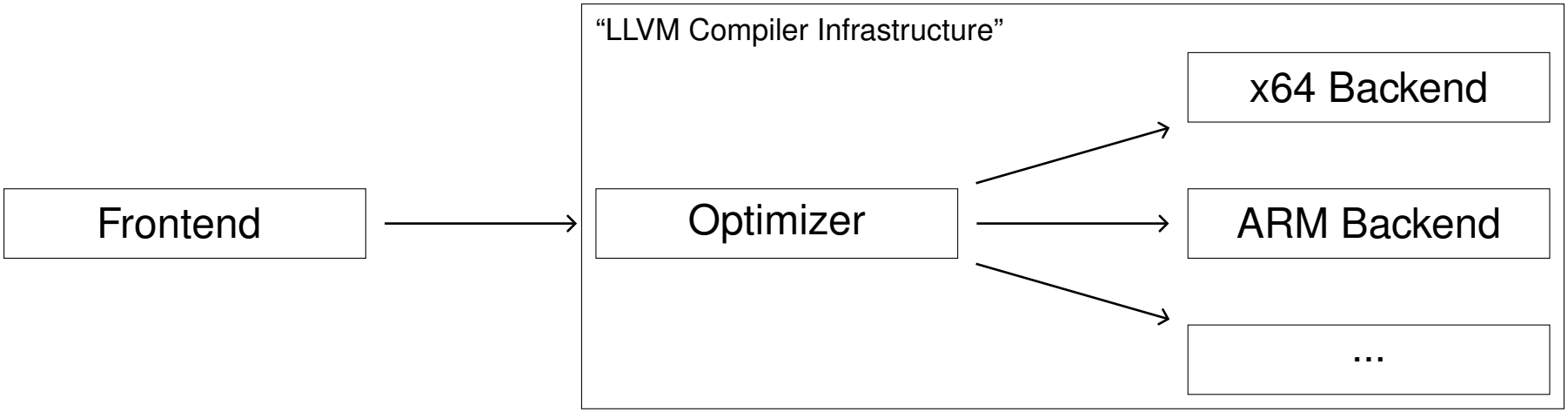
Compiler Architecture

General three-phase compiler:

Frontend Check syntax, build Abstract Syntax Tree

Optimizer Optimizes some representation of the input program

Backend Generates target-specific machine code



Realization by LLVM Compiler Infrastructure:

- Reusable Design: LLVM provides the optimizer and several backends

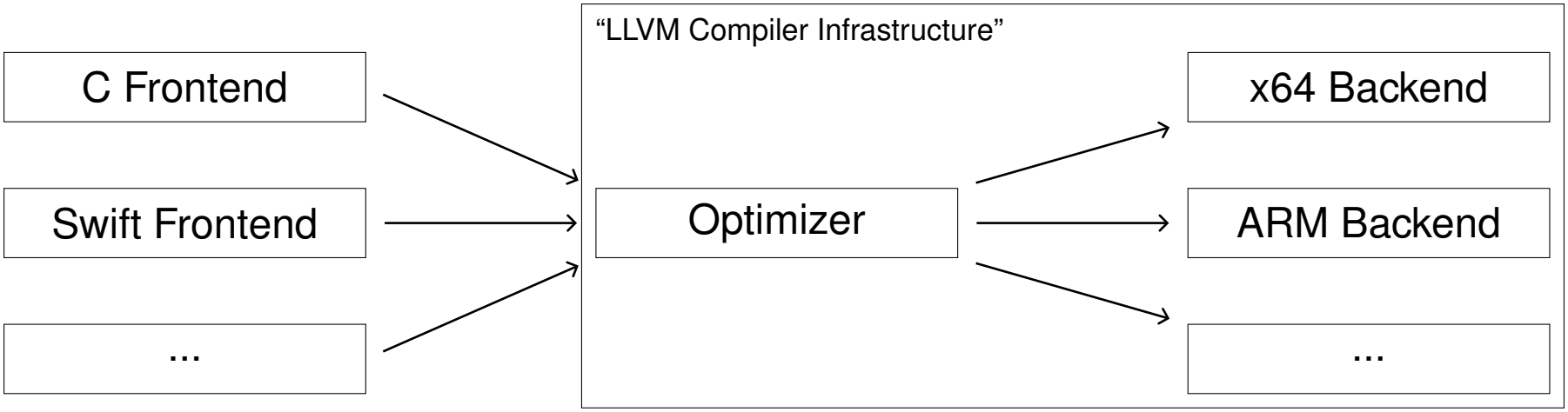
Compiler Architecture

General three-phase compiler:

Frontend Check syntax, build Abstract Syntax Tree

Optimizer Optimizes some representation of the input program

Backend Generates target-specific machine code



Realization by LLVM Compiler Infrastructure:

- Reusable Design: LLVM provides the optimizer and several backends
- Language-specific frontend + LLVM yields a whole compiler, e.g `clang`

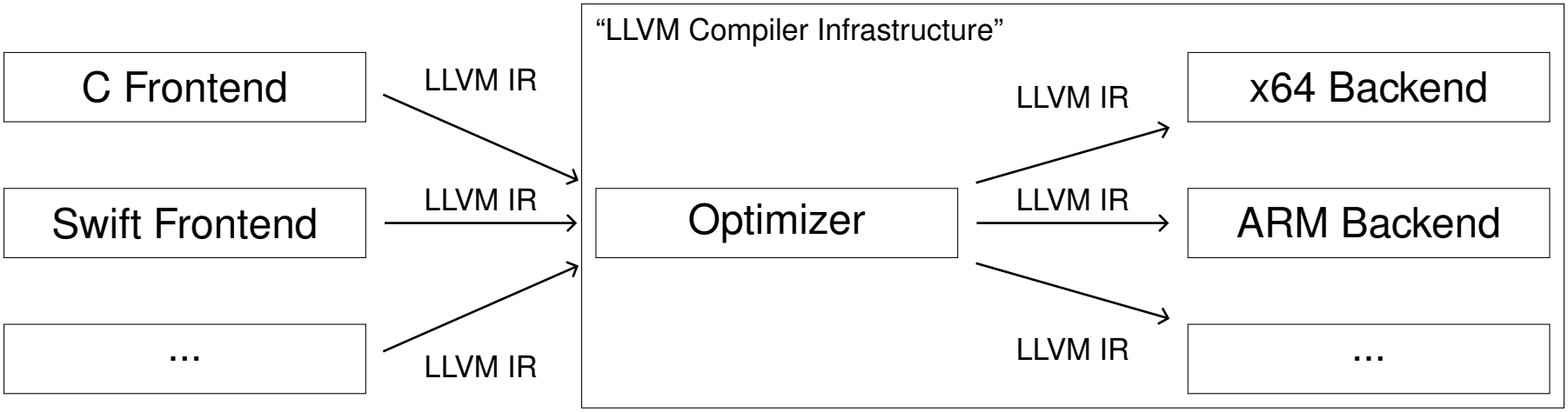
Compiler Architecture

General three-phase compiler:

Frontend Check syntax, build Abstract Syntax Tree

Optimizer Optimizes some representation of the input program

Backend Generates target-specific machine code



Realization by LLVM Compiler Infrastructure:

- Reusable Design: LLVM provides the optimizer and several backends
- Language-specific frontend + LLVM yields a whole compiler, e.g `clang`
- Optimizer works on the LLVM Intermediate Representation ("LLVM IR")
  - Assembly-like language
  - Complete representation of the original input program

LLVM Intermediate Representation

Example C program:

```
int example(int *arr1, int *arr2, int size, int a, int b) {
    int res = 0;
    for(int i = 0; i < size; i++) {
        int x = a + 3;
        int y = x + b;

        int cur = arr1[i];
        if(cur > 0) {
            arr1[i] = y;
        }

        res = arr2[0];
    }
    return res;
}
```

LLVM Intermediate Representation

Unoptimized Translation

`clang -S -emit-llvm -O0 example.c` yields the following:

LLVM Intermediate Representation

Unoptimized Translation

`clang -S -emit-llvm -O0 example.c` yields the following:

```
; ModuleID = 'c_example_0_hinten.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @example(i32* %arr1, i32* %arr2, i32 %size, i32 %a, i32 %b) #0 {
  %1 = alloca i32*, align 8
  %2 = alloca i32*, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %res = alloca i32, align 4
  %i = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %cur = alloca i32, align 4
  store i32* %arr1, i32** %1, align 8
  store i32* %arr2, i32** %2, align 8
  store i32 %size, i32* %3, align 4
  store i32 %a, i32* %4, align 4
  store i32 %b, i32* %5, align 4
  store i32 0, i32* %res, align 4
  store i32 0, i32* %i, align 4
  br label %6

; <label>:6                                       ; preds = %33, %0
  %7 = load i32, i32* %i, align 4
  %8 = load i32, i32* %3, align 4
  %9 = icmp slt i32 %7, %8
  br i1 %9, label %10, label %36

; <label>:10                                      ; preds = %6
  %11 = load i32, i32* %4, align 4
  %12 = add nsw i32 %11, 3
  store i32 %12, i32* %x, align 4
  %13 = load i32, i32* %x, align 4
  %14 = load i32, i32* %5, align 4
  %15 = add nsw i32 %13, %14
  store i32 %15, i32* %y, align 4
  %16 = load i32, i32* %i, align 4
  %17 = sext i32 %16 to i64
  %18 = load i32*, i32** %1, align 8
  %19 = getelementptr inbounds i32, i32* %18, i64 %17
  %20 = load i32, i32* %19, align 4
  store i32 %20, i32* %cur, align 4
  %21 = load i32, i32* %cur, align 4
  %22 = icmp sgt i32 %21, 0
  br i1 %22, label %23, label %29

; <label>:23                                      ; preds = %10
  %24 = load i32, i32* %y, align 4
  %25 = load i32, i32* %i, align 4
  %26 = sext i32 %25 to i64
  %27 = load i32*, i32** %1, align 8
  %28 = getelementptr inbounds i32, i32* %27, i64 %26
  store i32 %24, i32* %28, align 4
  br label %29

; <label>:29                                      ; preds = %23, %10
  %30 = load i32*, i32** %2, align 8
  %31 = getelementptr inbounds i32, i32* %30, i64 0
  %32 = load i32, i32* %31, align 4
  store i32 %32, i32* %res, align 4
  br label %33

; <label>:33                                      ; preds = %29
  %34 = load i32, i32* %i, align 4
  %35 = add nsw i32 %34, 1
  store i32 %35, i32* %i, align 4
  br label %6

; <label>:36                                      ; preds = %6
  %37 = load i32, i32* %res, align 4
  ret i32 %37
}

attributes #0 = { nounwind uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-
     non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr
     ,+mmx,+sse,+sse2" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"clang␣version␣3.8.0-2ubuntu3␣(tags/RELEASE_380/final)"}
```

(~80 lines)

LLVM Intermediate Representation

Manual Translation

```
define i32 @example(i32* %arr1, i32* %arr2, i32 %size, i32 %a, i32 %b)  {
entry:        %enterLoop = icmp sgt i32  %size, 0
              br i1 %enterLoop, label %loopHeader, label %return


loopHeader:   %i = phi i32 [ 0, %entry], [ %nextI, %loopLast ]
              %x = add i32 %a, 3
              %y = add i32 %x, %b

              %arr1AtI = getelementptr i32, i32* %arr1, i32 %i
              %cur = load i32, i32* %arr1AtI

              %shouldStore = icmp sgt i32 %cur, 0
              br i1 %shouldStore, label %loopStore, label %loopLast


loopStore:    store i32 %y, i32* %arr1AtI
              br label %loopLast


loopLast:     %res = load i32, i32* %arr2

              %nextI = add i32 %i, 1
              %continueLoop = icmp slt i32 %nextI, %size
              br i1 %continueLoop, label %loopHeader, label %return


return:       %finalres = phi i32 [ 0, %entry], [ %res, %loopLast ]
              ret i32 %finalres
}
```

LLVM Intermediate Representation

Manual Translation

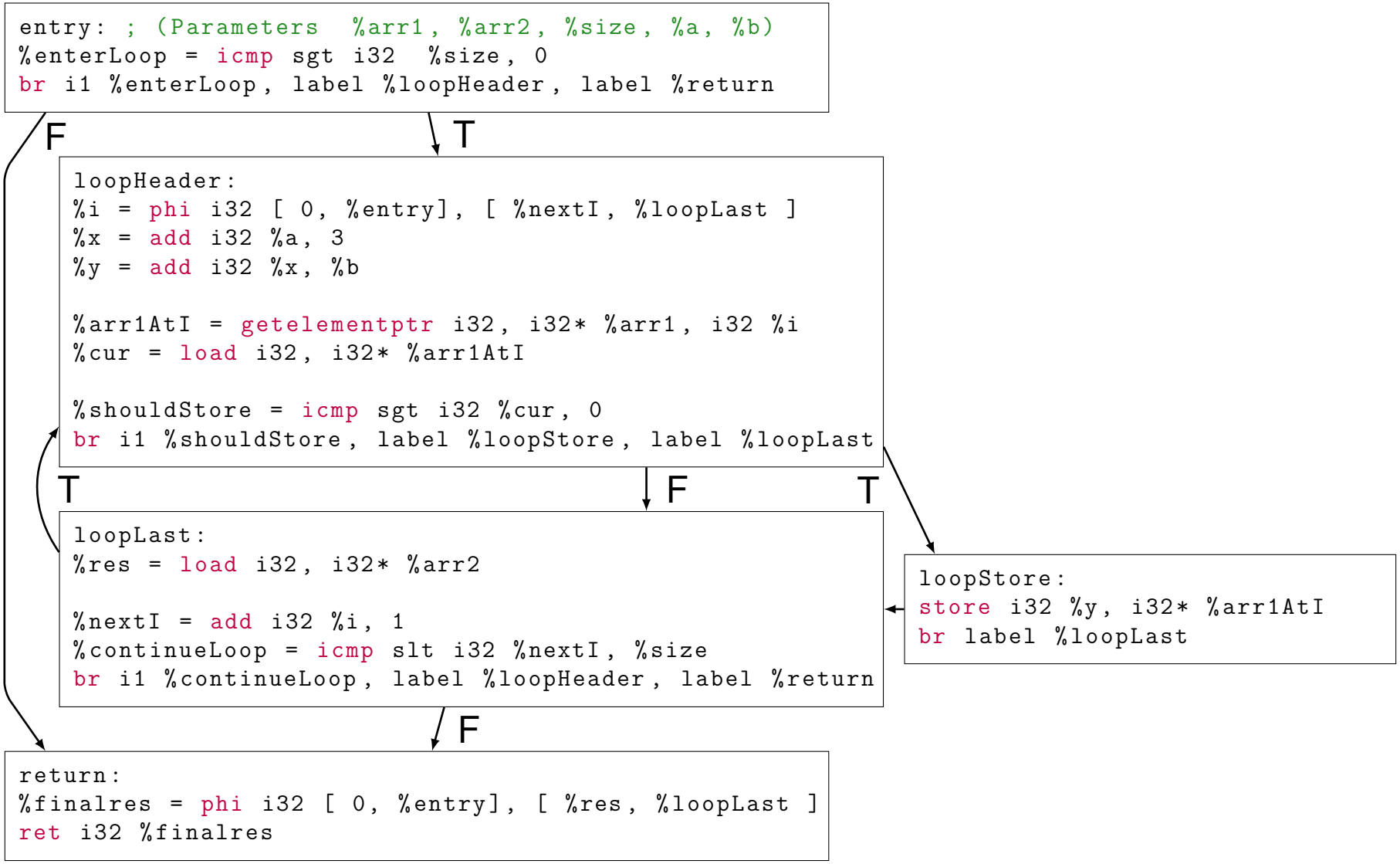
```
define i32 @example(i32* %arr1, i32* %arr2, i32 %size, i32 %a, i32 %b)  {
entry:        %enterLoop = icmp sgt i32  %size, 0
              br i1 %enterLoop, label %loopHeader, label %return


loopHeader:   %i = phi i32 [ 0, %entry], [ %nextI, %loopLast ]
              %x = add i32 %a, 3
              %y = add i32 %x, %b

              %arr1AtI = getelementptr i32, i32* %arr1, i32 %i
              %cur = load i32, i32* %arr1AtI

              %shouldStore = icmp sgt i32 %cur, 0
              br i1 %shouldStore, label %loopStore, label %loopLast


loopStore:    store i32 %y, i32* %arr1AtI
              br label %loopLast


loopLast:     %res = load i32, i32* %arr2

              %nextI = add i32 %i, 1
              %continueLoop = icmp slt i32 %nextI, %size
              br i1 %continueLoop, label %loopHeader, label %return


return:       %finalres = phi i32 [ 0, %entry], [ %res, %loopLast ]
              ret i32 %finalres
}
```

First Observations:

- Functions
- Strongly typed
- Global vs. function scope
- Basic Blocks
- Conditional vs. unconditional branching

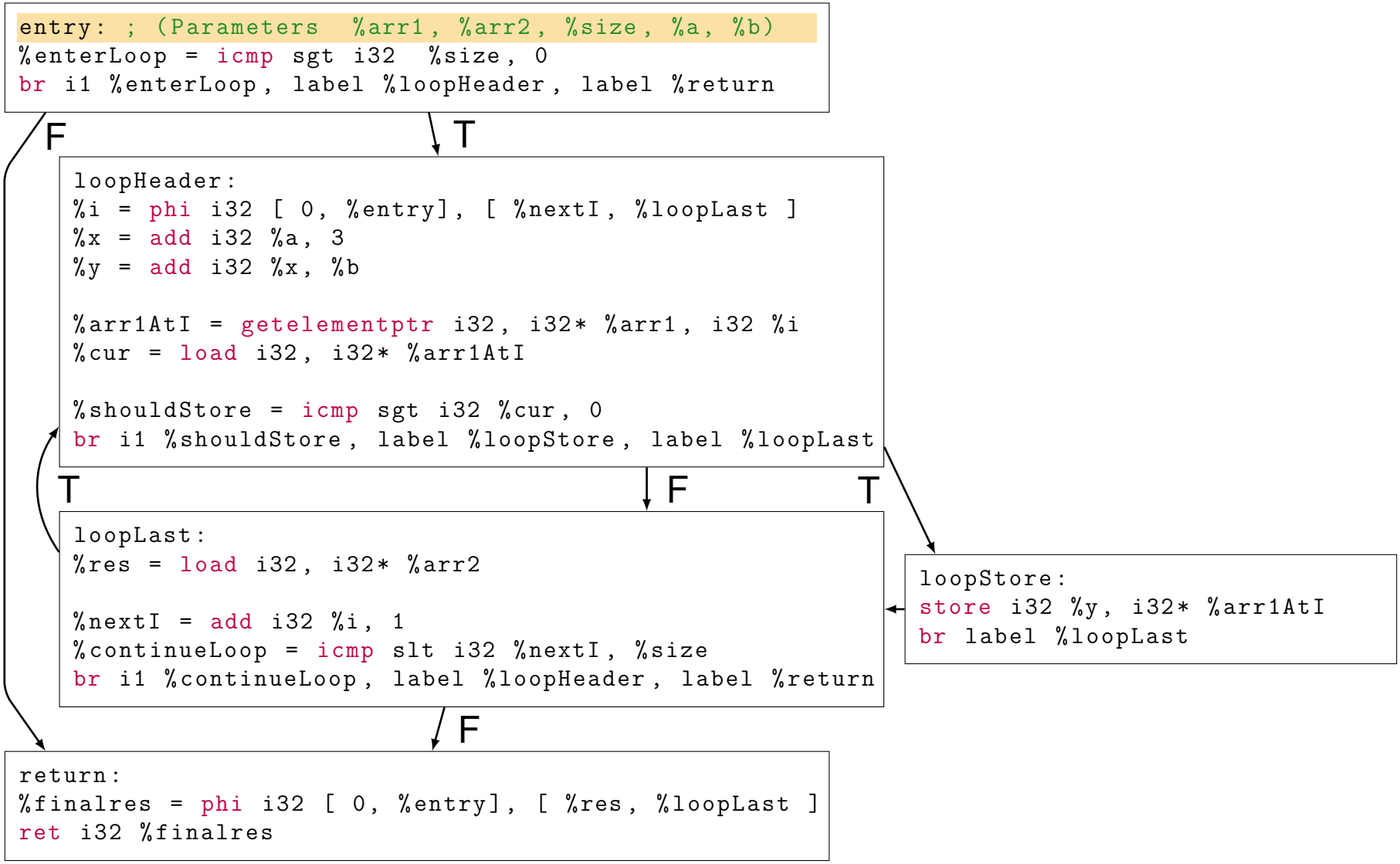
LLVM Intermediate Representation

Semantics (1)



