CSC488/2107 Winter 2019 — Compilers & Interpreters

https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu

Agenda

- Local storage
  - Activation records
  - Lexical level
  - Displays
  - Closures
- Code generation
  - IR
  - Expressions and statements
  - Control flow graphs

Local storage

- Whereas a given program will have a fixed amount of required global storage (code, constants, variables, static data), the dynamic nature of procedural programming requires another level of *local storage*

- For most typical languages, this means some amount of memory is allocated on *entry* into a procedure/function, and deallocated on *exit*

    - Stack-like LIFO ordering: Last In, First Out

- A stack is an ideal structure for this because allocations & deallocations are very cheap operations (simple arithmetic on a stack pointer)

- These per-function-call allocations are laid out as *activation records*

Activation Records

Activation Records

- Organization of local storage for a particular function/procedure call (each call has its own copy of the record)
- Typical contents:
    - Return address
    - Parameters originally passed to function
    - Control block (extra bookkeeping)
    - Storage for *all* locally scoped variables (including enclosed minor scopes)
    - Pointers to any dynamically allocated memory (arrays, variable length arguments)
    - Temporary storage
- Use one hardware register to store address, and then use *register+constant displacement* instruction addressing to index into it

Activation Record — Example

```
func F(P, Q int32) {
   var x, y, z int32
   var c [4]int8

   if true {
      var t [4]int32
   }

   while true {
      var u, v int32
   }
}
```

Offset	Record
76	**v**
72	**u**
68	**t[3]**
64	**t[2]**
60	**t[1]**
56	**t[0]**
52	**c[0], c[1], c[2], c[3]**
48	**z**
44	**y**
40	**x**
36	**Q**
32	**P**
0	Control *(example 32B)*

Activation Record — Compact

```
func F(P, Q int32) {
   var x, y, z int32
   var c [4]int8

   if true {
      var t [4]int32
   }

   while true {
      var u, v int32
   }
}
```

Offset	Record
68	t[3] -
64	t[2] -
60	t[1] v
56	t[0] u
52	c[0] c[1] c[2] c[3]
48	z
44	y
40	x
36	Q
32	P
0	Control

Lexical Level

- The *lexical level* is the depth of static nested enclosing scopes within a program
    - Usually counted in terms of *major* scopes (since minor scopes can be compacted into the parent enclosing major scope)
        - Top level definitions at 0, top level defined routines at 1, nested routines at 2, etc.
- Each activation record is associated at the lexical level of the corresponding major scope
    - For most languages, at most one activation record at each lexical level is in scope at each point in the program

Lexical Levels — Static Depth

```
func F() {
   func A() {
      func P() { … }
      func Q() { … }
   }
   func B() { … }
}
```

Activation Records
Q
A
B
Q
P
A
B
A
F

Call Sequence: F ➡ A ➡ B ➡ A ➡ P ➡ Q ➡ B ➡ A ➡ Q

Lexical Levels — Static Depth

```
                          // Lvl  Visible Scopes
func F() {                //   0    F
  func A() {              //   1    F, A
    func P() { … }        //   2    F, A, P
    func Q() { … }        //   2    F, A, Q
  }
  func B() { … }          //   1    F, B
}
```