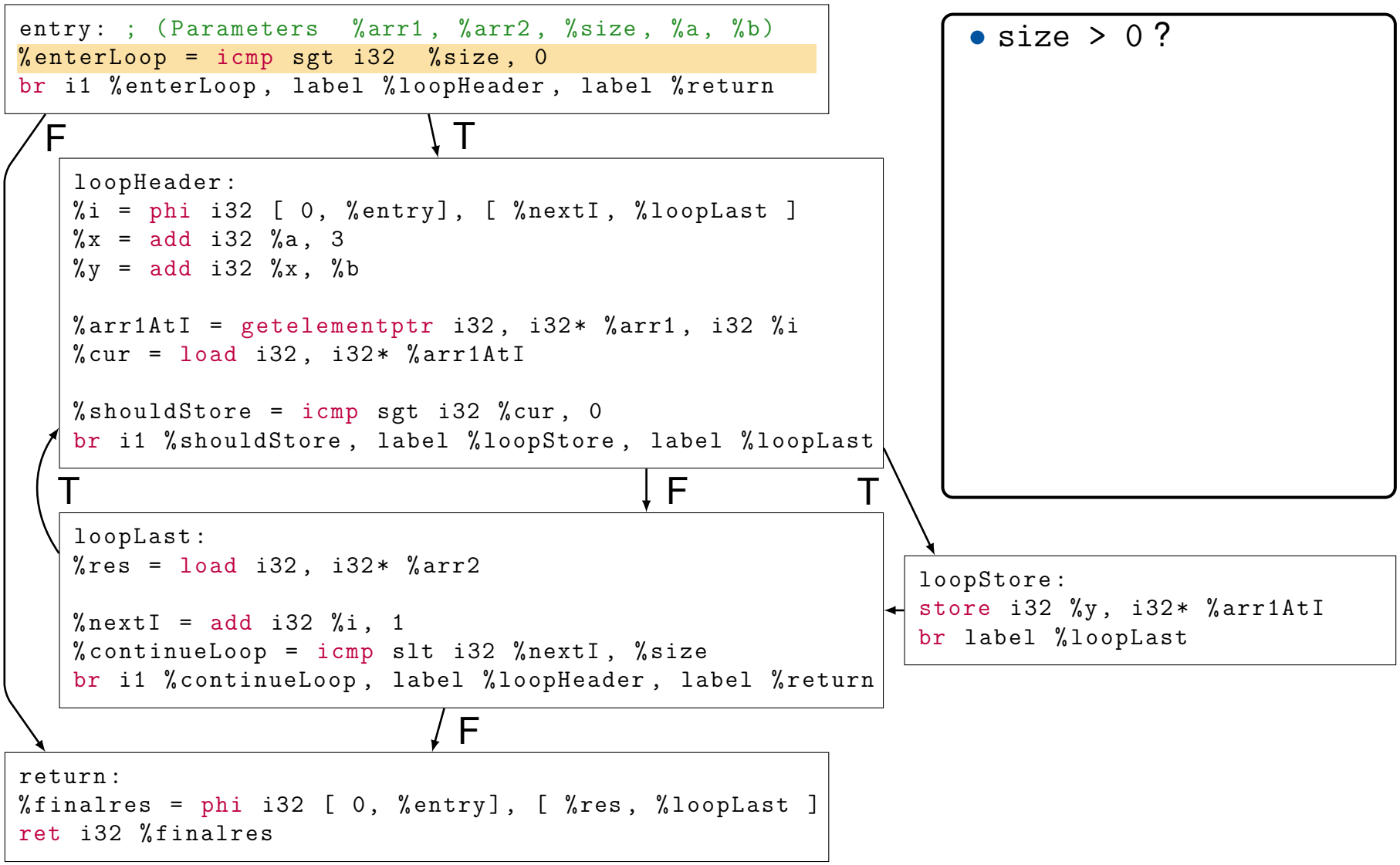
LLVM Intermediate Representation

Semantics (1)



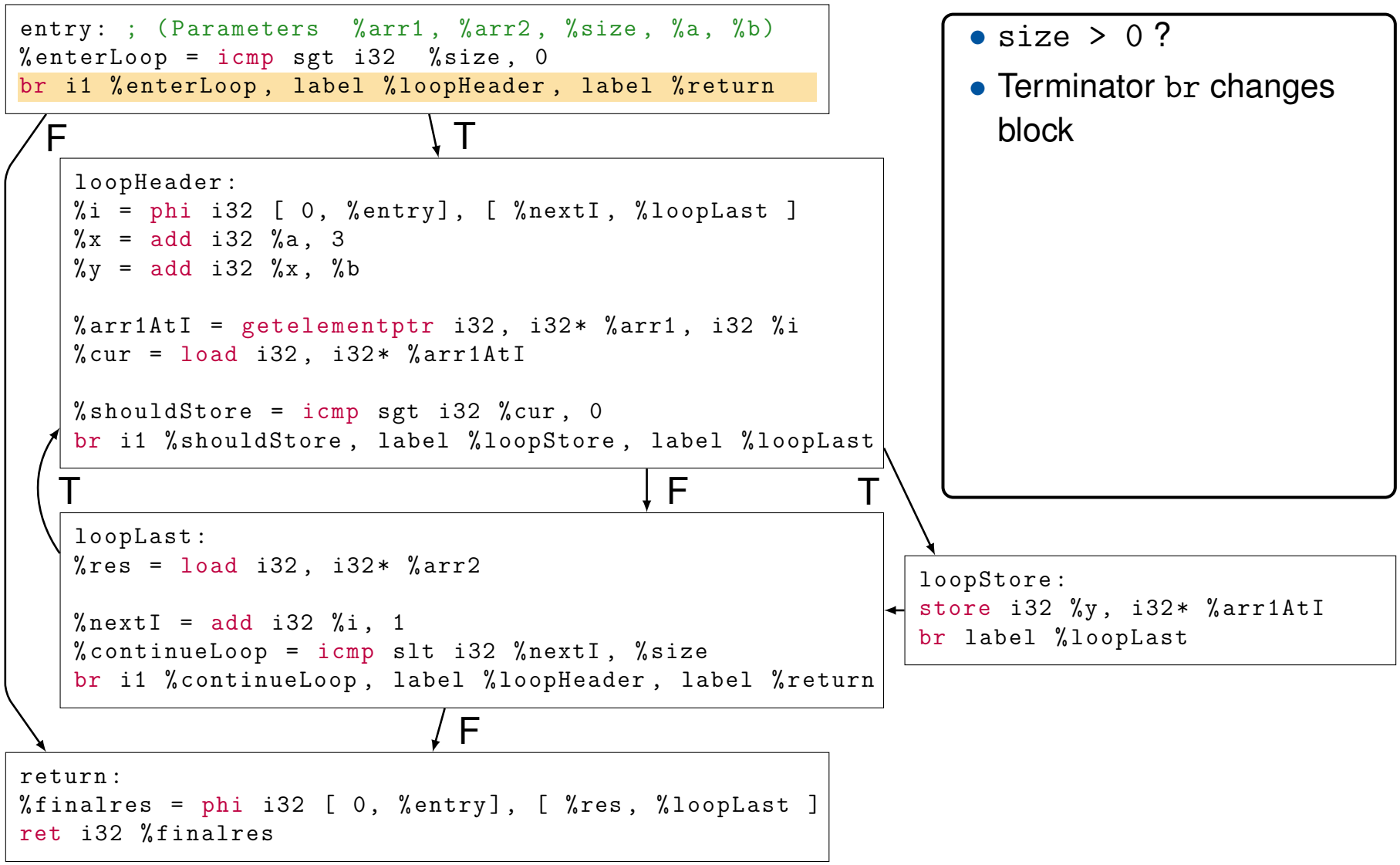
LLVM Intermediate Representation

Semantics (1)



LLVM Intermediate Representation

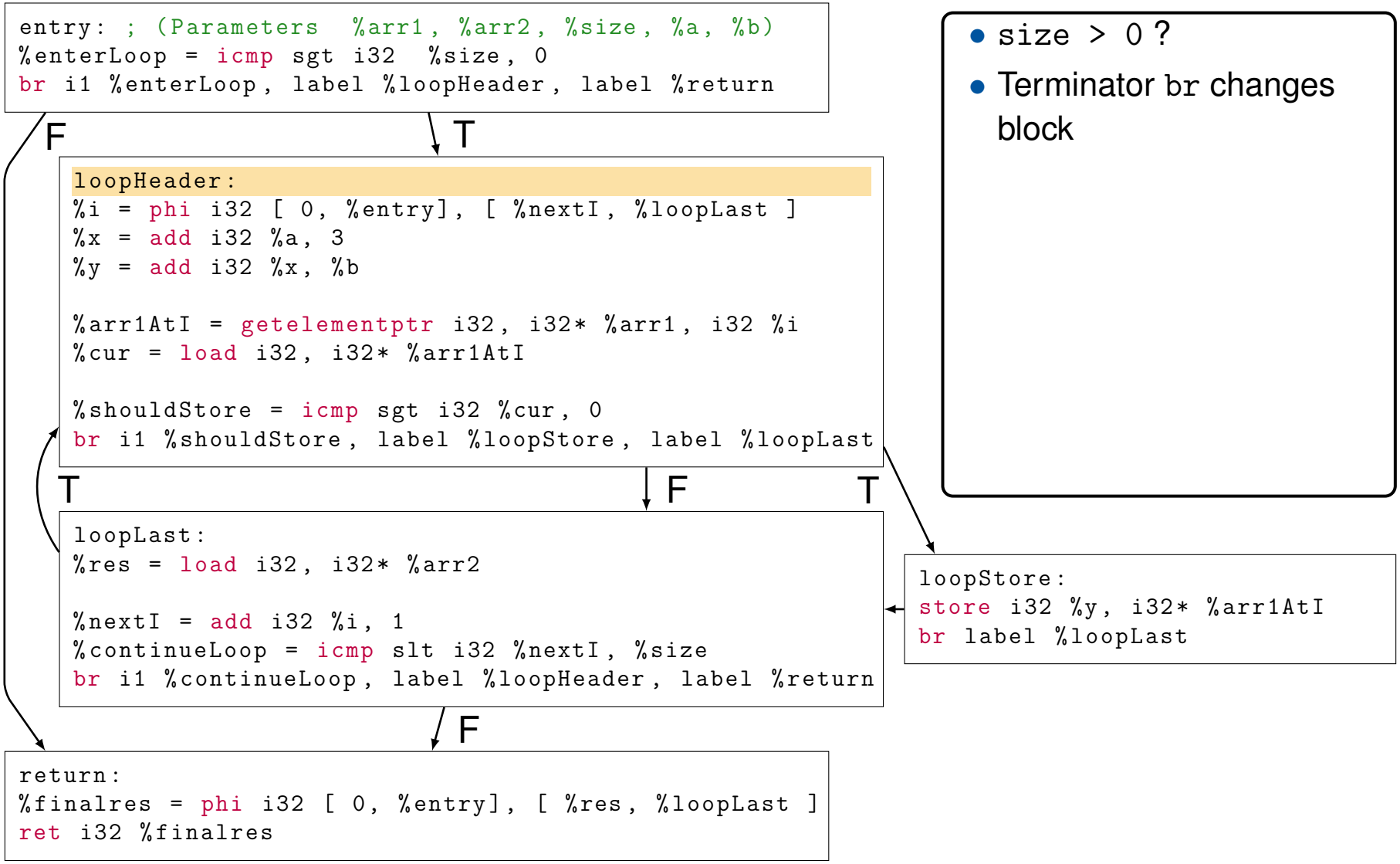
Semantics (1)



- `size > 0` ?
- Terminator `br` changes block

LLVM Intermediate Representation

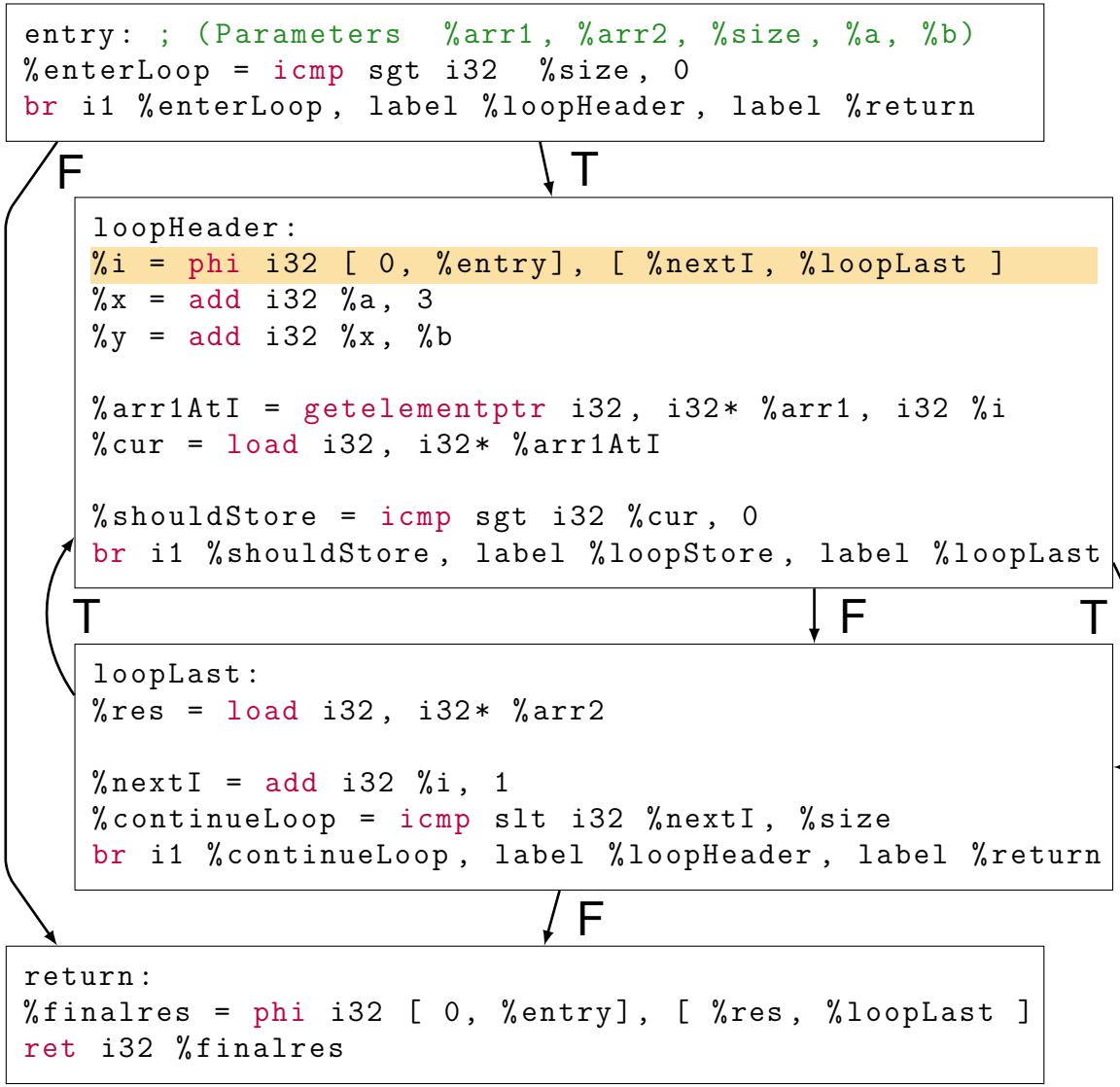
Semantics (1)



- `size > 0` ?
- Terminator `br` changes block

LLVM Intermediate Representation

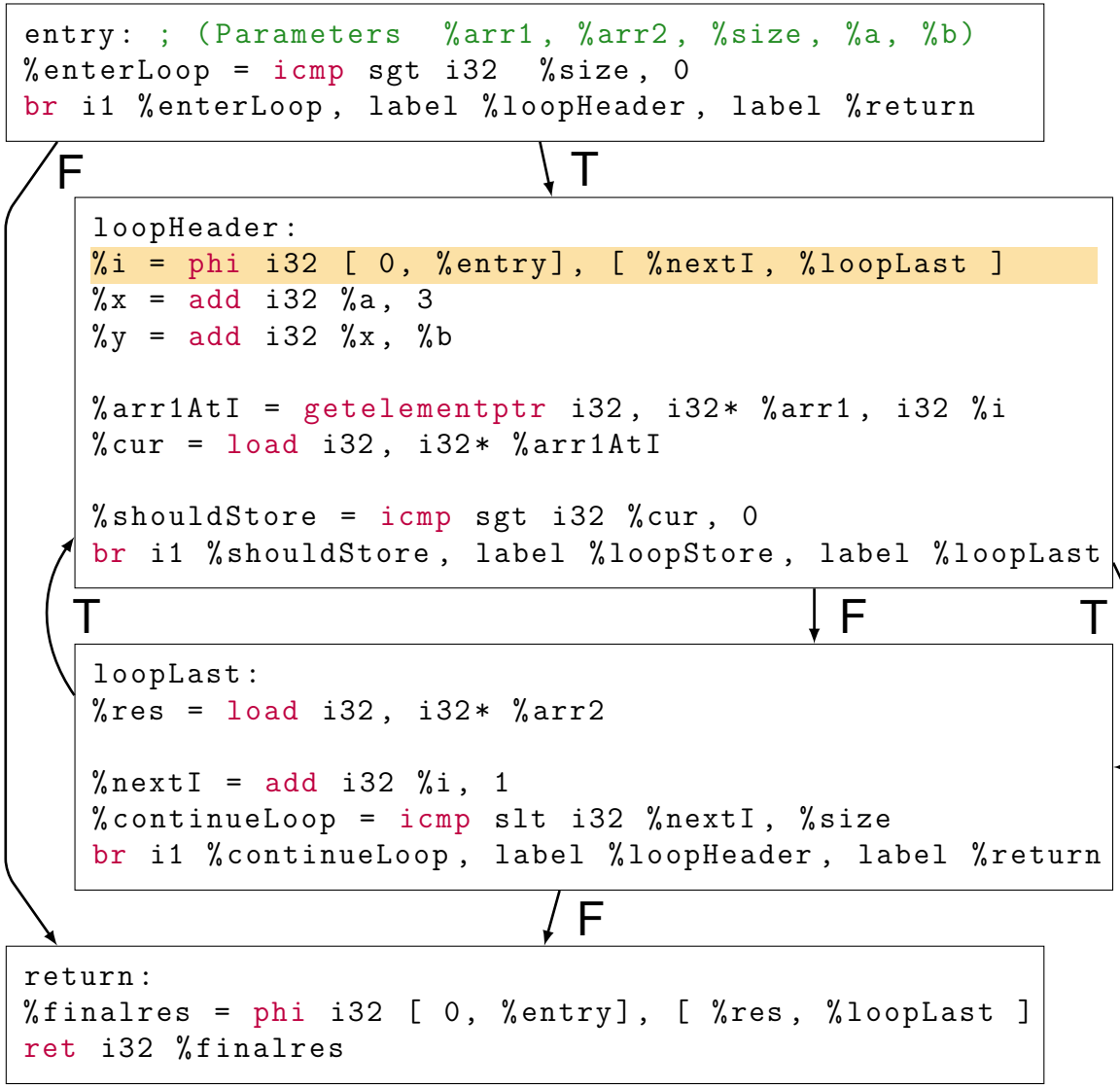
Semantics (1)



- `size > 0` ?
- Terminator `br` changes block
- Single Static Assignment: Only one assignment per variable