Displays

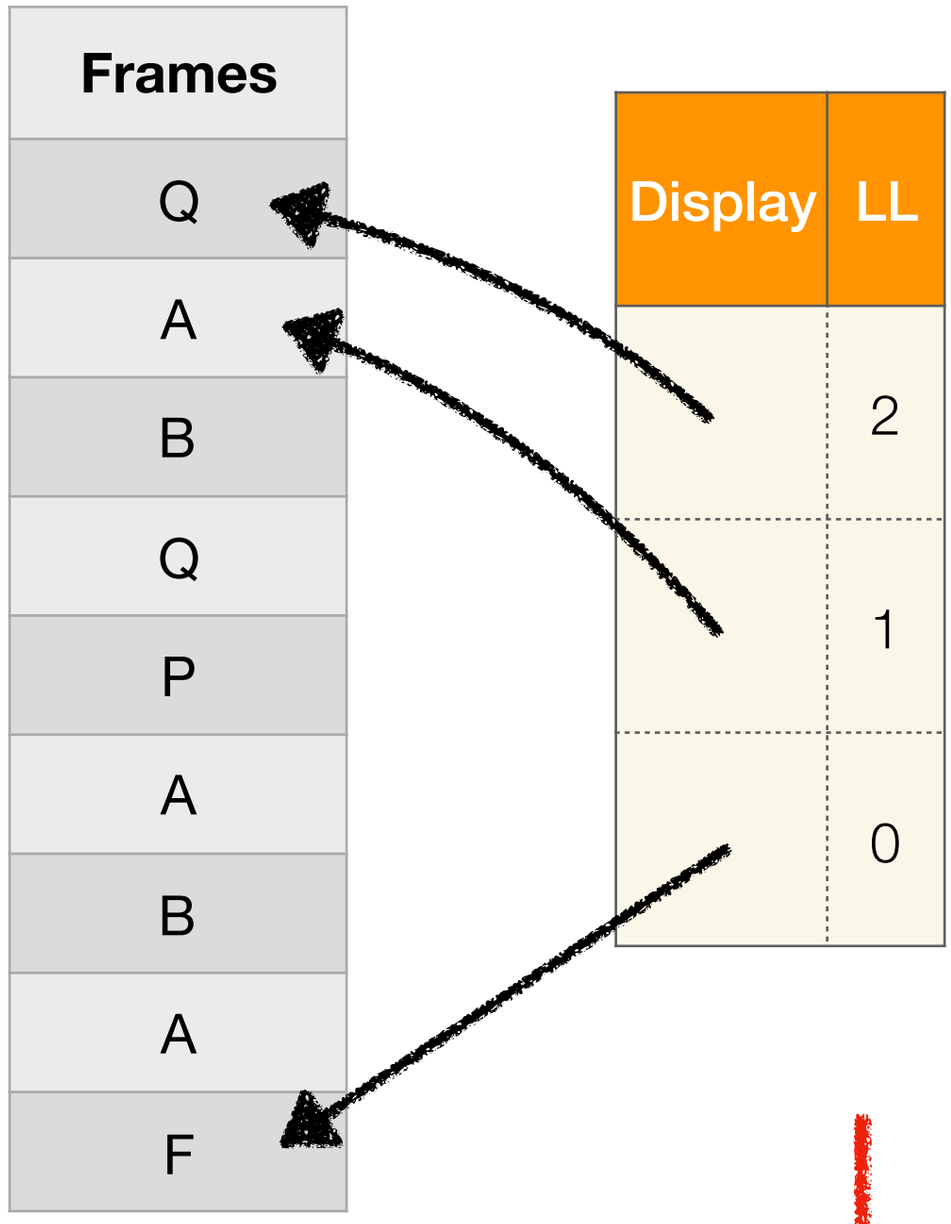
- The *display* is an array of activation record stack addresses, indexed by lexical level

- Combining a display entry with an *offset* lets you relatively address memory within the activation records
    - Non-negative (≥0) offsets indexed into the activation record itself
    - Negative offsets index below the activation record on the stack (this can be useful depending on *when* the activation frame is setup)

- Enables the access of activation records of enclosing parent scopes

Activation Records and Displays

```
func F() {
   func A() {
      func P() { … }
      func Q() { … }
   }
   func B() { … }
}
```



Call Sequence: F ➡ A ➡ B ➡ A ➡ P ➡ Q ➡ B ➡ A ➡ Q

Display update

- The display must be updated whenever the list of visible scopes changes

- Updates happen at functions/procedures call and return boundaries

- Activation records correspond with the major scopes of functions/procedures

    - All minor scopes are encapsulated within parent major scope

    - Semantic analysis controls visibility, even though the physical memory may overlap or be adjacent with the local storage of sibling minor scopes

Display update

Call type	Relationship	Example
Same level	Caller and calle have same lexic level	A calls B, P calls Q
Up level	Caller lexic level is less than callee level	F calls A/B, A calls P/Q
Down level	Caller lexic level is greater than callee level	P/Q calls F/A/B, A/B calls F

LL 0	LL 1	LL 2
F		
	A	
		P
		Q
	B	

Display update

Call Sequence: F_0 ➡ A_1 ➡ B_2 ➡ A_3 ➡ P_4 ➡ Q_5 ➡ B_6 ➡ A_7 ➡ Q_8

Depth/ Record #	Func	Display LL 0	Display LL 1	Display LL 2
0	F	**F 0**		
1	A	F 0	**A 1**	
2	B	F 0	**B 2**	
3	A	F 0	**A 3**	
4	P	F 0	A 3	**P 4**
5	Q	F 0	A 3	**Q 5**
6	B	F 0	**B 6**	*Q 5*
7	A	F 0	**A 7**	*Q 5*
8	Q	F 0	A 7	**Q 8**

LL 0	LL 1	LL 2
F		
	A	
		P
		Q
	B	

Where F 0, A 1, etc. represent activation record stack address

Display update & restore

- Both entry *to* and exit *from* a major scope requires updating the display

- On *entry* at lexical level L:

    - Save current display entry L to local storage

    - Prepare new activation record

    - Save record stack address to display entry L

- On *exit* at lexical level L:

    - Tear down the activation record

    - Restore previous value for display entry L

Problems with displays

- Only works with languages that have specific notion of nested functions that are only visible within the scope of the immediately enclosing parent scope

- Problems with languages that support passing *function pointers:*

  - The activation records of enclosing parent scopes may have gone away by the time the function pointer is called (since they may have returned)

  - Other activation frames may be currently in the display at a parents' lexical level

Closures

- Modern languages allow passing of *functional closures*, whereby a kind of activation record is dynamically allocated (i.e. does not live on the stack) and thus can outlive the duration of the enclosing function

- Requires a strategy for dealing with dynamically allocated closures that are no longer needed/referenced

- Automatic garbage collection, or a form of *lifetime analysis* to determine when its safe to free

Closures

```
func F(p int) func() int {
  var x int
  var e int

  func G() int {
    var y int
    e += 1
    return e
  }

  e = p

  return G
}

G := F(488)
G() // returns 489
G() // returns 490
```

- Activation frames (stack)
  - Stack allocated (fixed lifetime)
  - F: p, x
  - G: y
- G closure
  - Heap allocated (open ended lifetime)
  - e (*escaped* F when G referenced it)

- F returns a *function value*
- A *function value* consists of a function pointer (code reference) and a pointer to the dynamically allocated closure
  - Code reference known at compile time
  - Closure pointer is runtime variable

Closures

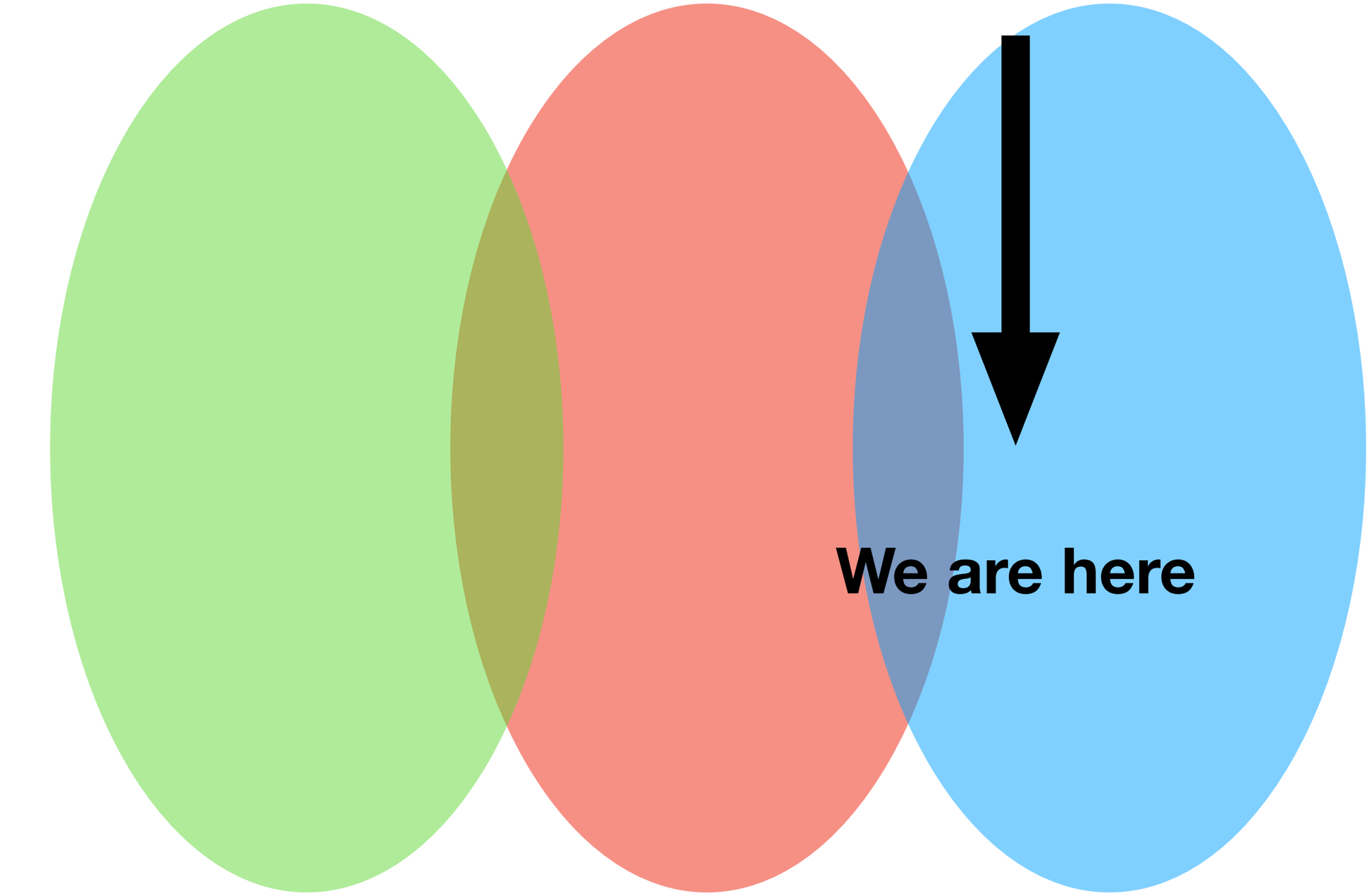
```
typedef struct {
  int (*G)(ClosureG *);
  int e;
} ClosureG;

int F_G(ClosureG *cg) {
  int y;
  cg->e += 1;
  return cg->e;
}

ClosureG *F(int p) {
  int x;
  int e;
  e = p;

  ClosureG *cg = malloc(…);
  cg->e = e;
  cg->G = F_G;
  return cg;
}
```

```
ClosureG *cg;
cg = F(488);

cg->G(cg);  // == 489
cg->G(cg);  // == 490



// Eventually???
free(cg);
```