LLVM Intermediate Representation

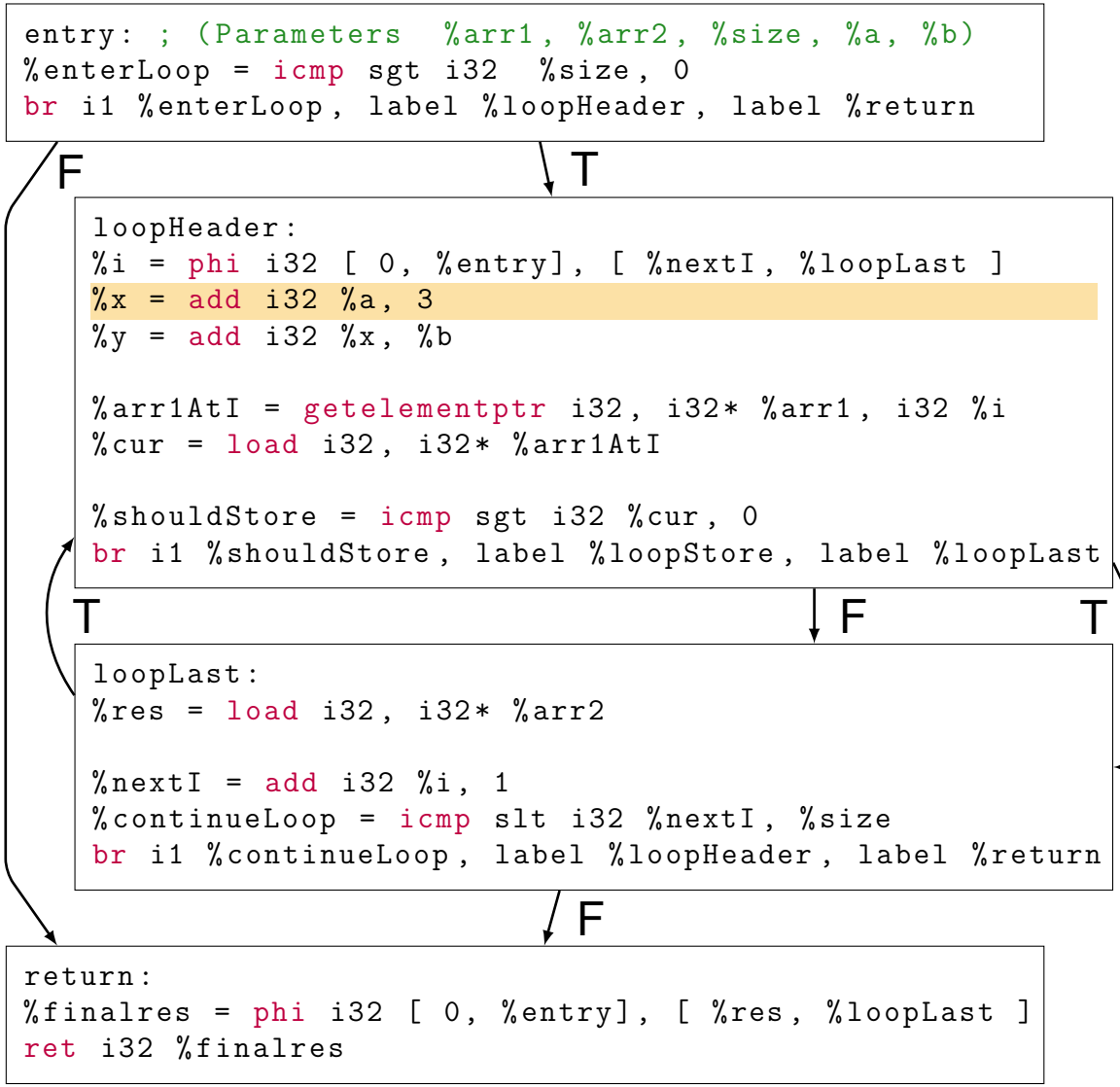
Semantics (1)



- `size > 0` ?
- Terminator `br` changes block
- Single Static Assignment: Only one assignment per variable
- Phi instructions respect previous block

LLVM Intermediate Representation

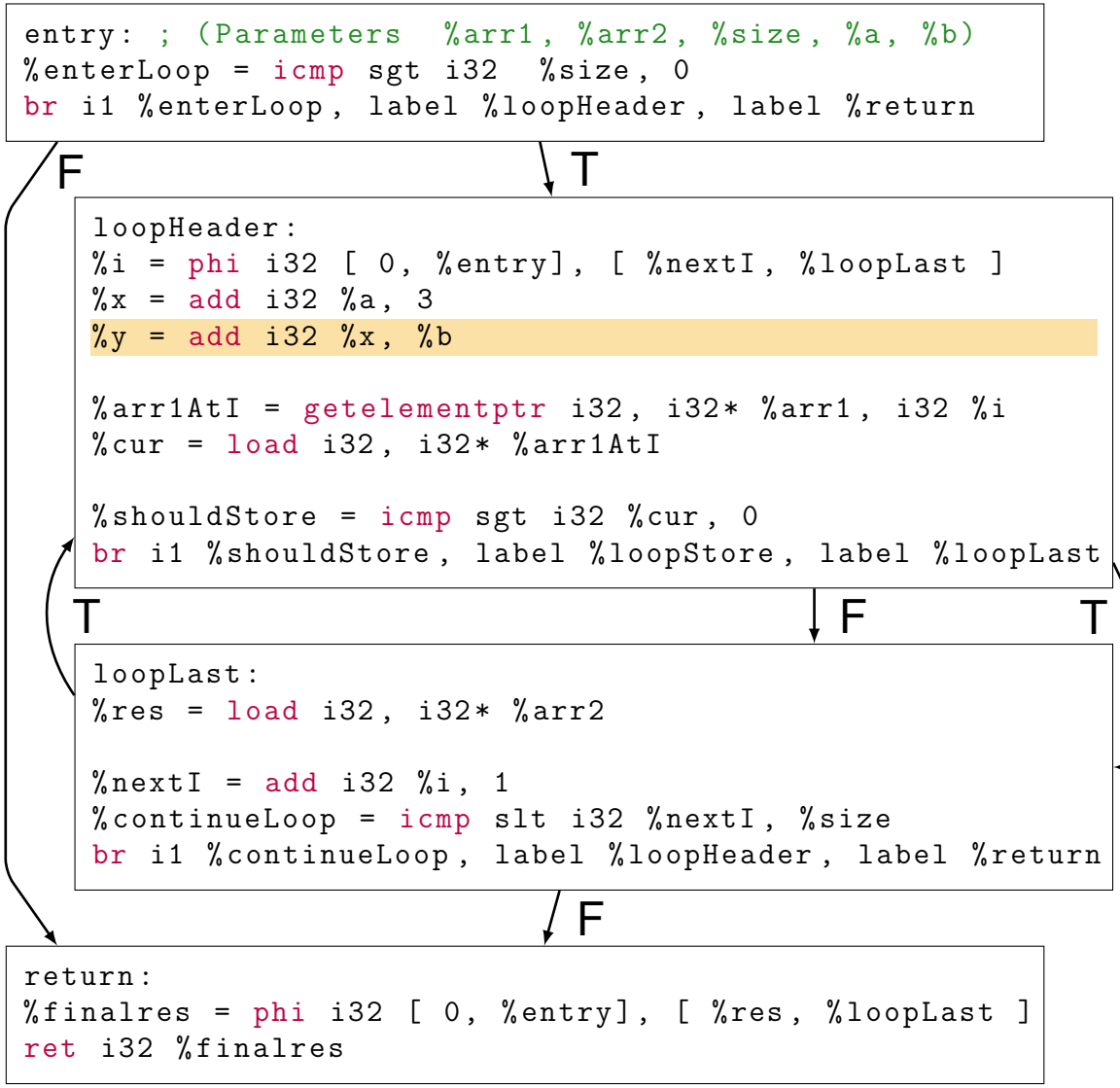
Semantics (1)



- `size > 0 ?`
- Terminator `br` changes block
- Single Static Assignment: Only one assignment per variable
- Phi instructions respect previous block
- `int x = a + 3`

LLVM Intermediate Representation

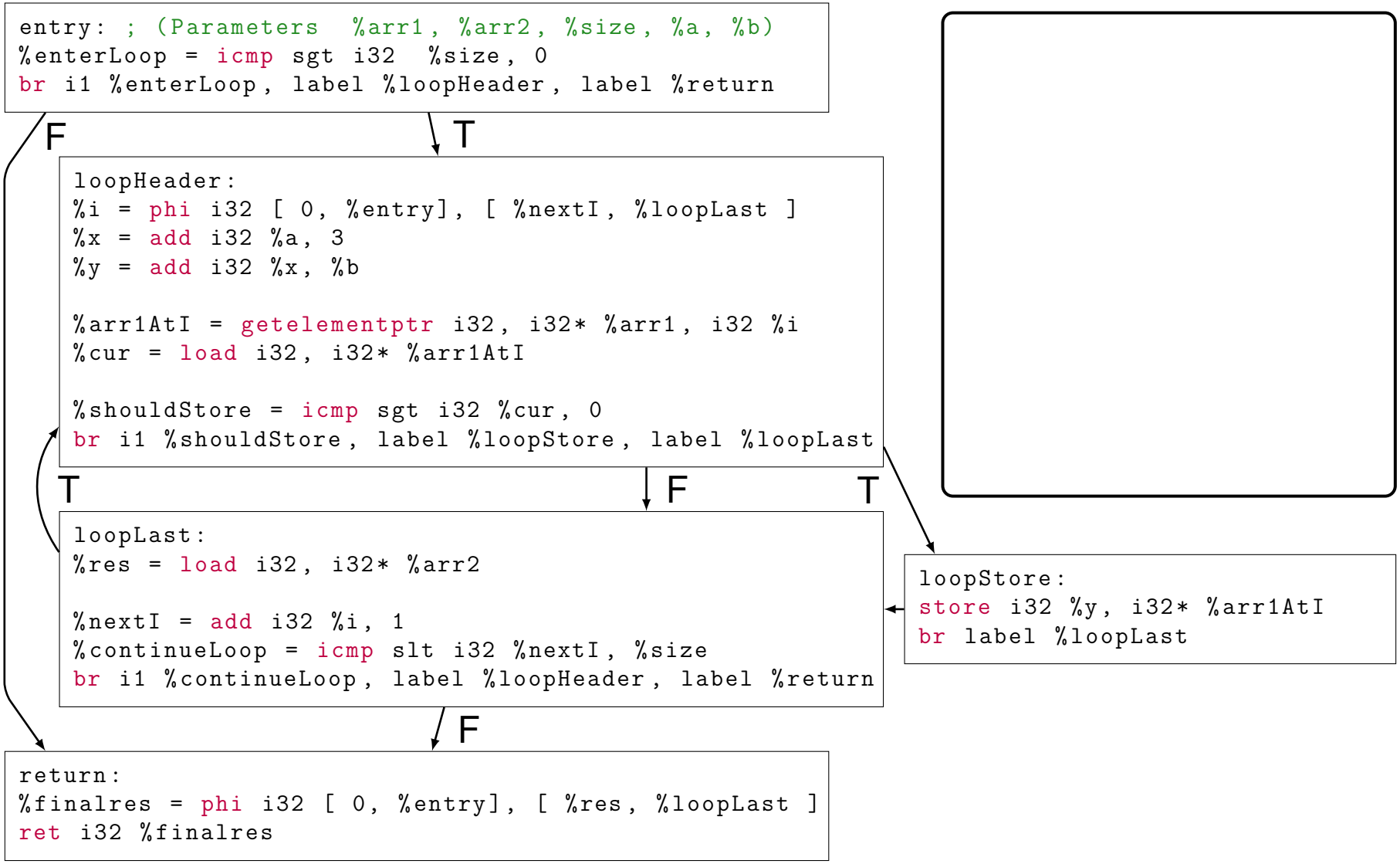
Semantics (1)



- `size > 0 ?`
- Terminator `br` changes block
- Single Static Assignment: Only one assignment per variable
- Phi instructions respect previous block
- `int x = a + 3`
- `int y = x + b`

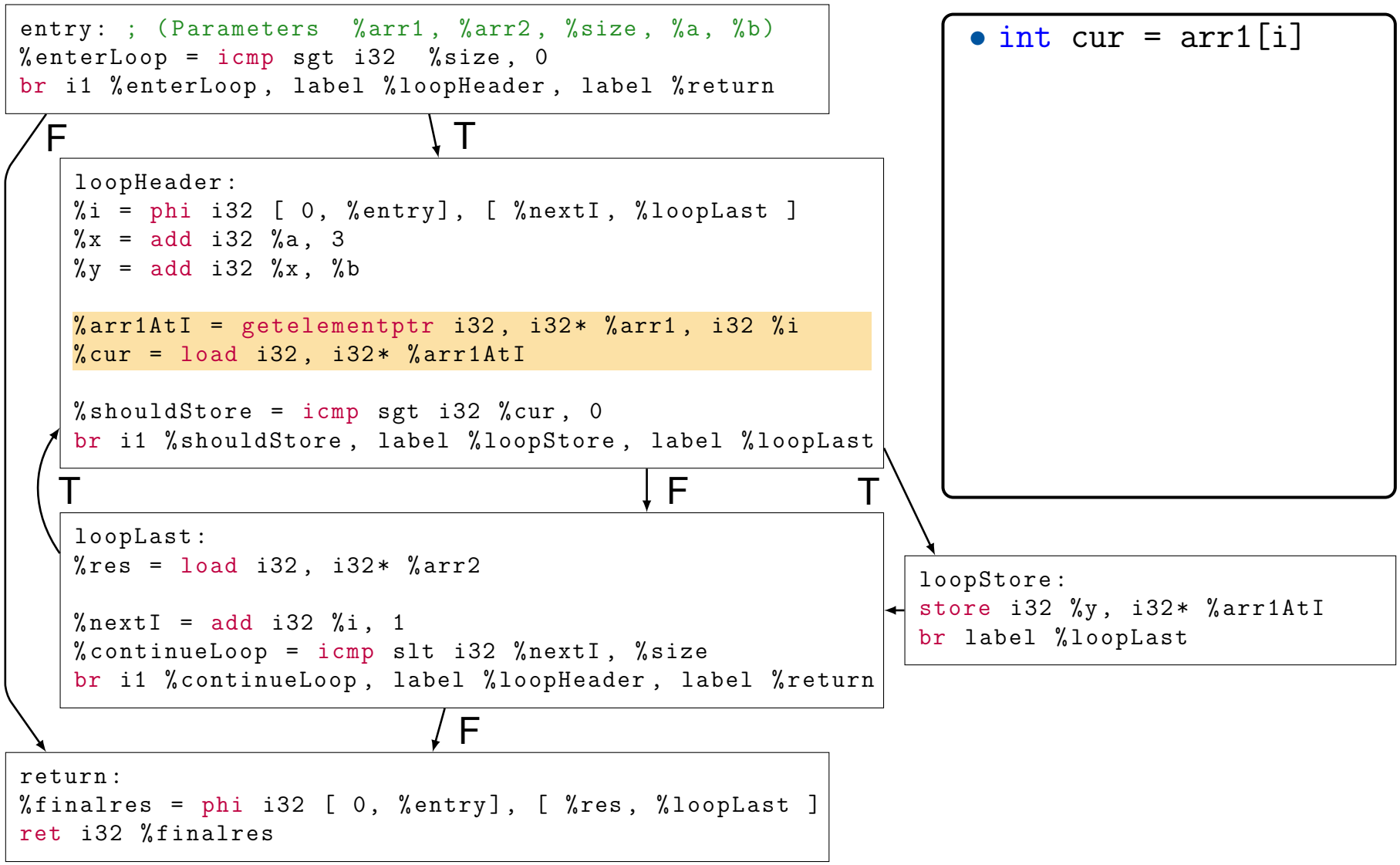
LLVM Intermediate Representation

Semantics (2)



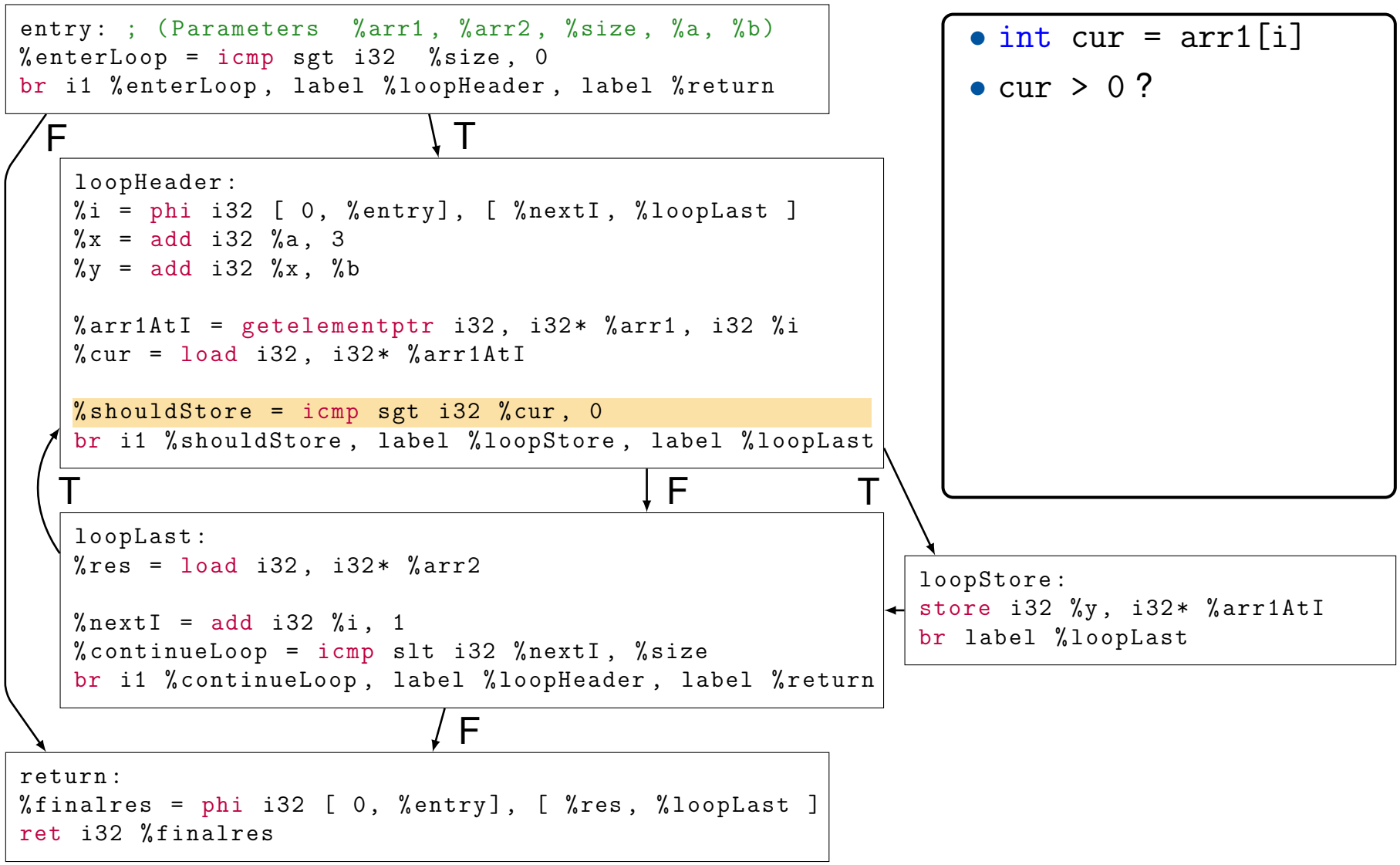
LLVM Intermediate Representation

Semantics (2)



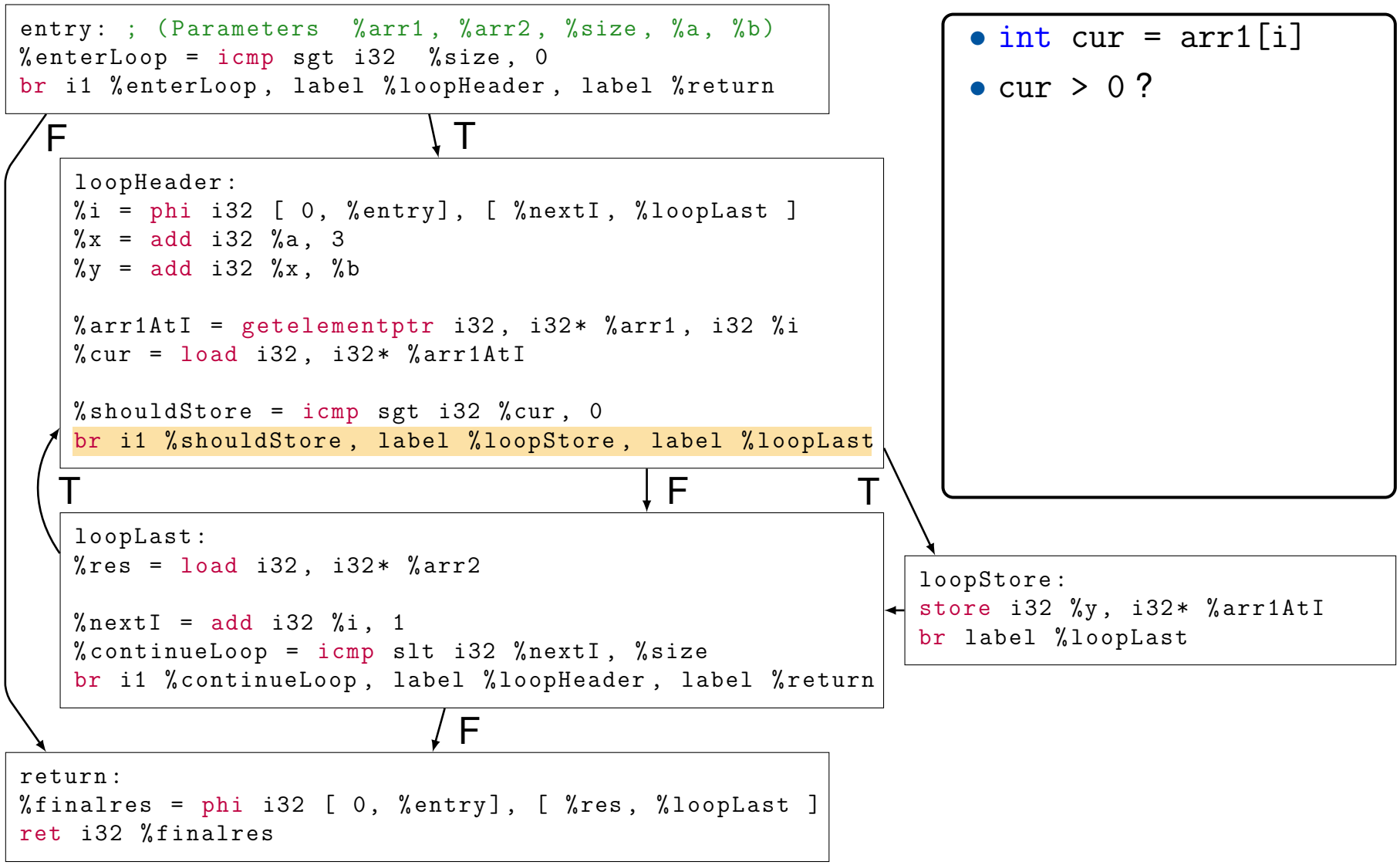
LLVM Intermediate Representation

Semantics (2)



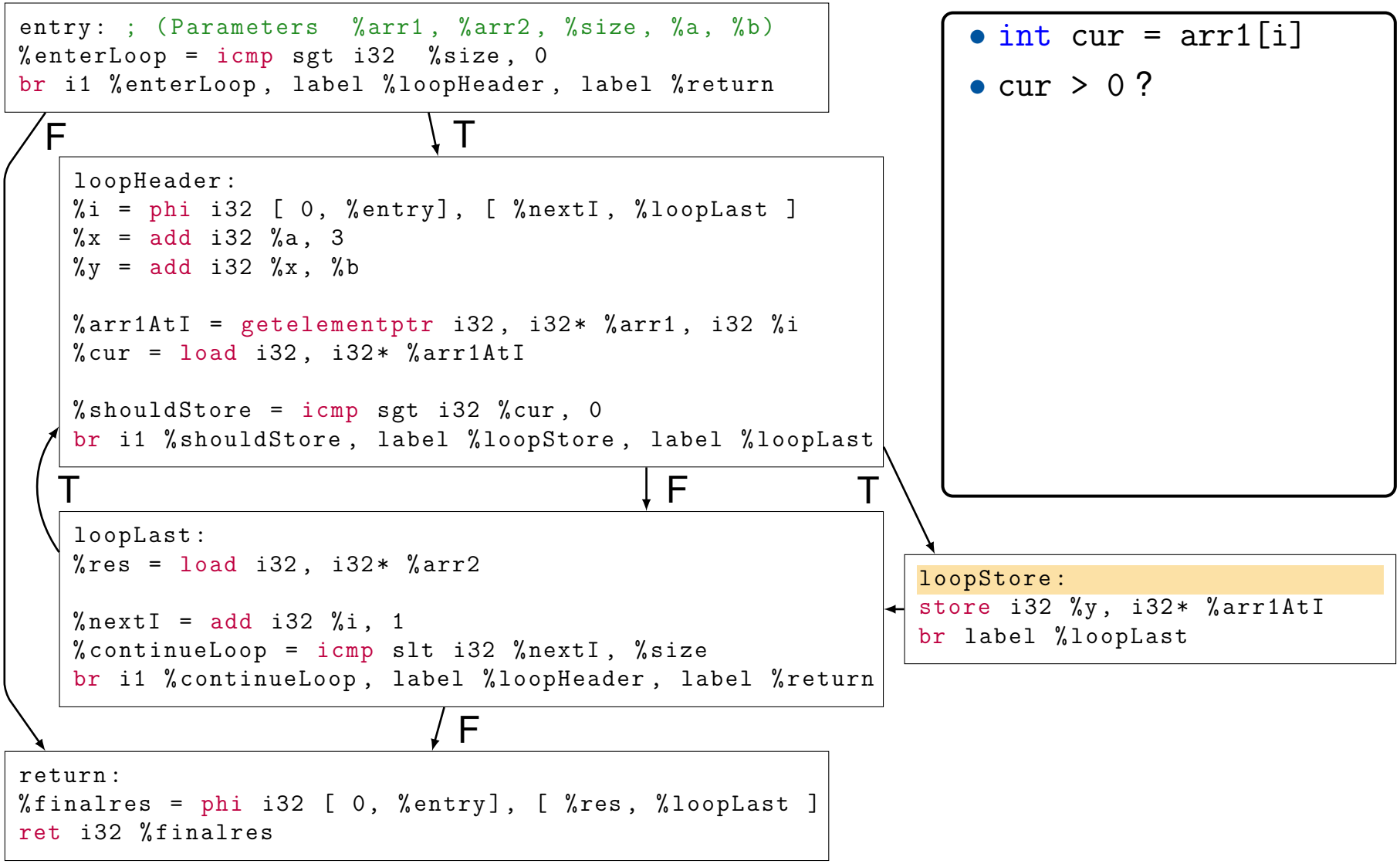
LLVM Intermediate Representation

Semantics (2)



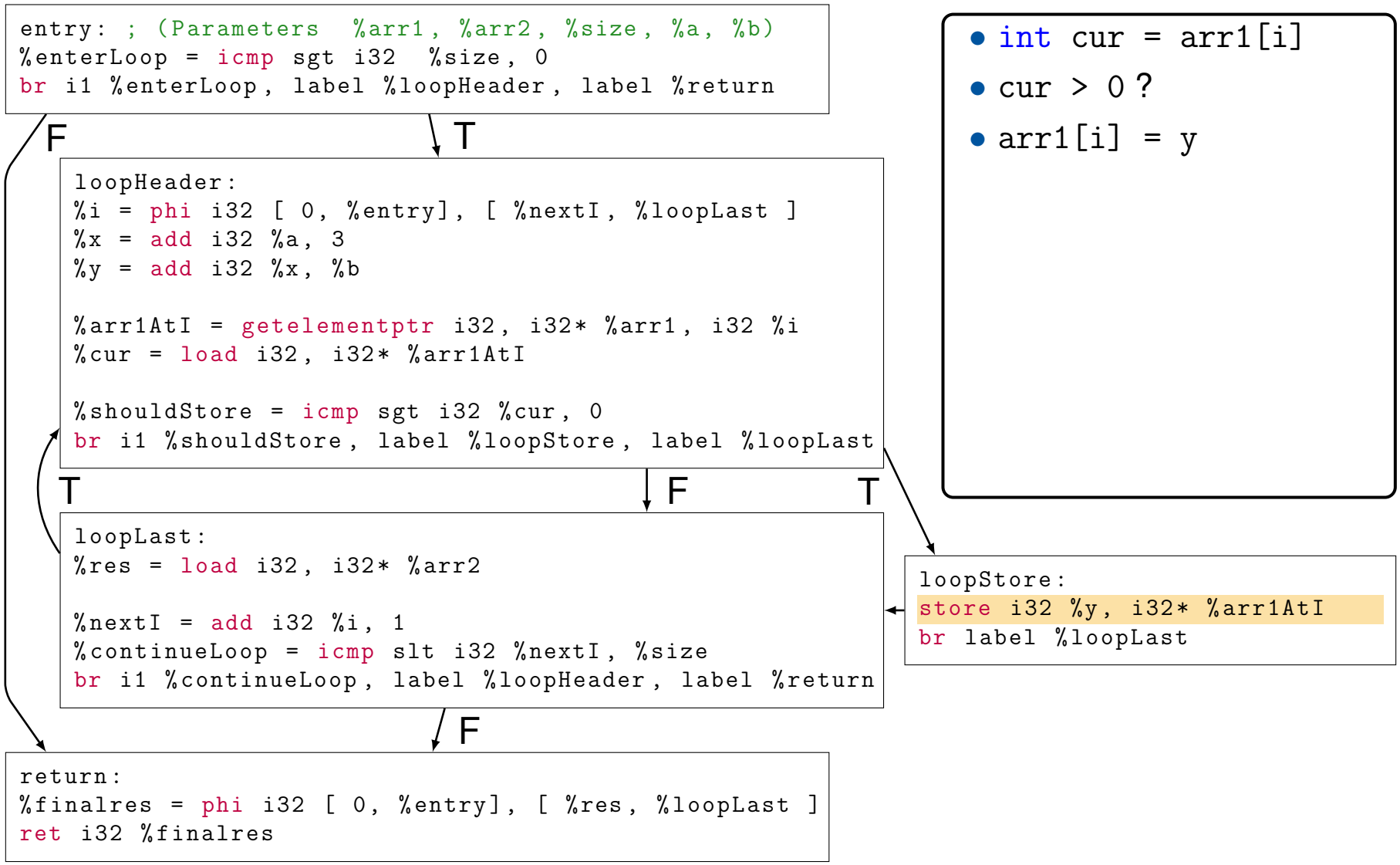
LLVM Intermediate Representation

Semantics (2)



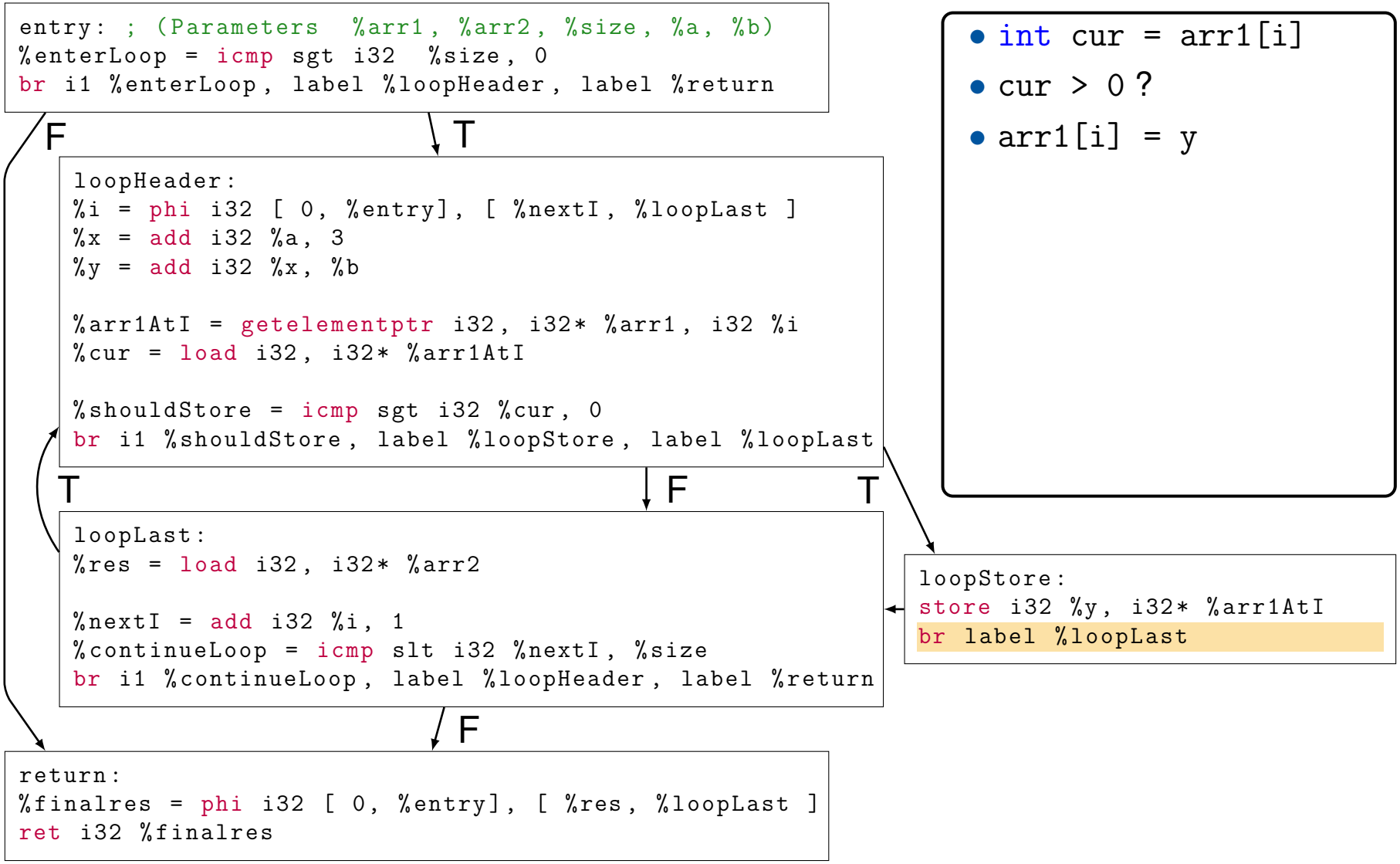
LLVM Intermediate Representation

Semantics (2)



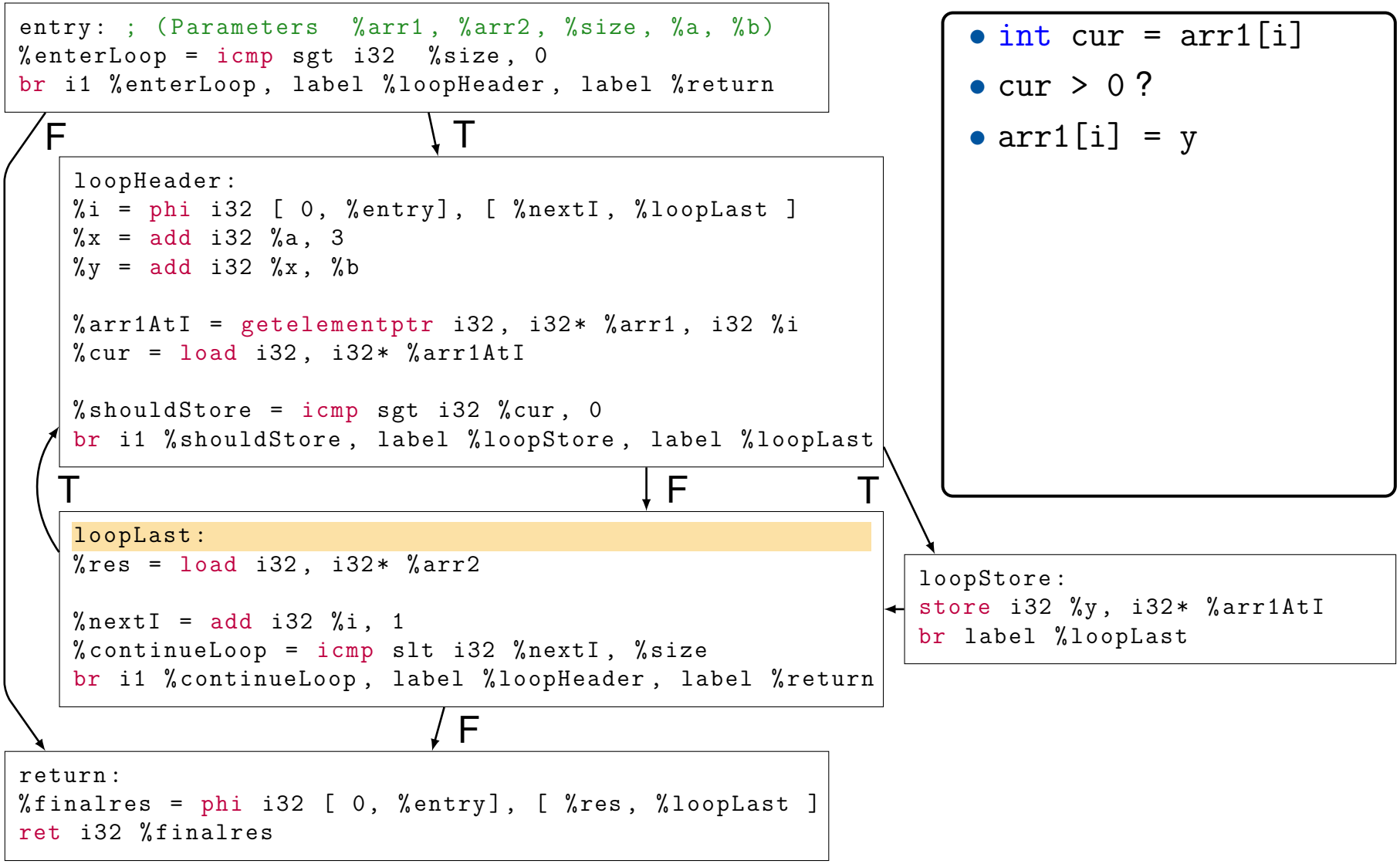
LLVM Intermediate Representation

Semantics (2)



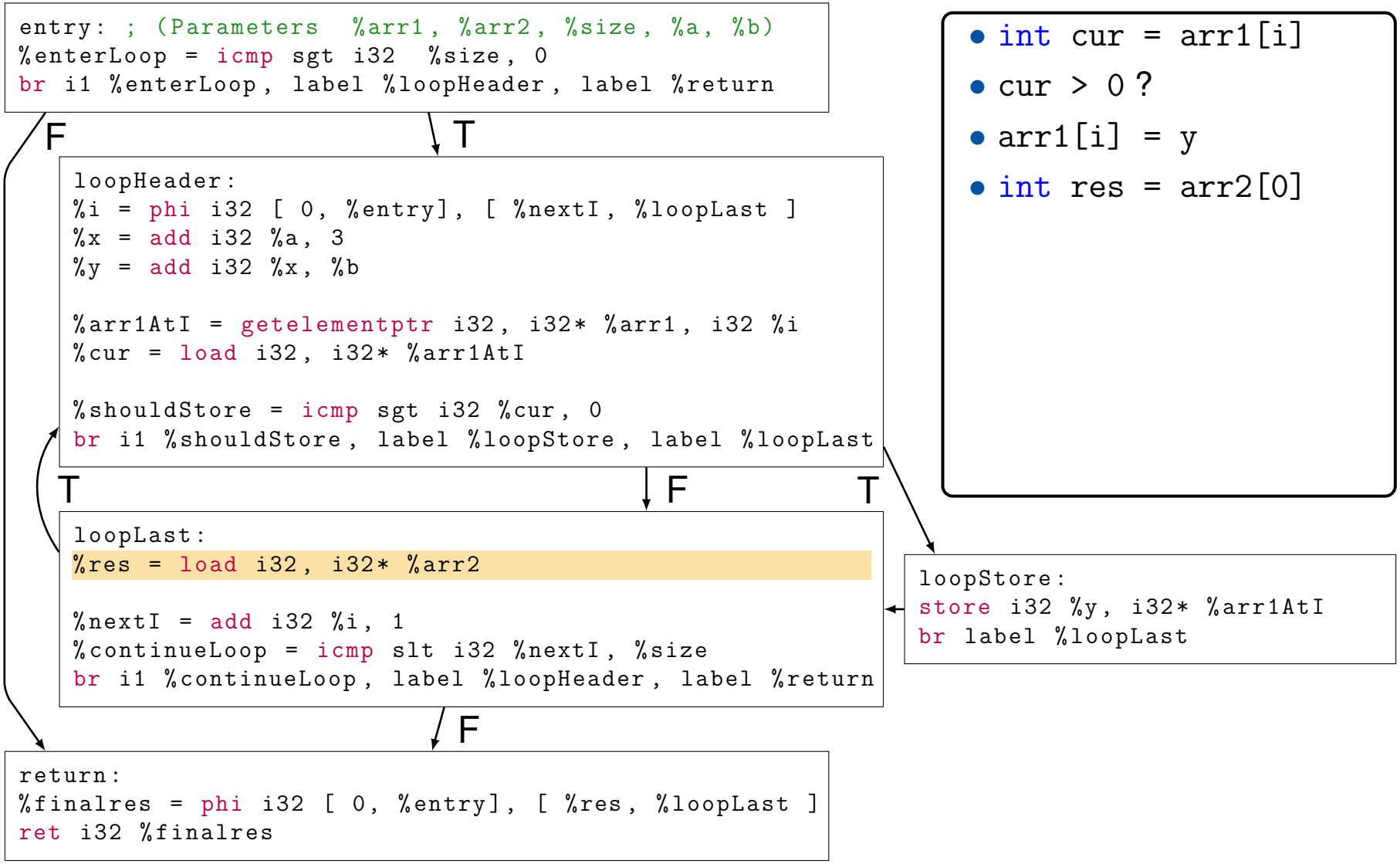
LLVM Intermediate Representation

Semantics (2)