Implementation details

- Each local variable has storage requirements in terms of units of the underlying memory model
    - Parameters, local scalars, arrays
- Laying out activation records can be done at semantic analysis time (as you encounter major and minor scopes), or as another pass between semantic analysis and code generation
    - Consider what information is specifically required for particulars of the language, and how to order the record fields for your convenience
- Determine and record total size of record by summing all variable storage within the major scope, plus the sizes of all enclosed minor scopes
    - Sum of all minor scope sizes, or maximum size over all minor scopes in compacted case
- Each local name/identifier/symbol needs to be associated with its lexical level, its storage size and an offset into the record

Recognize Analyze Transform



Frontend

Backend

Code Generation

Code Generation

- In the end, the user wants the compiler to transform their program sources into something the machine can actually execute (and hopefully quickly!)

- This transformation must be faithful to the particulars of the input language, and to the reasonable expectations of users

- During code generation, the compiler must translate each facet of the language into an appropriate sequence of machine instructions

- While the AST is a helpful abstraction for performing language-focused semantic analyses like name resolution and type checking, it is also helpful to have a machine-focused *intermediate representation*

Quadruple Intermediate Representation

Express machine instructions with their input and output registers as 4-tuples

```
( opcode, left, right, result )
```

- Assume an infinite number of temporary registers R_i
  - Hence the term *intermediate*, these will have to be mapped onto a finite set of physically available machine registers
- Special opcode called *label* (not an actual machine instruction)
- Results can include registers R_i, constant indices T_i and *labels*:
  - Constant T_i refers to tuple i in sequence
  - Symbolic label L_{name} will be resolved to a specific target tuple index later

IR — Example expression

```
A + B * C - D / E
```

(mul , B , C , R_1) $R_1 \leftarrow B * C$

(div , D , E , R_2) $R_2 \leftarrow D / E$

(add , A , R_1 , R_3) $R_3 \leftarrow A + R_1$

(sub , R_3 , R_2 , R_4) $R_4 \leftarrow R_3 - R_2$

Inputs: A, B, C, D, E
Output: R_4
Temporaries: R_1, R_2, R_3

IR — Branching

```
( X if (A < B + C) else Y )
```

(add	, B	, C	, R_1)	$R_1 \leftarrow B + C$
(lt	, A	, R_1	, R_2)	$R_2 \leftarrow A < R_1$
(bf	, R_2	, L_{false})		PC $\leftarrow L_{false}$ if !R_2
(mov	, X	, R_3)		$R_3 \leftarrow X$
(jmp		, L_{end})		PC $\leftarrow L_{end}$
(label		, L_{false})		_Lfalse:
(mov	, Y	, R_3)		$R_3 \leftarrow Y$
(label		, L_{end})		_Lend:

Both of these are forward *branch jumps*

Resolving label addresses

- Given a list of quads, resolve away symbolic labels through a two pass procedure

- 1st pass:

  - Iterate through list of quads, assigning an tuple index for each non-*label* opcode

  - For each *label* opcode, record label→tuple index for the index of the first following non-*label* opcode, and discard the original *label* opcode

- 2nd pass:

  - Iterate through list of quads, and for any symbolic label, lookup label→tuple index mapping, and replace with tuple index

  - If no such mapping, *error* (instruction refers to a label that was never declared)

IR — Branching

```
( X if (A < B + C) else Y )
```

(add	, B	, C	, R_1)	$R_1 \leftarrow B + C$
(lt	, A	, R_1	, R_2)	$R_2 \leftarrow A < R_1$
(bf	, R_2		, **L_{false}**)	PC ← **L_{false}** if !R_2
(mov	, X		, R_3)	$R_3 \leftarrow X$
(jmp			, **L_{end}**)	PC ← **L_{end}**
(label			, L_{false})	_Lfalse:
(mov	, Y		, R_3)	$R_3 \leftarrow Y$
(label			, L_{end})	_Lend:

IR — Branching

```
( X if (A < B + C) else Y )
```

T_1	(add , B , C , R_1)		$R_1 \leftarrow B + C$
T_2	(lt , A , R_1 , R_2)		$R_2 \leftarrow A < R_1$
T_3	(bf , R_2 ,	T_6 ~~Lfalse~~)	PC $\leftarrow$ T_6 if !R_2
T_4	(mov , X ,	R_3)	$R_3 \leftarrow X$
T_5	(jmp ,	T_7 ~~Lend~~)	PC $\leftarrow$ T_7
	~~(label~~ ,	~~Lfalse)~~	~~_Lfalse:~~
T_6	(mov , Y ,	R_3)	$R_3 \leftarrow Y$
	~~(label~~ ,	~~Lend)~~	~~_Lend:~~
T_7			