LLVM Intermediate Representation

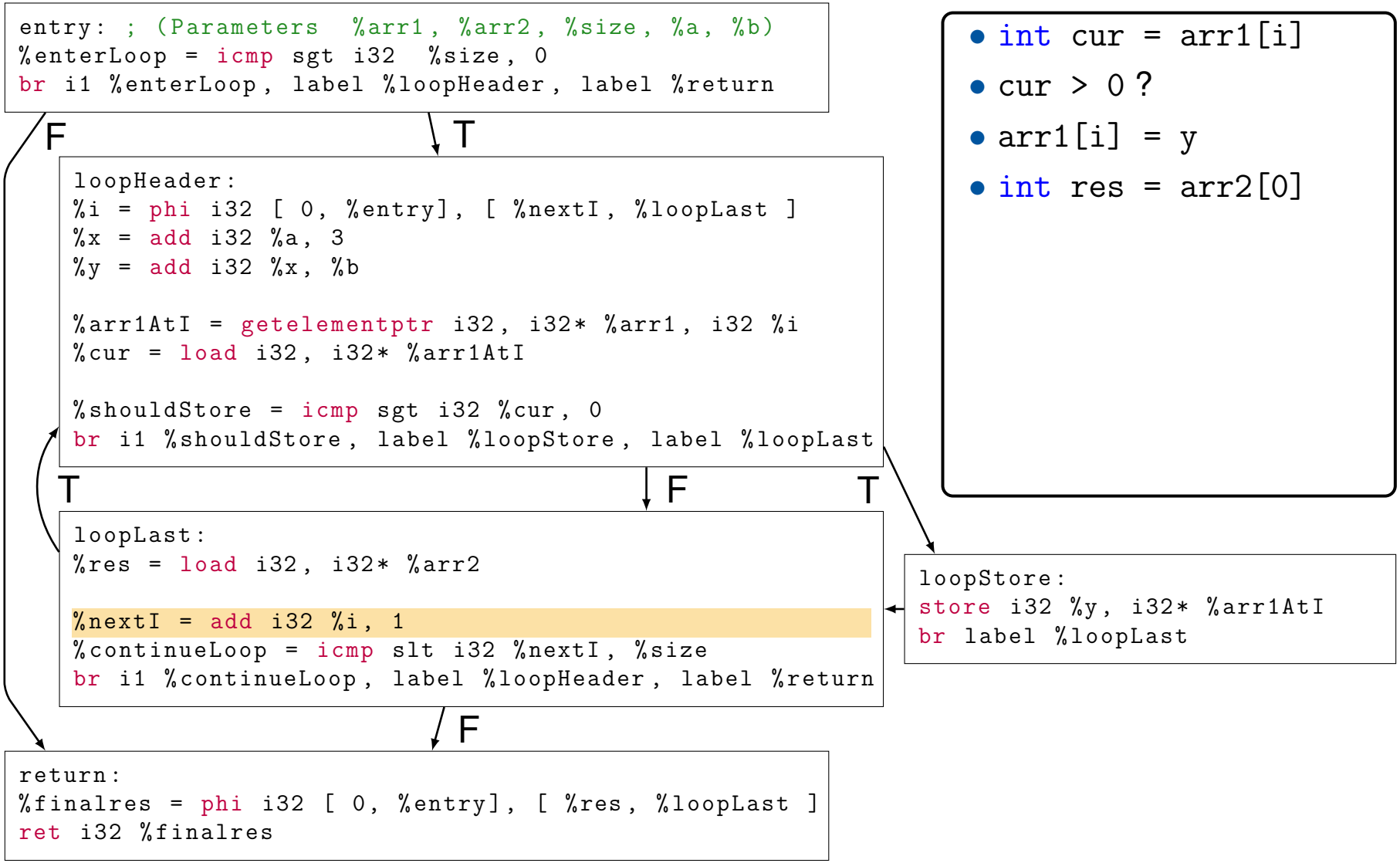
Semantics (2)



- int cur = arr1[i]
- cur > 0 ?
- arr1[i] = y
- int res = arr2[0]

LLVM Intermediate Representation

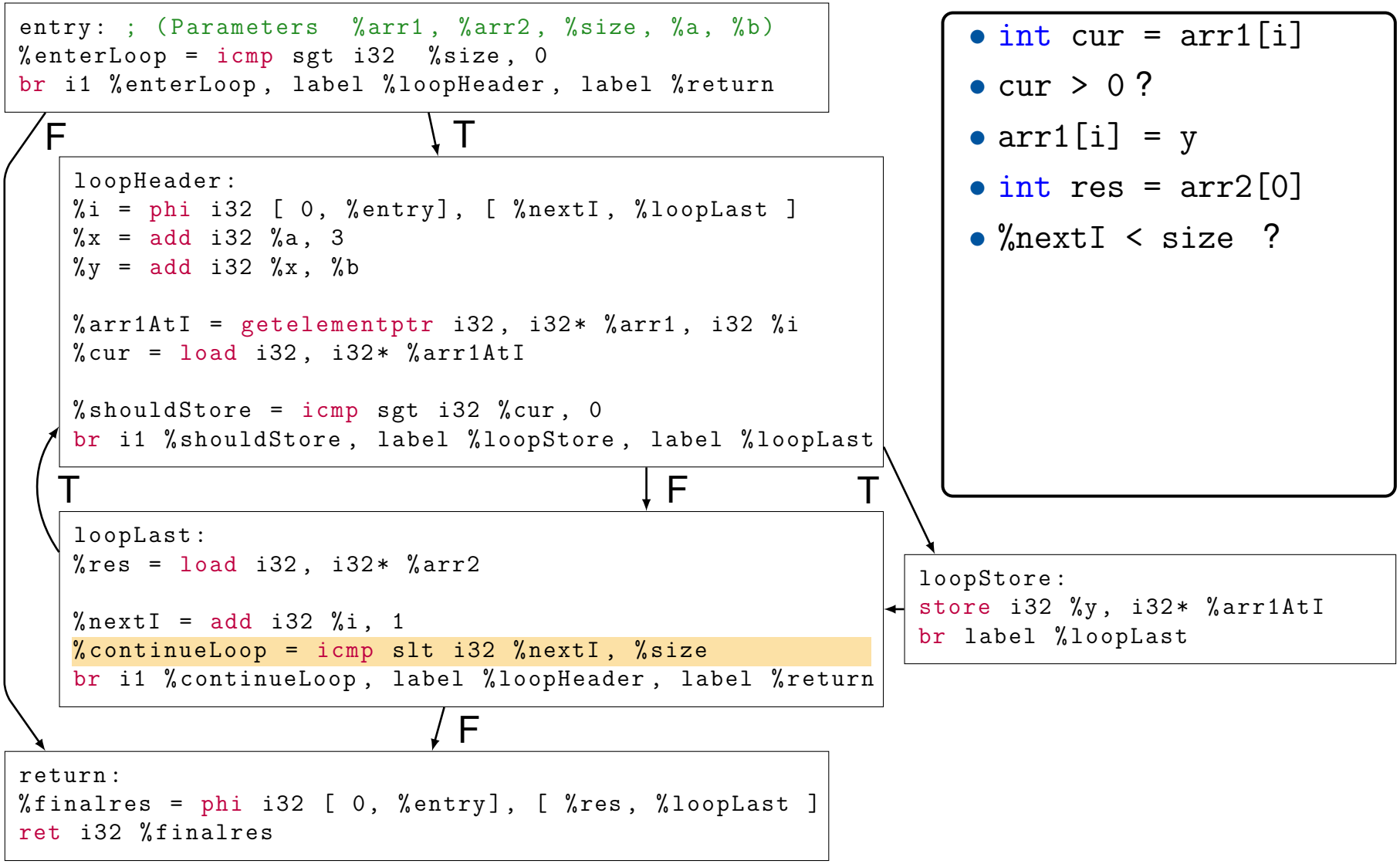
Semantics (2)



- int cur = arr1[i]
- cur > 0 ?
- arr1[i] = y
- int res = arr2[0]

LLVM Intermediate Representation

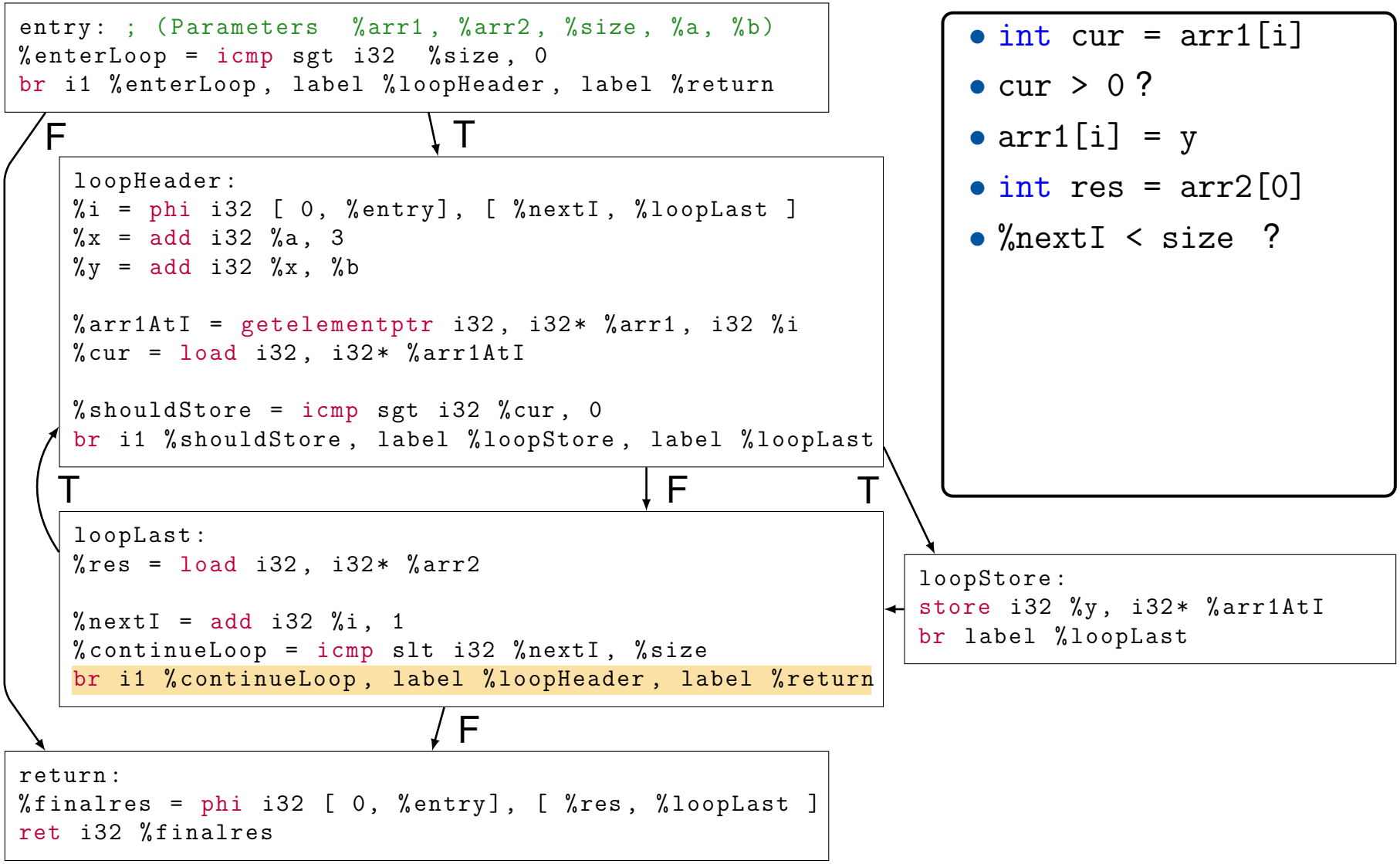
Semantics (2)



- int cur = arr1[i]
- cur > 0 ?
- arr1[i] = y
- int res = arr2[0]
- %nextI < size ?

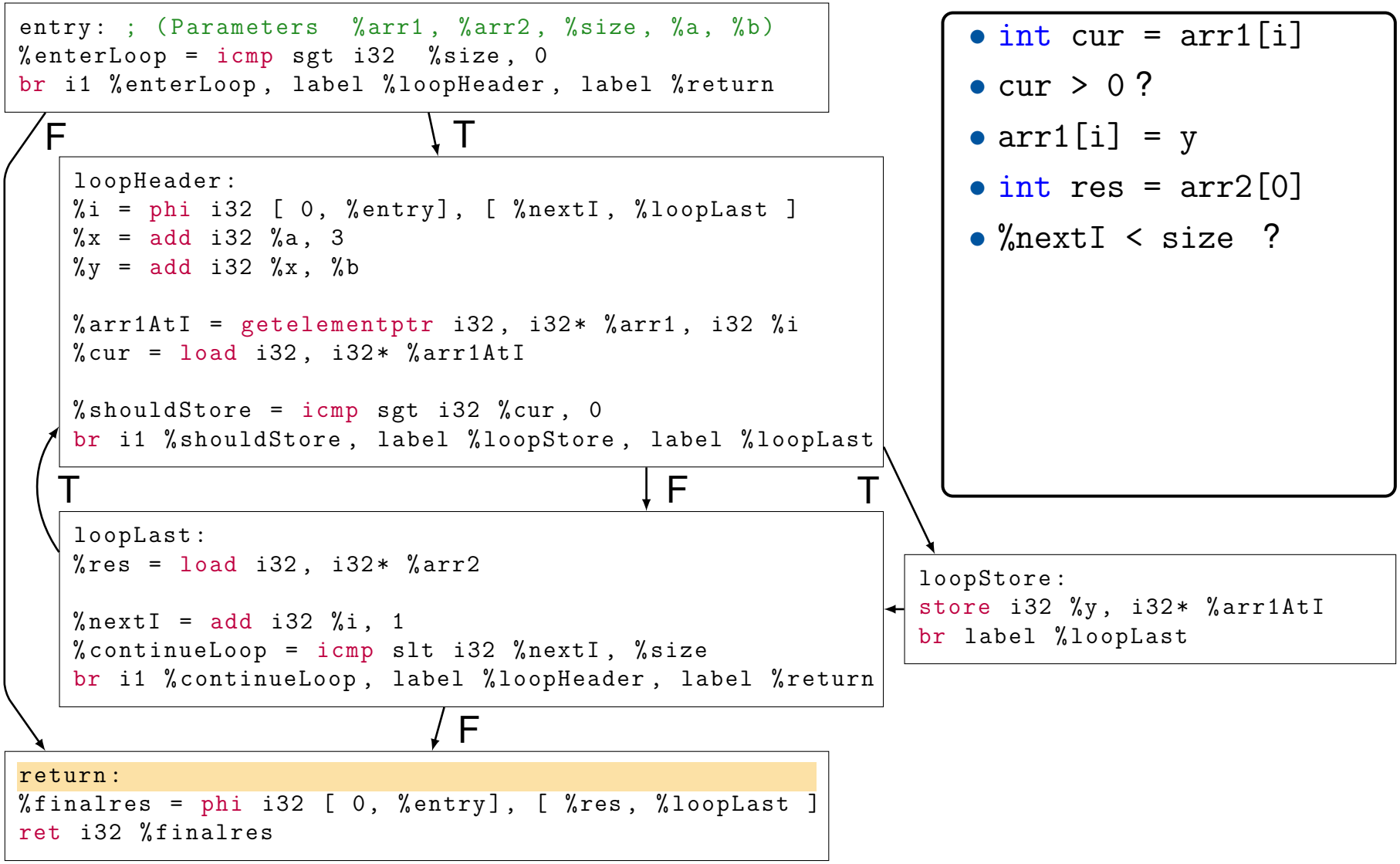
LLVM Intermediate Representation

Semantics (2)



LLVM Intermediate Representation

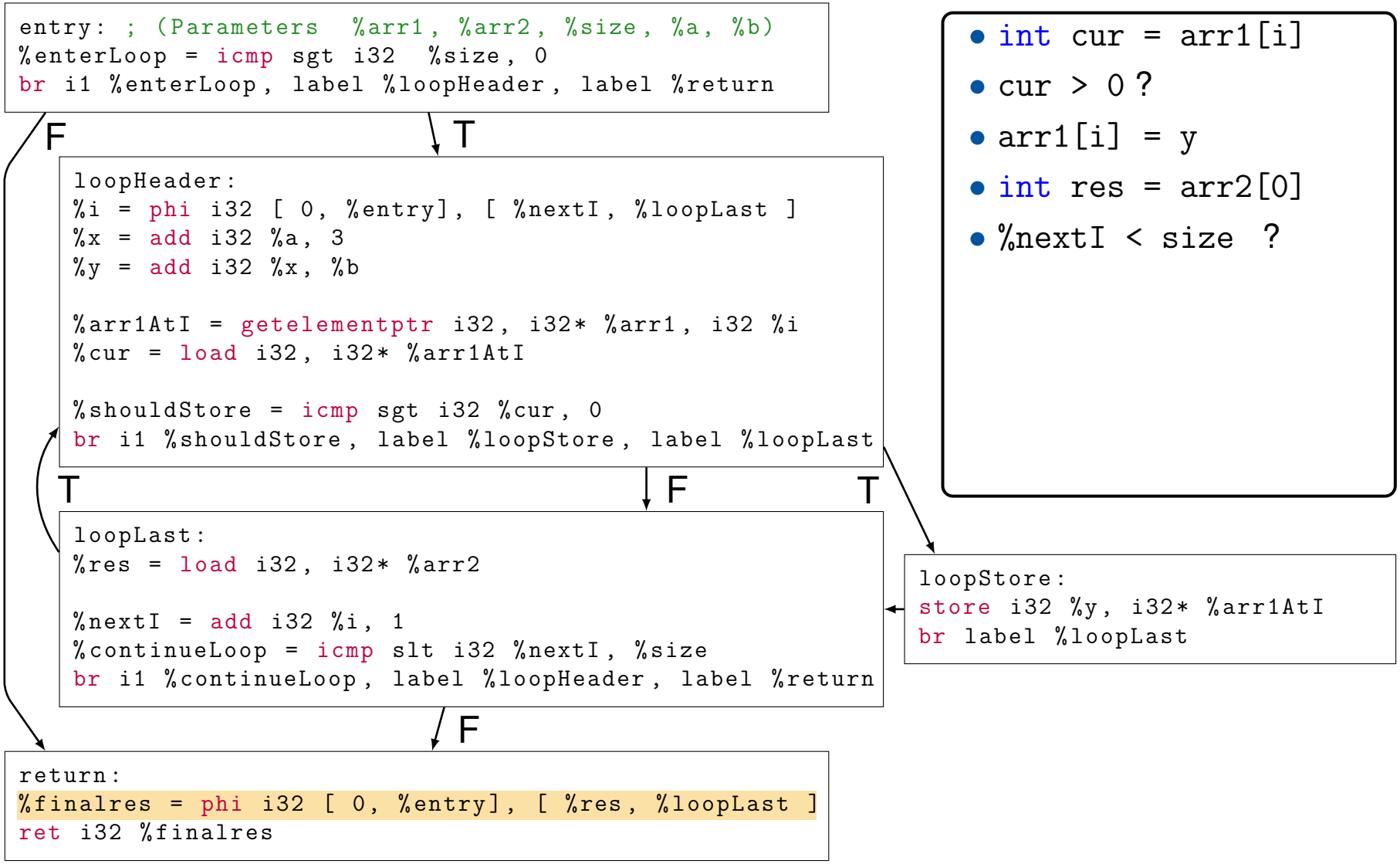
Semantics (2)



- int cur = arr1[i]
- cur > 0 ?
- arr1[i] = y
- int res = arr2[0]
- %nextI < size ?