Generating Quadruples

- *emit(opcode, left, right, result)*: generate an quad where *opcode* is an available machine instruction

- *label(name)*: generates a uniquely identifiable symbolic label quad with optional prefix *name*

- *tempReg()*: generates a new unique temporary register

- *resolveLabels(code)*: performs two pass label address resolution process, returning the resultant pure-opcode code list

From AST to Quadruples

- In order to generate code for a given type of AST node, there will be some number of input registers & labels, and output registers, labels and code lists

- General from: *genCodeFoo(a_foo, inRegs, inLabels)* → *(outRegs, outLabels, outCode)*

    - More useful to pass in what we want and only return code, more on this later

- *genCode(node, …)* can be polymorphic in the type of node

    - Calls *genCode_Foo* for *node* of type *Foo*, calls *genCode_Bar* for *node* of type *Bar*, etc.

Unary operators

```
genCode_UnaryExpr(node):
   argCode, argOutput = genCode(node.operand)

   result = tempReg()
   code = argCode + [
      emit(node.operatorOpcode,
           argOutput, result)
      ]

   return code, result
```

Binary operators

```
genCode_BinaryExpr(node):
   leftCode, leftReg = genCode(node.left)
   rightCode, rightReg = genCode(node.right)

   result = tempReg()
   code = leftCode + rightCode + [
      emit(node.operatorOpcode,
           leftReg, rightReg, result)
      ]

   return code, result
```

Non-short circuiting

Assignment statement

```
genCode_AssignStmt(node):
   addrCode, addrReg = genCode(node.lhs)
   valCode, valReg = genCode(node.rhs)

   code = addrCode + valCode + [
      #      *addrReg = valReg
      emit(Store, valReg, addrReg)
    ]

   return code
```

Order of operations

Does it matter what order operations are executed in?

```
P()
Q()
```

```
A[f()] = g()
```

Order of operations

Does it matter what order operations are executed in?

```
A[f()][g()] = h()
```

```
print f(g(), h())
```

Boolean Conditional Expressions

- Many languages feature Boolean conditional expressions that *short circuit*, that is, they evaluate only the operands they absolutely have in order to determine the final result

- Example: ***false and F()***

  - *F* is never called, since there is no way that its' return value could have any impact on the resultant *false* value

- Example: ***true or G()***

  - *G* is never called, since the result of the expression will already have been determined to be *true*

Boolean Conditional Expressions

```
result = (P and Q)
```

```
result = P
if result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P or Q)
```

```
result = P
if !result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P and Q)
```

```
/* genCode(P) */          R1 ← P
( bf     , R1     , Lend ) PC ← Lend if !R1
/* genCode(Q) */          R1 ← Q
( label          Lend )   _Lend:
```

```
result = P
if result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P or Q)
```

```
/* genCode(P) */          R1 ← P
( bt     , R1     , Lend ) PC ← Lend if R1
/* genCode(Q) */          R1 ← Q
( label          Lend )   _Lend:
```

```
result = P
if !result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P or Q)
```

/* genCode(P) */		$R_1 \leftarrow P$
(**bf**	, R_1 , L_{right})	PC $\leftarrow L_{right}$ if !R_1
(**jmp**	L_{end})	PC $\leftarrow L_{end}$
(label	**L_{right}**)	_Lright:
/* genCode(Q) */		$R_1 \leftarrow Q$
(label	L_{end})	_Lend:

```
result = P
if !result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P or Q)
```

/* genCode(P) */	$R_1 \leftarrow P$
(**bf** , R_1 , **L_{right}**)	PC ← L_{right} if !R_1
(**jmp** L_{end})	PC ← L_{end}
(label **L_{right}**)	Lright:
/* genCode(Q) */	**$R_1 \leftarrow Q$**
(label L_{end})	_Lend:

But *genCode(Q)* tells *us* what register it put its return value into…

Boolean Conditional Expressions

```
result = (P or Q)
```

/* genCode(P) */	$R_1 \leftarrow P$
(bf , R_1 , L_{right})	PC $\leftarrow$ L_{right} if !R_1
(jmp , L_{end})	PC $\leftarrow$ L_{end}
(label , L_{right})	_Lright:
/* genCode(Q) */	**$R_2 \leftarrow Q$**
(mov , R_2 , R_1)	**$R_1 \leftarrow R_2$**
(label , L_{end})	_Lend:

Binary operators

```
genCode_BinaryExpr(node):
   leftCode, leftReg = genCode(node.left)
   rightCode, rightReg = genCode(node.right)

   result = tempReg()
   code = leftCode + rightCode + [
      emit(node.operatorOpcode,
           leftReg, rightReg, result)
      ]

   return code, result
```