LLVM Intermediate Representation

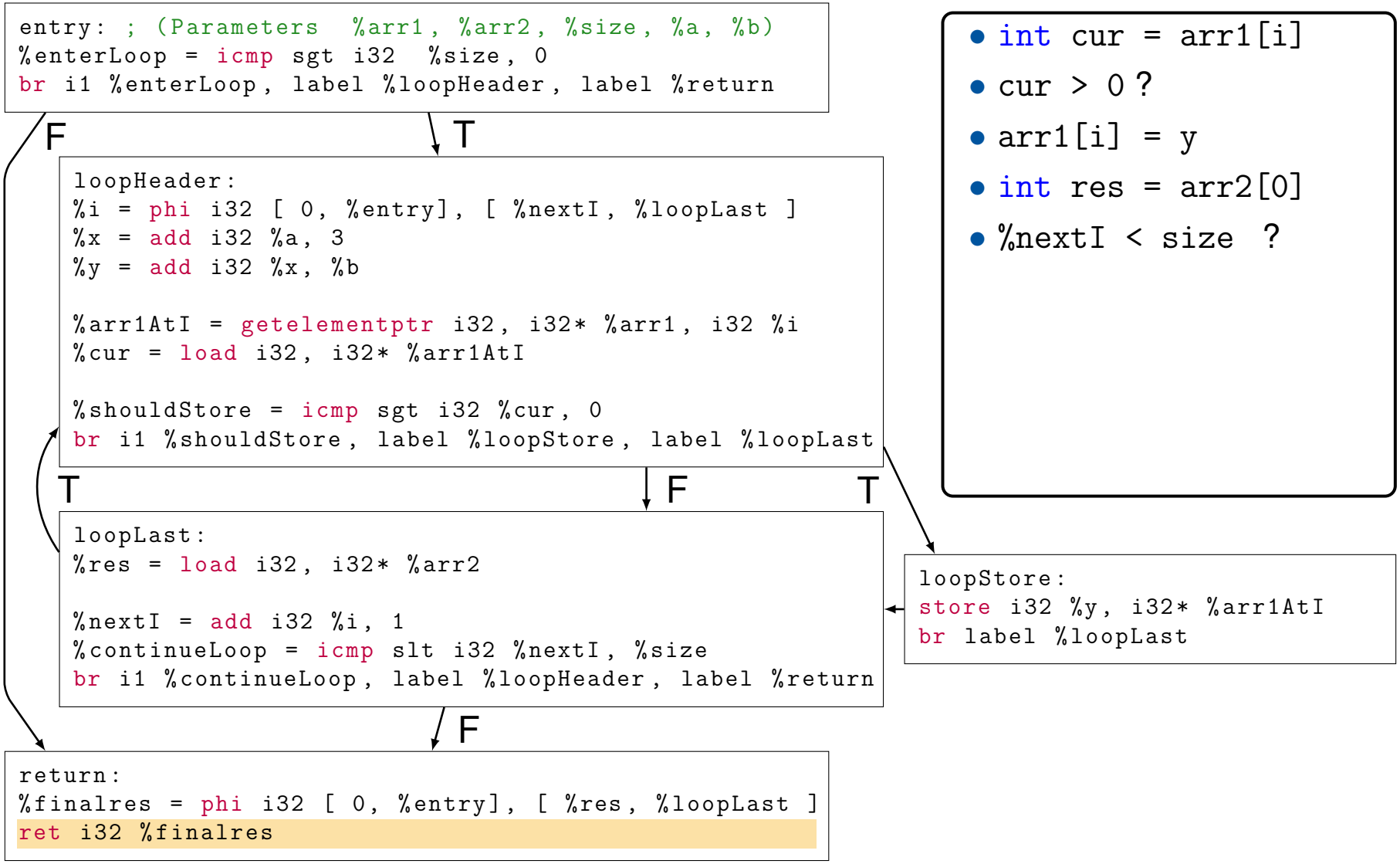
Semantics (2)



- int cur = arr1[i]
- cur > 0 ?
- arr1[i] = y
- int res = arr2[0]
- %nextI < size ?

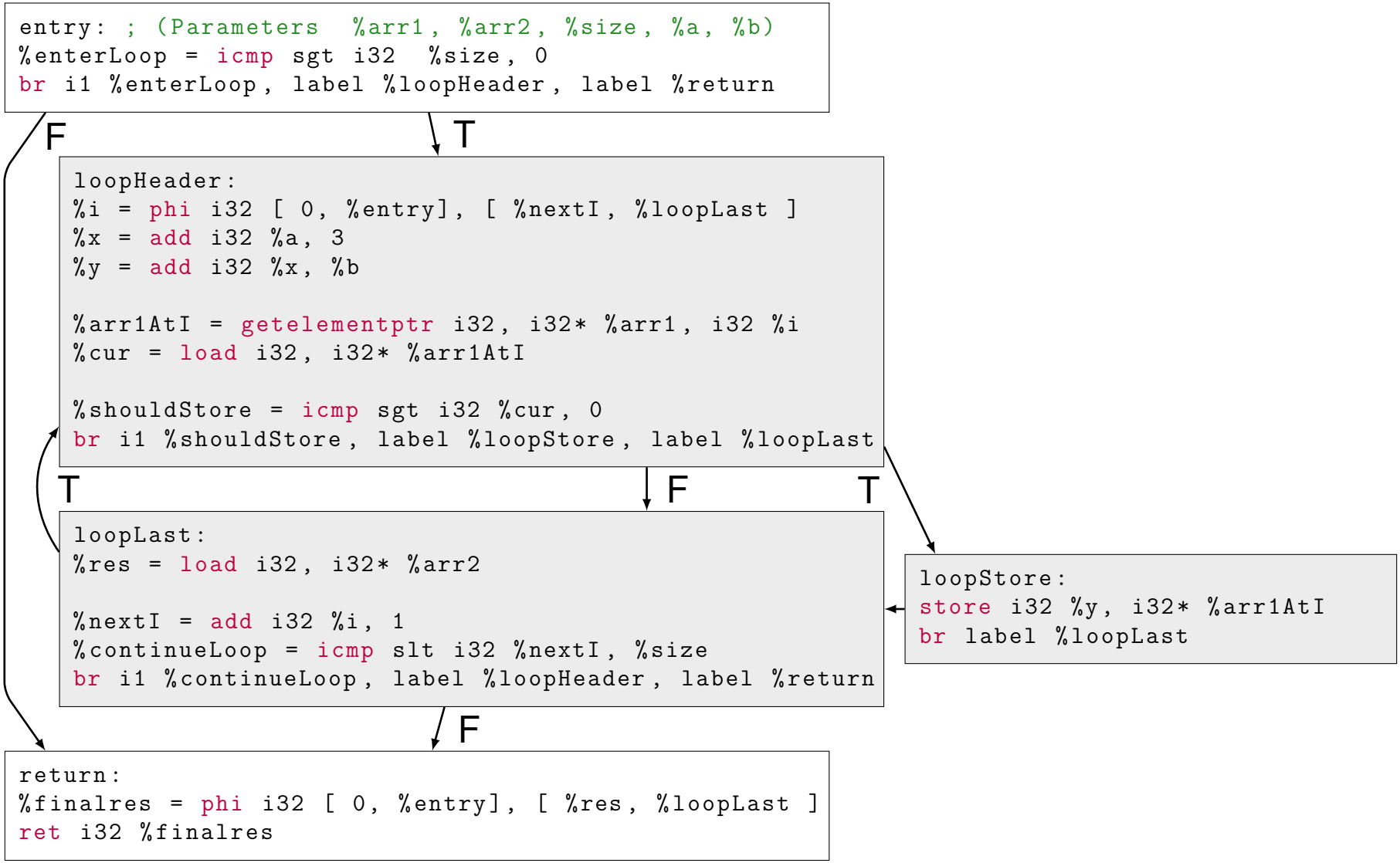
LLVM Intermediate Representation

Semantics (2)

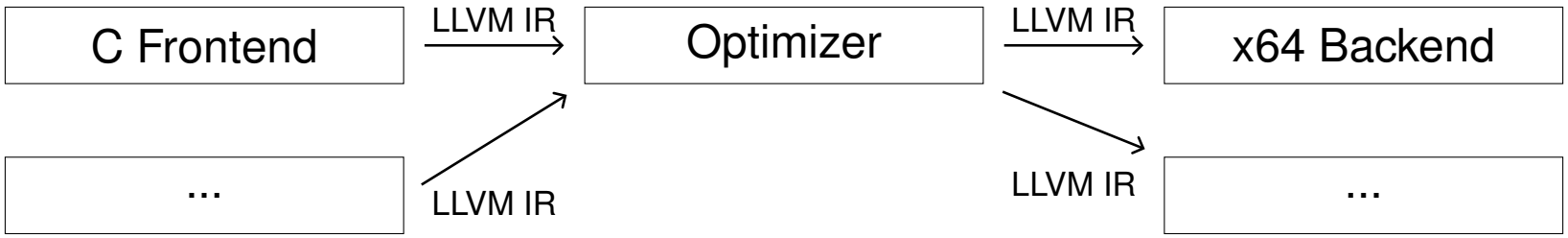


LLVM Intermediate Representation

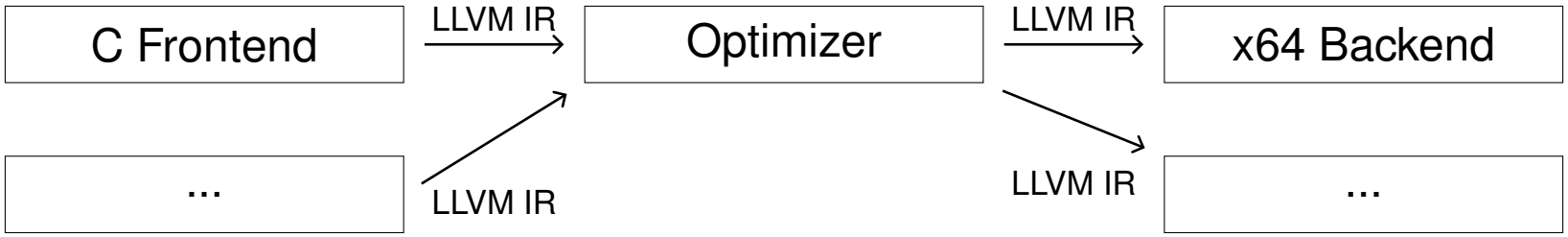
Semantics (3)



LLVM's Optimizer Design

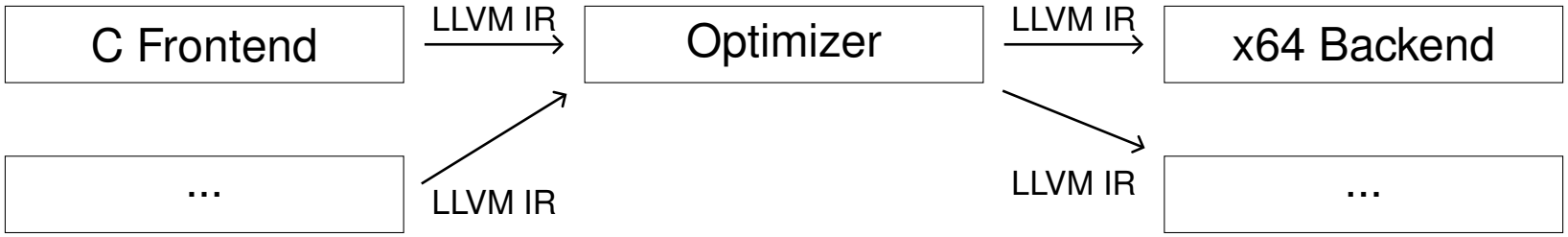


LLVM's Optimizer Design



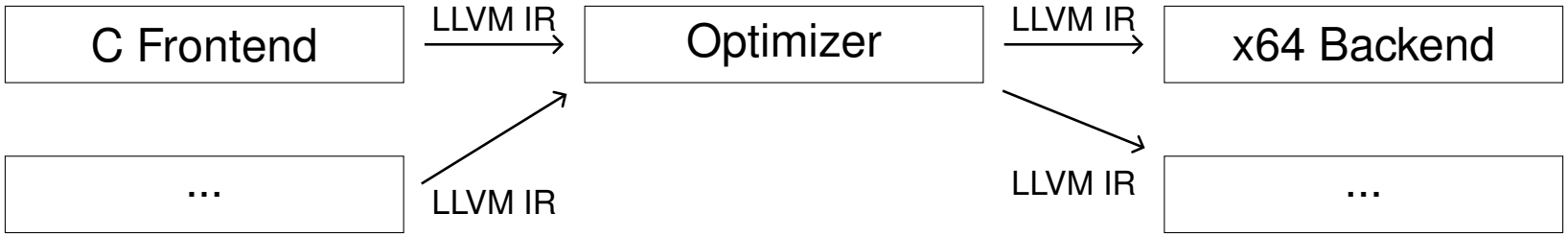
- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)

LLVM's Optimizer Design



- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)
- Modular organization as *passes*

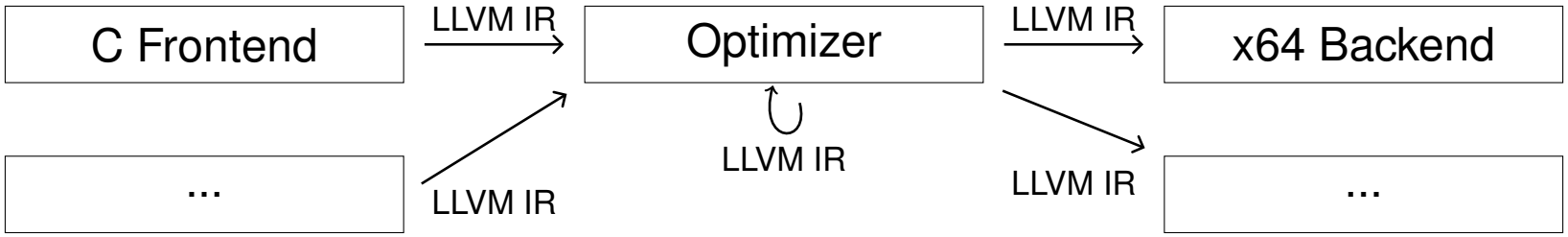
LLVM's Optimizer Design



- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)
- Modular organization as *passes*

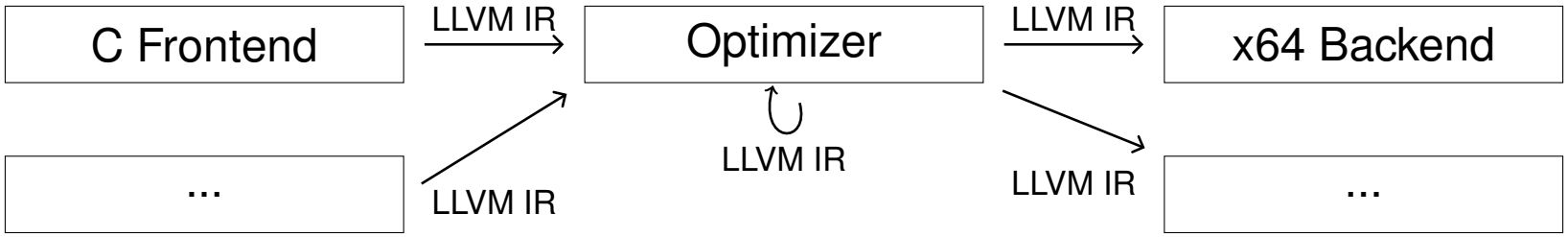
  Transformation Passes Actually optimize code

LLVM's Optimizer Design



- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)
- Modular organization as *passes*
  Transformation Passes Actually optimize code (LLVM IR $\rightarrow$ LLVM IR)

LLVM's Optimizer Design

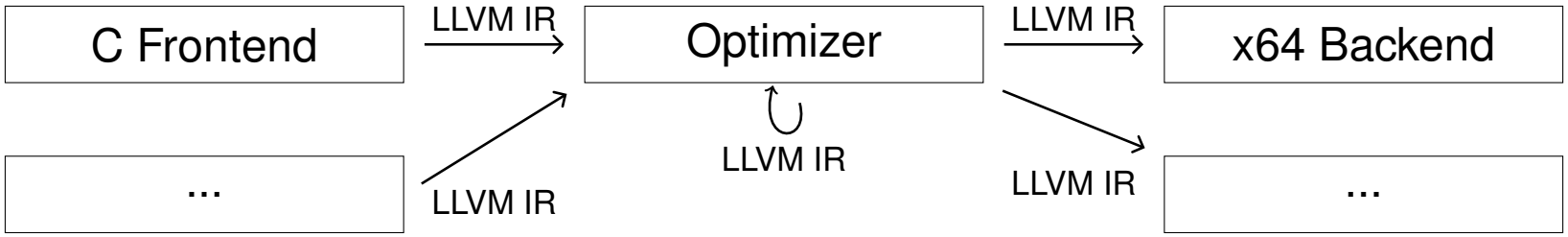


- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)
- Modular organization as *passes*

  Transformation Passes Actually optimize code (LLVM IR $\rightarrow$ LLVM IR)

  Analysis Passes Provide information to other passes (e.g. control flow graph)

LLVM's Optimizer Design

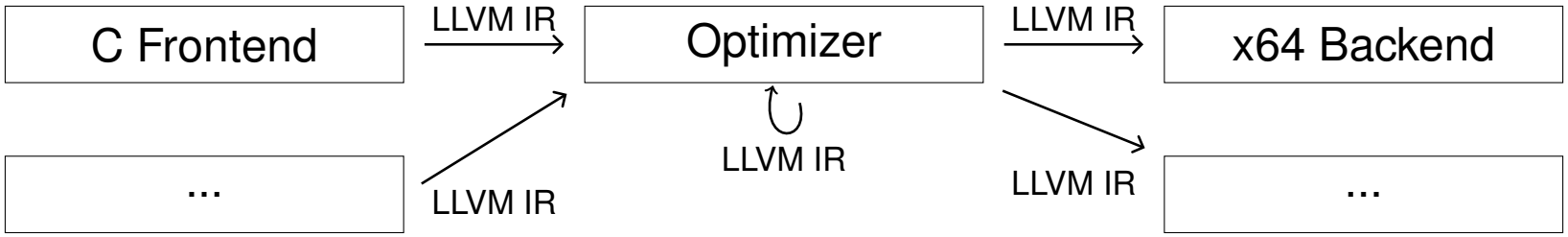


- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)
- Modular organization as *passes*

  Transformation Passes Actually optimize code (LLVM IR $\rightarrow$ LLVM IR)

  Analysis Passes Provide information to other passes (e.g. control flow graph)
- *Pass Manager* is told which passes should be performed and resolves which analysis passes these depend on

LLVM's Optimizer Design



- Variety of optimizations offered:
  (Inline functions, find dead code, pre-compute arithmetic operations on constants, ...)
- Modular organization as *passes*

  Transformation Passes Actually optimize code (LLVM IR $\rightarrow$ LLVM IR)

  Analysis Passes Provide information to other passes (e.g. control flow graph)
- *Pass Manager* is told which passes should be performed and resolves which analysis passes these depend on
- We look at the *Loop Invariant Code Motion* transformation pass

Loop Invariant Code Motion

- “Invariant” = Same result in each iteration

Loop Invariant Code Motion

- “Invariant” = Same result in each iteration
- Thus should do computation once outside loop

Loop Invariant Code Motion

- “Invariant” = Same result in each iteration
- Thus should do computation once outside loop
- Loop Invariant Code Motion: Move invariant code out of loops

Loop Invariant Code Motion

- “Invariant” = Same result in each iteration
- Thus should do computation once outside loop
- Loop Invariant Code Motion: Move invariant code out of loops

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];
   }
   return res;
}
```

Loop Invariant Code Motion

- “Invariant” = Same result in each iteration
- Thus should do computation once outside loop
- Loop Invariant Code Motion: Move invariant code out of loops

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];
   }
   return res;
}
```

Loop Invariant Code Motion

- “Invariant” = Same result in each iteration
- Thus should do computation once outside loop
- Loop Invariant Code Motion: Move invariant code out of loops