Binary operators

For expressions instead pass in the target result register

```
genCode_BinaryExpr(node, result):
   leftReg = tempReg()
   leftCode = genCode(node.left, leftReg)
   rightReg = tempReg()
   rightCode = genCode(node.right, rightReg)

   code = leftCode + rightCode + [
      emit(node.operatorOpcode,
           leftReg, rightReg, result)
      ]

   return code
```

Assignment statement

```
genCode_AssignStmt(node):
   addrCode, addrReg = genCode(node.lhs)
   valCode, valReg = genCode(node.rhs)

   code = addrCode + valCode + [
      emit(Store, valReg, addrReg)
      ]

   return code
```

Assignment statement

```
genCode_AssignStmt(node):
   addrReg = tempReg()
   addrCode = genCode(node.lhs, addrReg)
   valReg = tempReg()
   valCode = genCode(node.rhs, valReg)

   code = addrCode + valCode + [
      emit(Store, valReg, addrReg)
      ]

   return code
```

Boolean Conditional Expressions

```
result = (P and Q)
```

```
/* genCode(P) */          R1 ← P
( bf     , R1     , Lend ) PC ← Lend if !R1
/* genCode(Q) */          R1 ← Q
( label          Lend )   _Lend:
```

```
result = P
if result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P and Q)
```

```
genCode_BoolAndExpr(node, result):
 leftCode = genCode(node.left, result)
  rightCode = genCode(node.right, result)
Lend = label(“end”)
code =
    leftCode +
    [ emit(Bf, result, Lend) ] +
    rightCode +
    [ Lend ]
return code
```

Boolean Conditional Expressions

```
result = (P or Q)
```

/* genCode(P) */	$R_1 \leftarrow P$
(**bf** , R_1 , **L_{right}**)	PC $\leftarrow L_{right}$ if !R_1
(**jmp** **L_{end}**)	PC $\leftarrow L_{end}$
(label **L_{right}**)	_Lright:
/* genCode(Q) */	$R_1 \leftarrow Q$
(label **L_{end}**)	_Lend:

```
result = P
if !result {
    result = Q
}
```

Boolean Conditional Expressions

```
result = (P or Q)
```

```
genCode_BoolOrExpr(node):
  # result, leftCode, rightCode, Lend …
  Lright = label(“right”)
  code =
     leftCode +
     [     emit(Bf, result, Lright),
           emit(Jmp, Lend),
           Lright ] +
     rightCode +
     [ Lend ]
  return code
```

What about also passing in labels*?*

Control Flow and Boolean Expressions

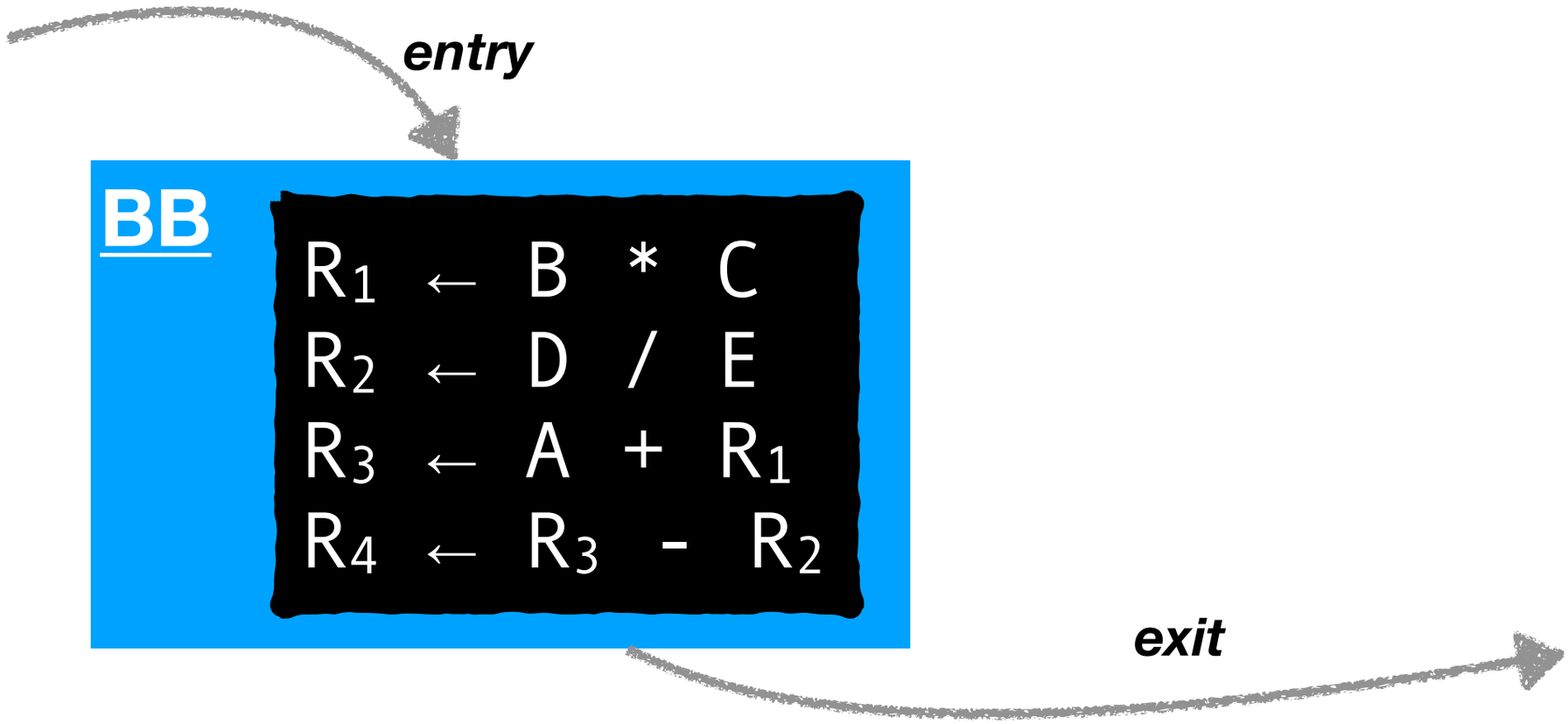
- Boolean expressions (both conditional and relational equality/inequality) always appear as the condition expression in *if* statements, *while* and *repeat…until* loops

- Instead of thinking of them solely as Boolean *valued*, it's useful to think of them in terms of how they affect *control flow*

Control Flow Graphs

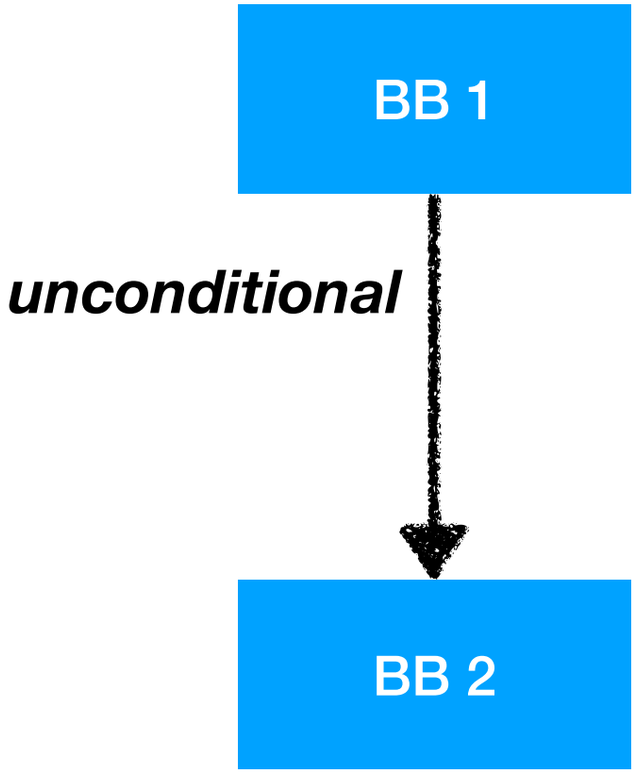
Control Flow Graphs

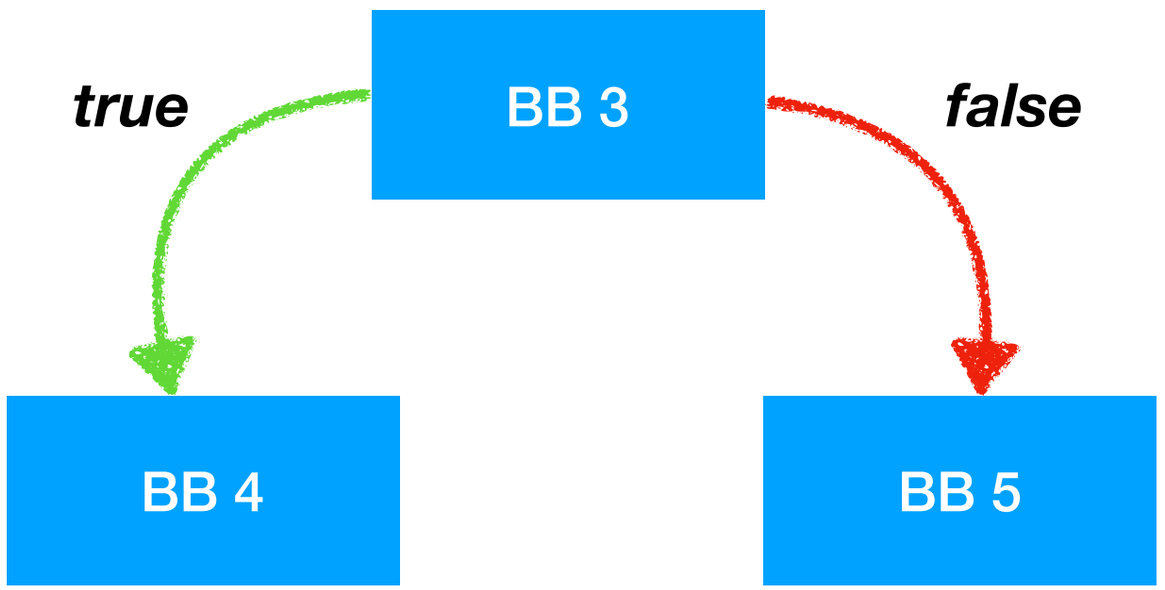
- A *basic block* is a sequence of instructions with exactly one entry point and one exit point
  - No branching or jump instructions
  - Execution will always run top to bottom uninterrupted
  - Nothing will ever jump into the middle of a basic block