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];
   }
   return res;
}
```

- Questions:
  1. How to detect invariant code in general?
  2. Where to move such code in general?

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)
- Expression x is invariant in some loop iff

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)
- Expression x is invariant in some loop iff
  1. x is a literal

Loop Invariant Code Motion

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)
- Expression x is invariant in some loop iff
  1. x is a literal
  2. x is a variable defined before the loop

Loop Invariant Code Motion

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)
- Expression x is invariant in some loop iff
  1. x is a literal
  2. x is a variable defined before the loop
  3. x is the result of an instruction all of whose operands are invariant

Loop Invariant Code Motion

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)
- Expression x is invariant in some loop iff
  1. x is a literal
  2. x is a variable defined before the loop
  3. x is the result of an instruction all of whose operands are invariant
- We call the instruction in condition 3 an *invariant instruction*

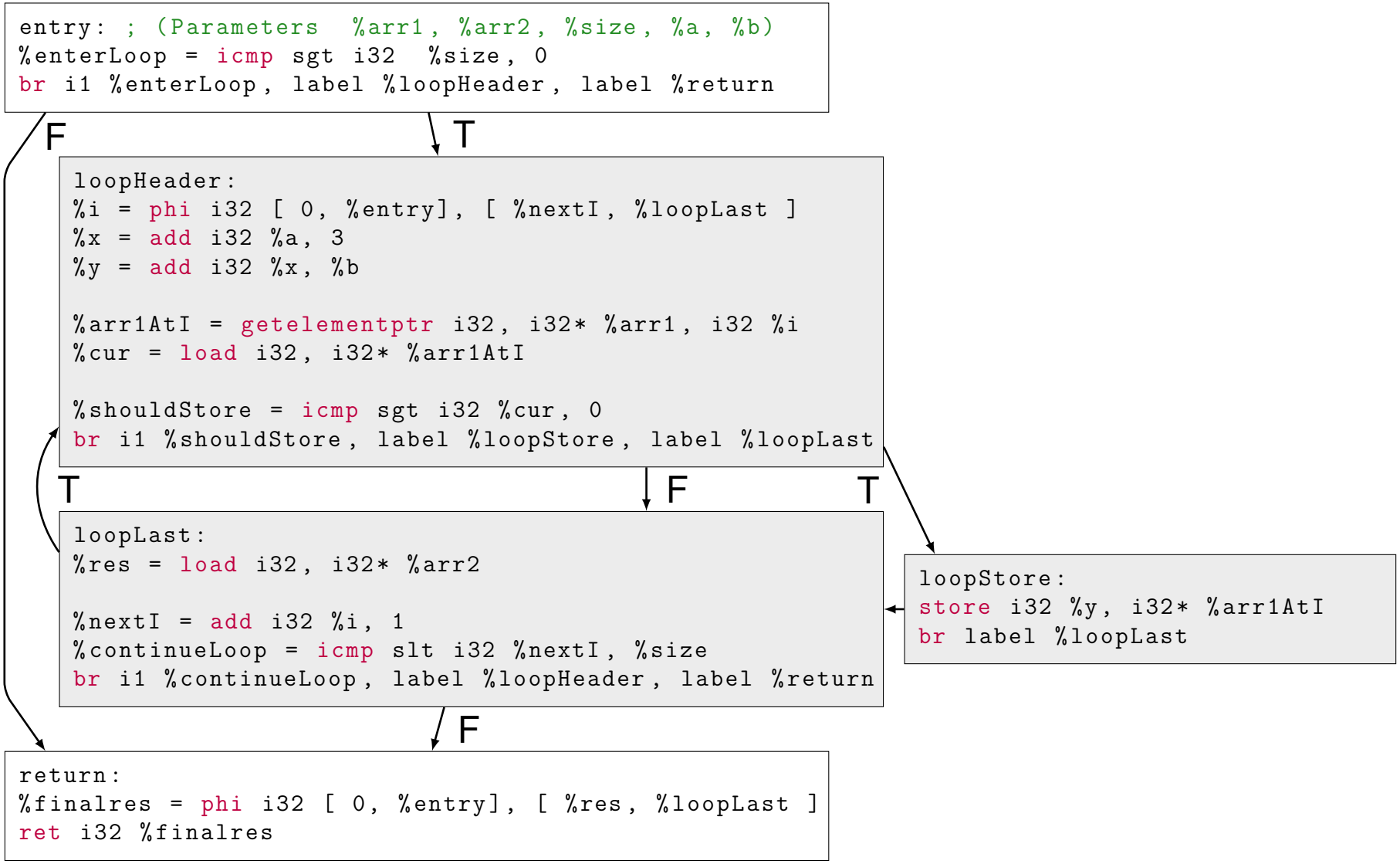
Loop Invariant Code Motion

Detecting Invariant Code

- We restrict ourselves to arithmetic instructions here (e.g. `%x = add i32 %a, 3`)
- Expression x is invariant in some loop iff
  1. x is a literal
  2. x is a variable defined before the loop
  3. x is the result of an instruction all of whose operands are invariant
- We call the instruction in condition 3 an *invariant instruction*
- These conditions are sufficient to move invariant instructions ahead of the loop (“Hoisting”)

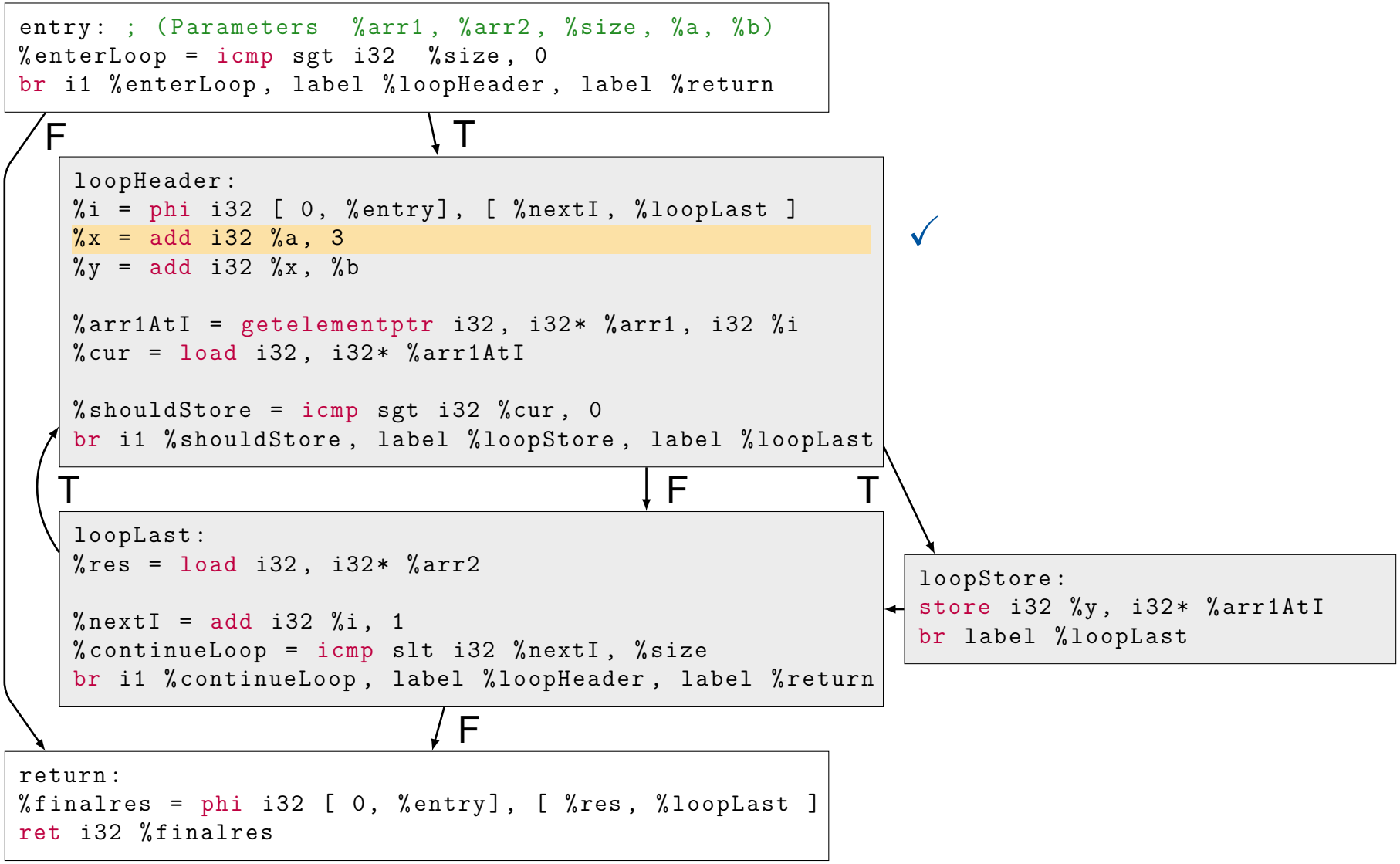
Loop Invariant Code Motion

Example Detecting Invariant Code



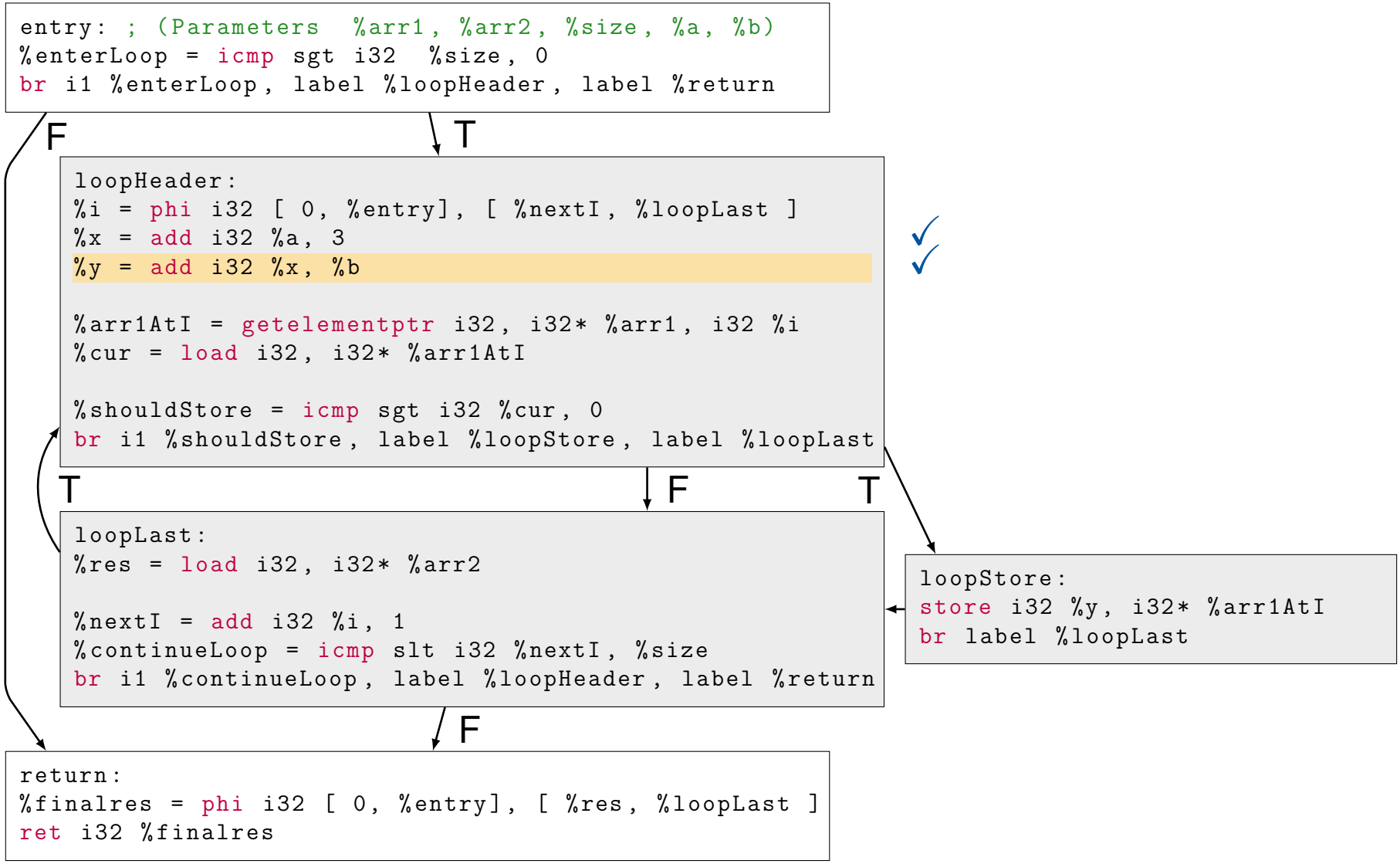
Loop Invariant Code Motion

Example Detecting Invariant Code



Loop Invariant Code Motion

Example Detecting Invariant Code



Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header

Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:

Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:
  1. Moved instructions may be executed even if loop skipped

Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:
  1. Moved instructions may be executed even if loop skipped
  2. Possibly multiple predecessors

Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:
  1. Moved instructions may be executed even if loop skipped
  2. Possibly multiple predecessors
- Solution: Create block that is executed once before entering loop

Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:
  1. Moved instructions may be executed even if loop skipped
  2. Possibly multiple predecessors
- Solution: Create block that is executed once before entering loop
  - Called “preheader”

Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:
  1. Moved instructions may be executed even if loop skipped
  2. Possibly multiple predecessors
- Solution: Create block that is executed once before entering loop
  - Called “preheader”
  - Only successor is the header of the loop

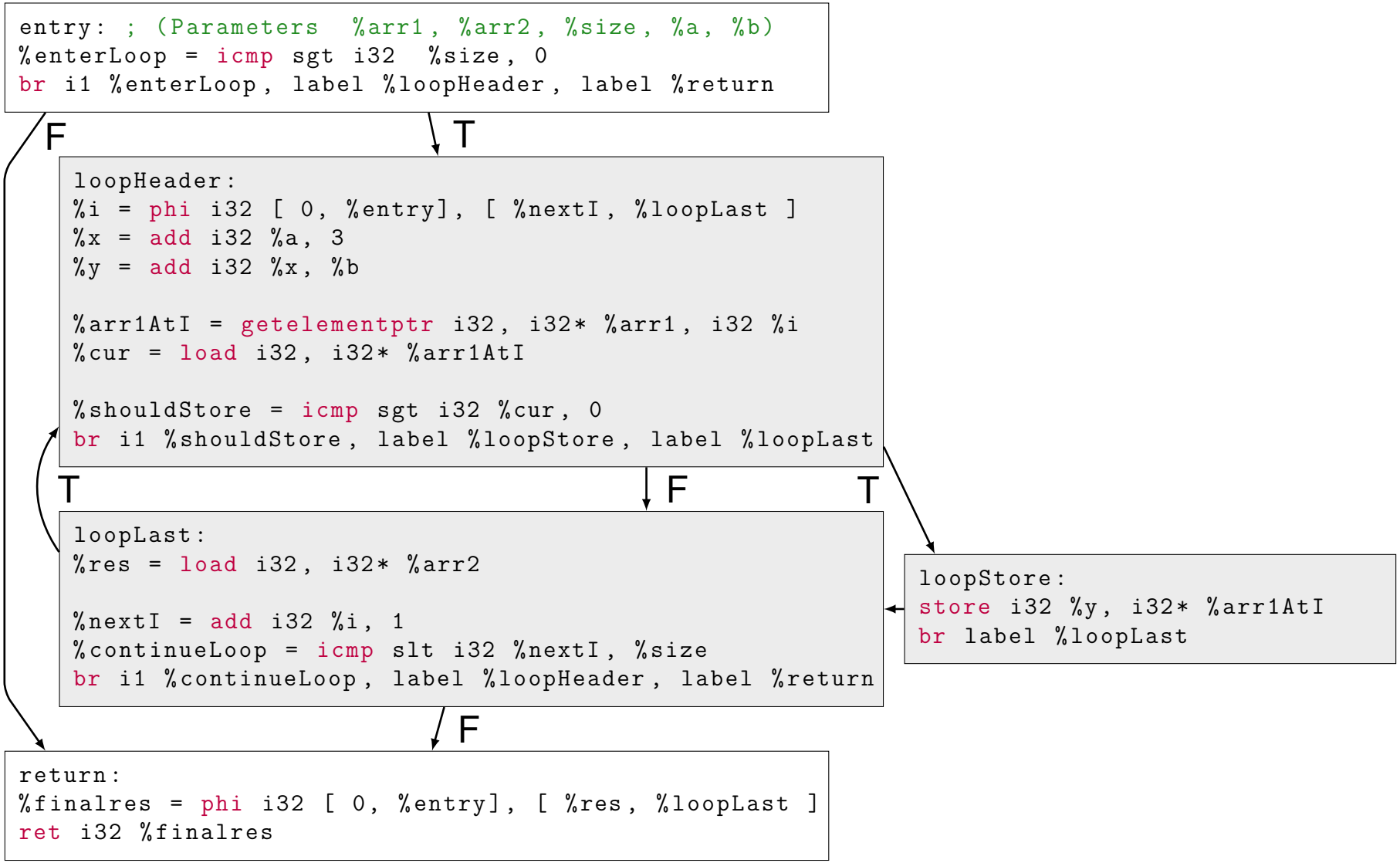
Loop Invariant Code Motion

Hoisting Invariant Code

- First idea: Hoist code to predecessor of header
- Problems:
  1. Moved instructions may be executed even if loop skipped
  2. Possibly multiple predecessors
- Solution: Create block that is executed once before entering loop
  - Called “preheader”
  - Only successor is the header of the loop
  - All incoming edges from outside of the loop to the header point to the preheader instead

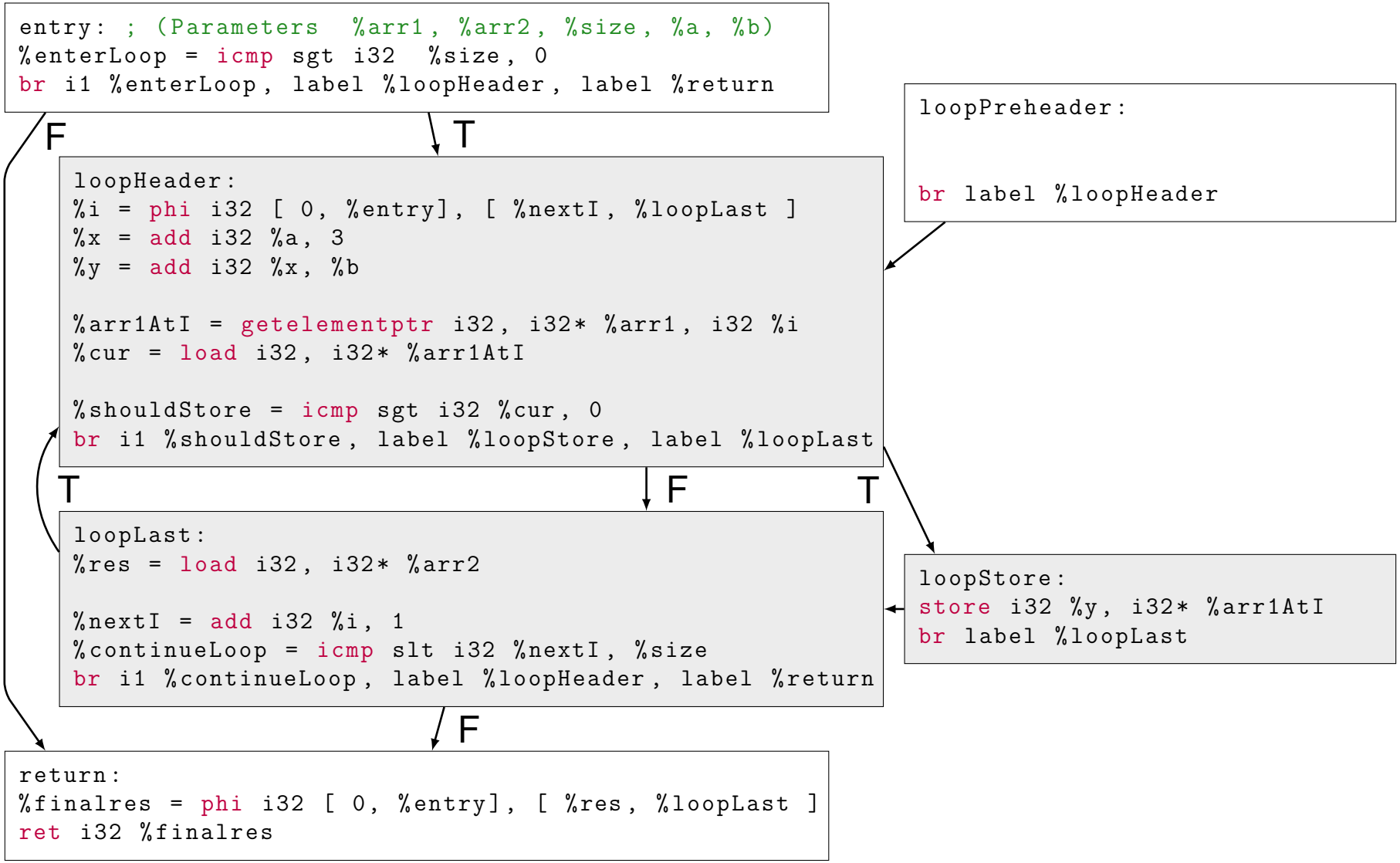
Loop Invariant Code Motion

Example Hoisting Invariant Code



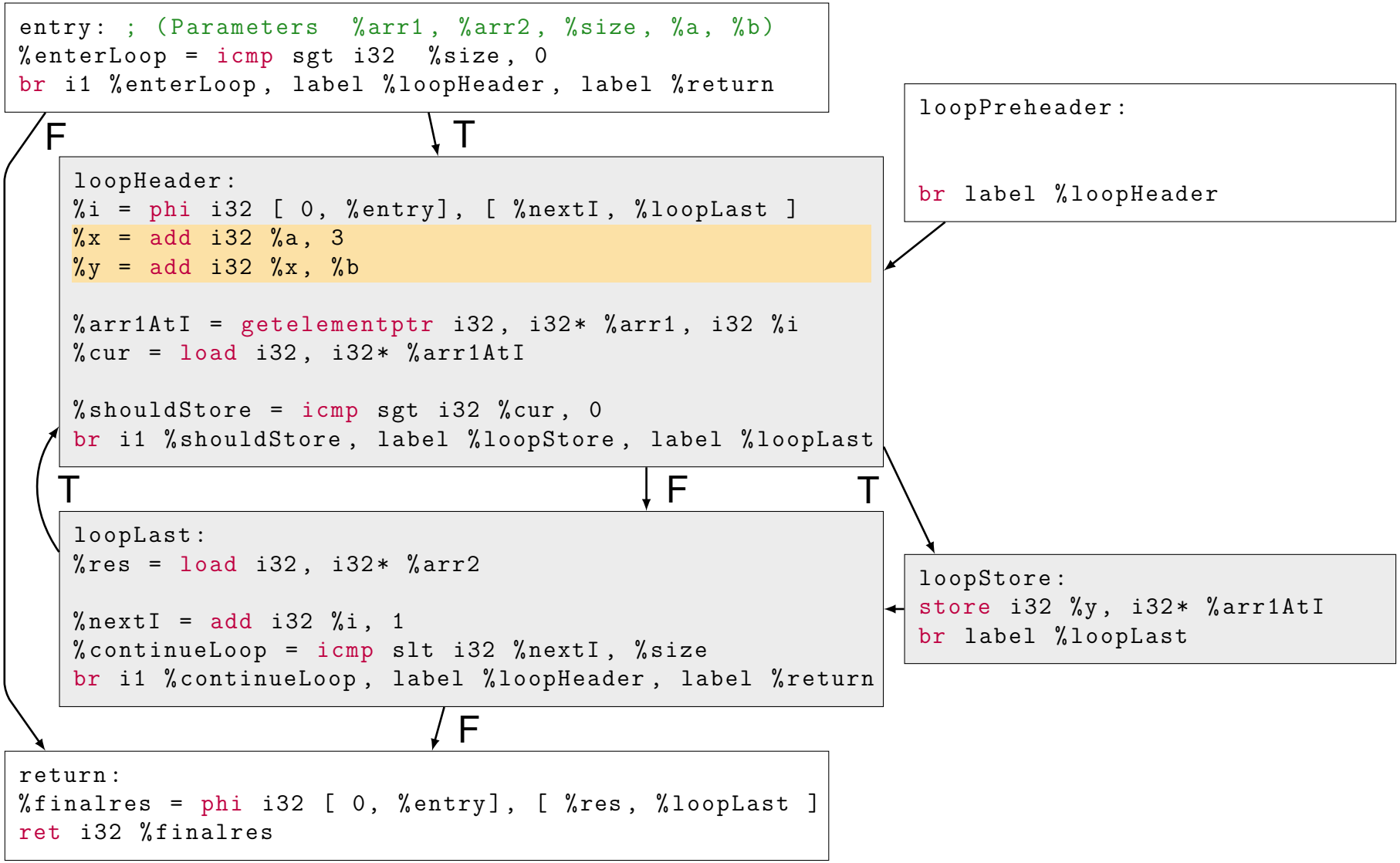
Loop Invariant Code Motion

Example Hoisting Invariant Code



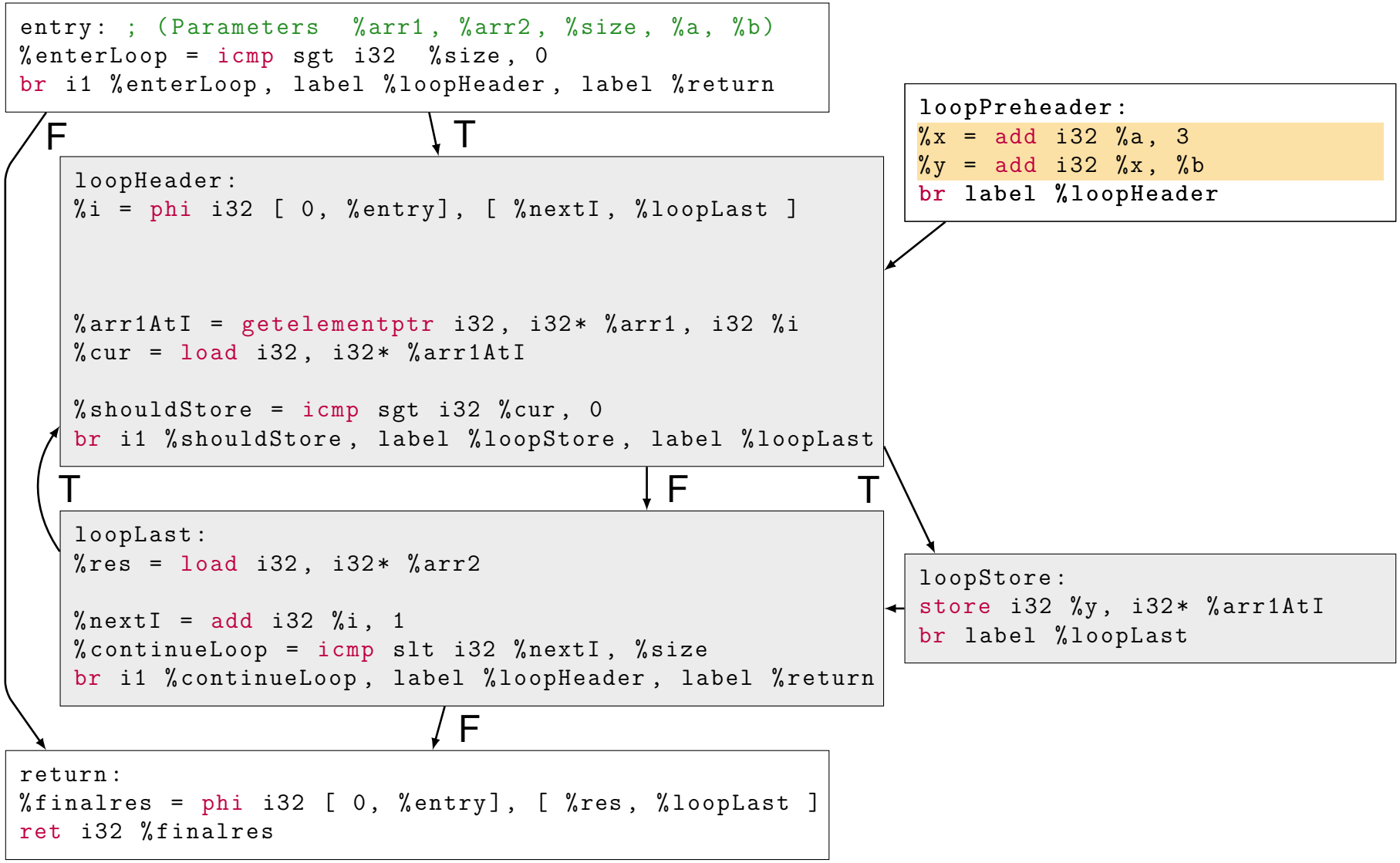
Loop Invariant Code Motion

Example Hoisting Invariant Code



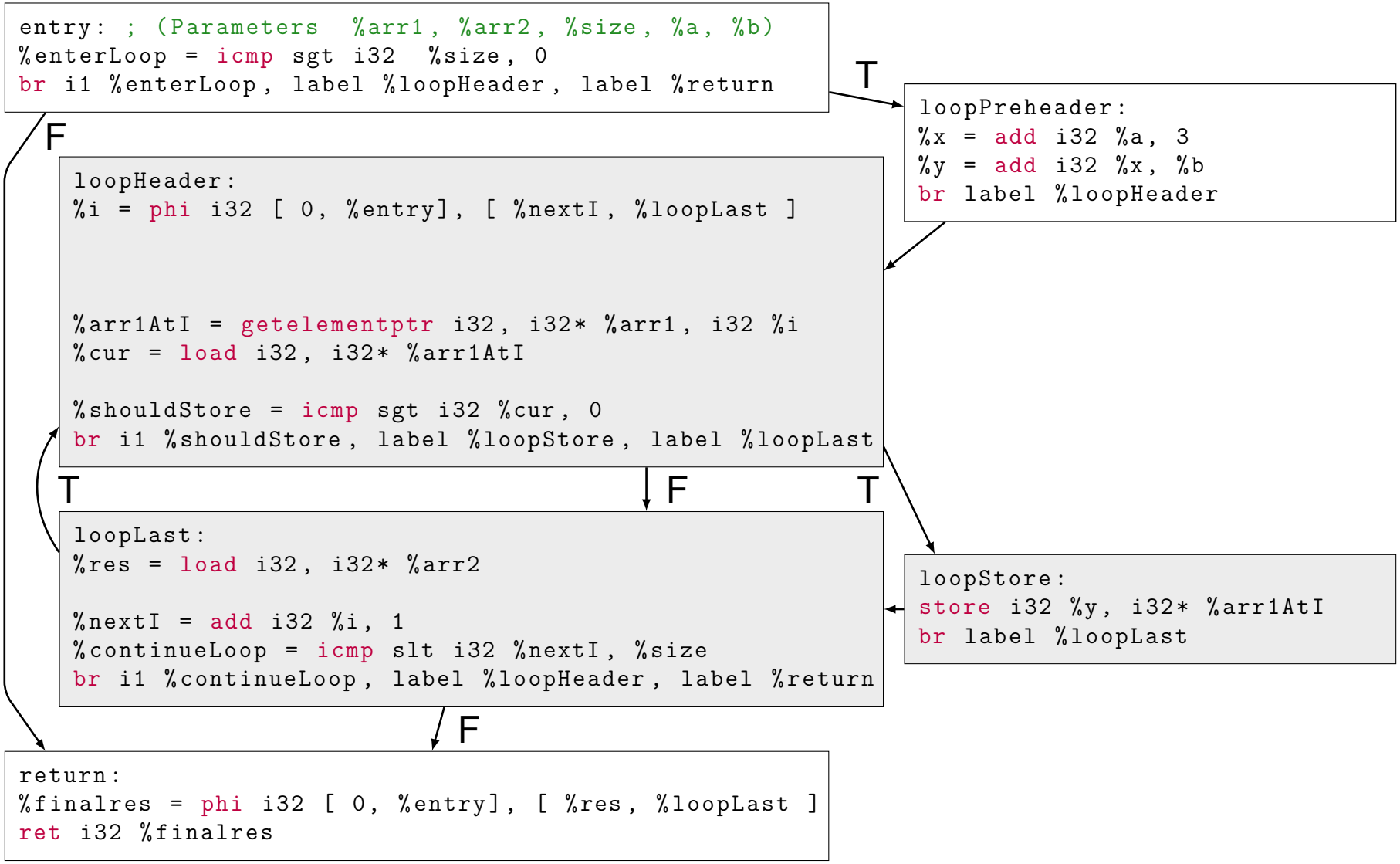
Loop Invariant Code Motion

Example Hoisting Invariant Code



Loop Invariant Code Motion

Example Hoisting Invariant Code



Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:

Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:
  Hoisting Move invariant code before loop

Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:

  Hoisting Move invariant code before loop

  Sinking Move invariant code behind loop
  (only if variable defined by invariant instruction not used in loop)

Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:

  Hoisting Move invariant code before loop

  Sinking Move invariant code behind loop

  (only if variable defined by invariant instruction not used in loop)

- Nested loops are processed from inner loops to outer loops

Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:

  Hoisting Move invariant code before loop

  Sinking Move invariant code behind loop

  (only if variable defined by invariant instruction not used in loop)
- Nested loops are processed from inner loops to outer loops
- Supports moving function calls, if they are side effect free

Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:

  Hoisting Move invariant code before loop

  Sinking Move invariant code behind loop

  (only if variable defined by invariant instruction not used in loop)
- Nested loops are processed from inner loops to outer loops
- Supports moving function calls, if they are side effect free
- Supports moving `load` and `store` instructions

Loop Invariant Code Motion

- Supports hoisting and sinking of invariant instructions:

  Hoisting Move invariant code before loop

  Sinking Move invariant code behind loop

  (only if variable defined by invariant instruction not used in loop)
- Nested loops are processed from inner loops to outer loops
- Supports moving function calls, if they are side effect free
- Supports moving `load` and `store` instructions
- How to handle these memory-sensitive instructions?

Loop Invariant Code Motion

- We can hoist `load` instructions if

Loop Invariant Code Motion

- We can hoist `load` instructions if
  1. The address loaded from is invariant

Loop Invariant Code Motion

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

Loop Invariant Code Motion

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];
   }
   return res;
}
```

Loop Invariant Code Motion

Outlook: Handling Memory

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];
   }
   return res;
}
```

Loop Invariant Code Motion

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];                    %res = load i32, i32* %arr2
   }
   return res;
}
```

Loop Invariant Code Motion

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];                %res = load i32, i32* %arr2
   }
   return res;
}
```

- We could hoist the load from `arr2`

Loop Invariant Code Motion

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];
      if(cur > 0) {
         arr1[i] = y;
      }

      res = arr2[0];                %res = load i32, i32* %arr2
   }
   return res;
}
```

- We could hoist the load from `arr2` ...?

Loop Invariant Code Motion

Outlook: Handling Memory

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];            %arr1AtI = getelementptr i32, i32* %arr1, i32 %i
      if(cur > 0) {                 ...
         arr1[i] = y;               store i32 %y, i32* %arr1AtI
      }

      res = arr2[0];                %res = load i32, i32* %arr2
   }
   return res;
}
```

- We could hoist the load from `arr2` ...?
- No, because `arr1` (resp. `arr1AtI`) and `arr2` may overlap

Loop Invariant Code Motion

Outlook: Handling Memory

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];            %arr1AtI = getelementptr i32, i32* %arr1, i32 %i
      if(cur > 0) {                 ...
         arr1[i] = y;               store i32 %y, i32* %arr1AtI
      }

      res = arr2[0];                %res = load i32, i32* %arr2
   }
   return res;
}
```

- We could hoist the load from `arr2` ...?
- No, because `arr1` (resp. `arr1AtI`) and `arr2` may overlap
- The Loop Invariant Code Motion pass depends on *aliasing information*: Which pointers may share?

Loop Invariant Code Motion

Outlook: Handling Memory

- We can hoist `load` instructions if
  1. The address loaded from is invariant
  2. No `store` to that address in the loop

```
int example(int *arr1, int *arr2, int size, int a, int b) {
   int res = 0;
   for(int i = 0; i < size; i++) {
      int x = a + 3;
      int y = x + b;

      int cur = arr1[i];            %arr1AtI = getelementptr i32, i32* %arr1, i32 %i
      if(cur > 0) {                 ...
         arr1[i] = y;               store i32 %y, i32* %arr1AtI
      }

      res = arr2[0];                %res = load i32, i32* %arr2
   }
   return res;
}
```

- We could hoist the load from `arr2` ...?
- No, because `arr1` (resp. `arr1AtI`) and `arr2` may overlap
- The Loop Invariant Code Motion pass depends on *aliasing information*: Which pointers may share?
- Provided by an analysis pass executed beforehand

Conclusion

- LLVM Compiler Infrastructure provides optimizer and backends for building compilers

Conclusion

- LLVM Compiler Infrastructure provides optimizer and backends for building compilers
- LLVM IR: Assembly-like language used as internal representation of whole program at different stages of compilation

Conclusion

- LLVM Compiler Infrastructure provides optimizer and backends for building compilers
- LLVM IR: Assembly-like language used as internal representation of whole program at different stages of compilation
- Optimizer works in passes, either transforming the program or providing information to other passes

Conclusion

- LLVM Compiler Infrastructure provides optimizer and backends for building compilers
- LLVM IR: Assembly-like language used as internal representation of whole program at different stages of compilation
- Optimizer works in passes, either transforming the program or providing information to other passes
- Loop Invariant Code Motion attempts to move code out of loops that should just be executed once

Conclusion

- LLVM Compiler Infrastructure provides optimizer and backends for building compilers
- LLVM IR: Assembly-like language used as internal representation of whole program at different stages of compilation
- Optimizer works in passes, either transforming the program or providing information to other passes
- Loop Invariant Code Motion attempts to move code out of loops that should just be executed once
- Moving arithmetic instructions is straightforward

Conclusion

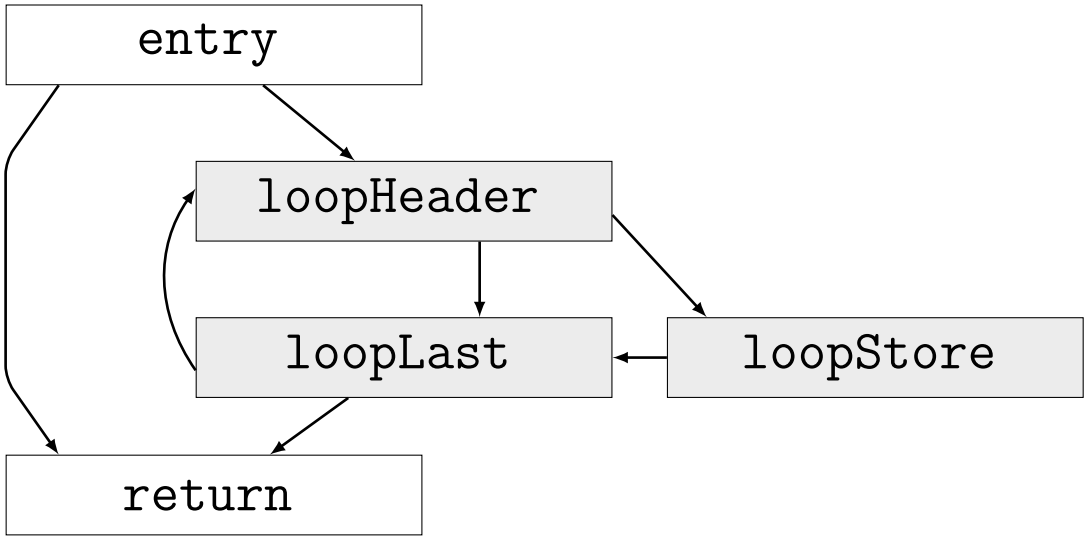
- LLVM Compiler Infrastructure provides optimizer and backends for building compilers
- LLVM IR: Assembly-like language used as internal representation of whole program at different stages of compilation
- Optimizer works in passes, either transforming the program or providing information to other passes
- Loop Invariant Code Motion attempts to move code out of loops that should just be executed once
- Moving arithmetic instructions is straightforward
- Moving memory-sensitive instructions is more involved

Conclusion

- LLVM Compiler Infrastructure provides optimizer and backends for building compilers
- LLVM IR: Assembly-like language used as internal representation of whole program at different stages of compilation
- Optimizer works in passes, either transforming the program or providing information to other passes
- Loop Invariant Code Motion attempts to move code out of loops that should just be executed once
- Moving arithmetic instructions is straightforward
- Moving memory-sensitive instructions is more involved

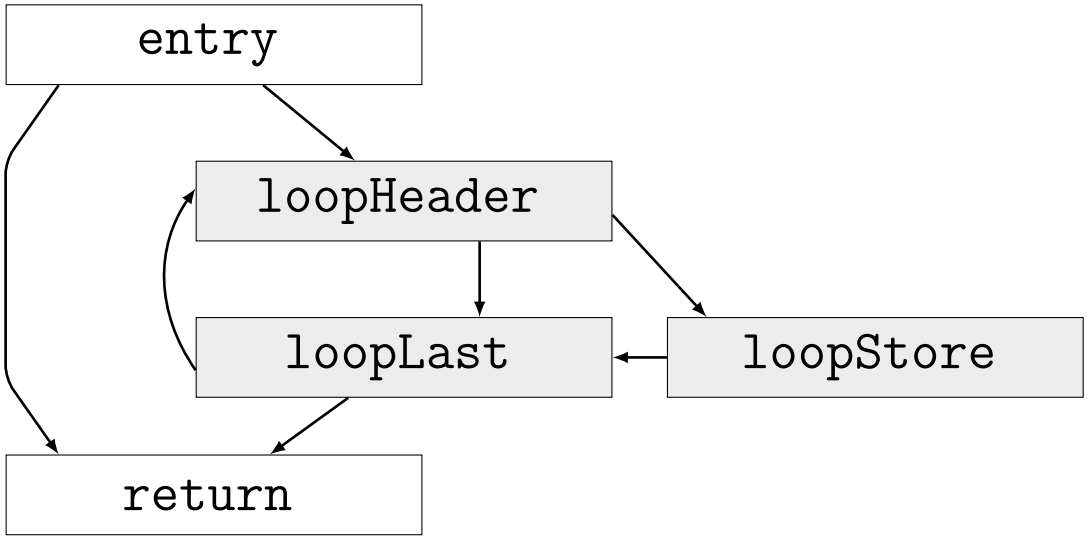
Thank you.

- Only branch at end of blocks $\Rightarrow$ Basic blocks as atomic units of control flow of function



- Block *A dominates* block *B* ("*A* dom *B*") iff in every execution, *A* is executed before *B*.
  - `entry` dominates all blocks
  - `loopHeader` dominates all blocks in the loop
  - `loopStore` and `return` dominate no blocks but themselves
- *Strict* domination ("*A* sdom *B*") iff additionally $A \neq B$

Dominator Tree



- Each block *B* but the initial block has a unique *immediate dominator I* such that
  1. *I* sdom *B* and
  2. There is no *A* with *I* sdom *A* and *A* sdom *B*
- Example:
  - `entry` sdom `loopLast`, but `entry` is not the immediate dominator of `loopLast`:
  - `entry` sdom `loopHeader` and `loopHeader` sdom `loopLast`
  - However, `loopHeader` is the immediate dominator of `loopLast`
- This yields the *dominator tree* in which the parent nodes dominate their child nodes

Natural Loops

Loop Invariant Code Motion operates on *natural loops*

- H is always executed before B and B has an edge back to H (“back edge”)
  $\Rightarrow$ H is header of the loop defined by the back edge
- All blocks A that have a path to B and H is always executed before A are part of the loop