Control Flow Graphs

- Two basic blocks can be connected through either an *unconditional* or a *conditional* branch

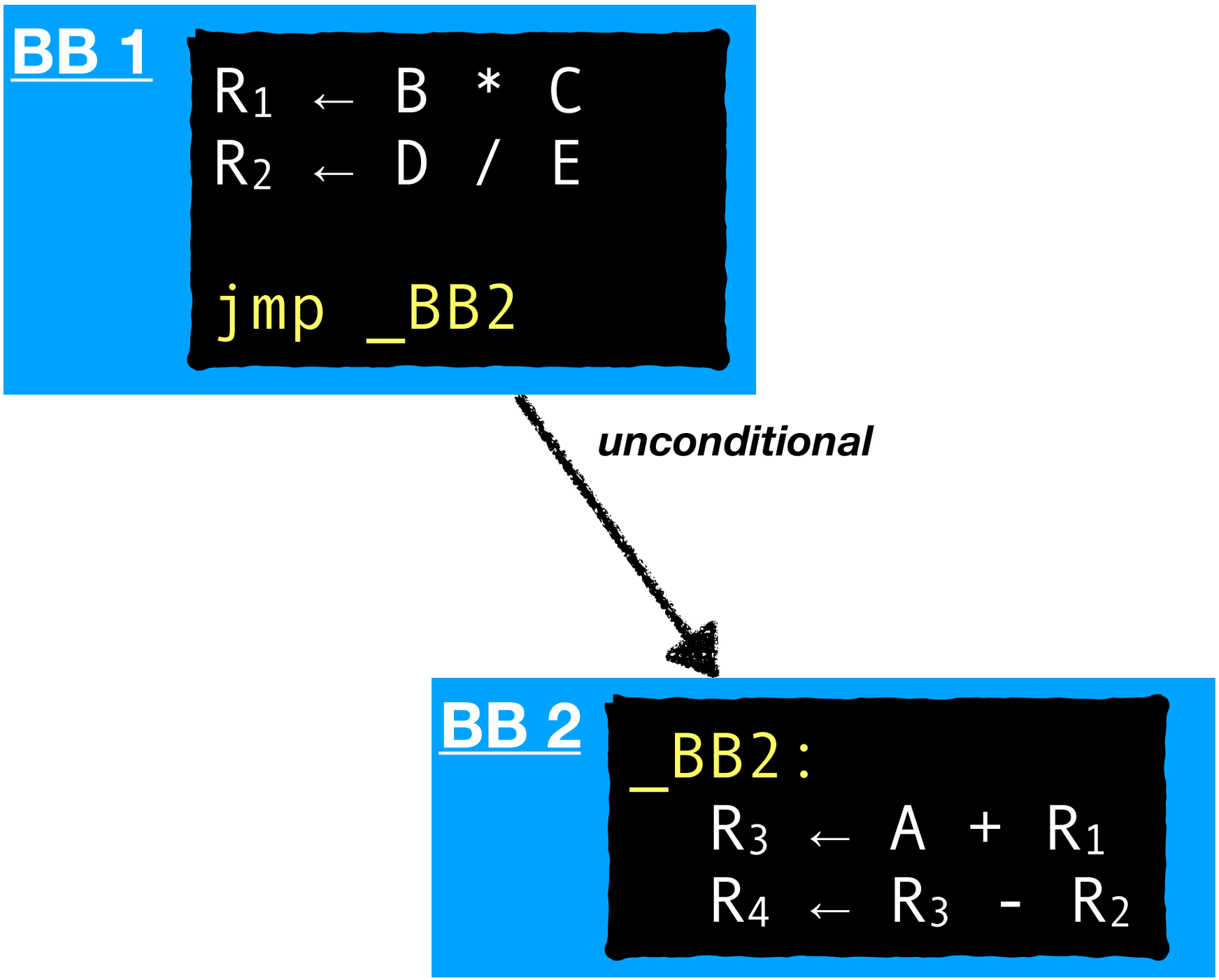




Control Flow Graphs

- The *control flow graph* for a program is the collection of basic blocks and all their interconnection edges
- It describes all possible flows of control through the program

Control Flow Graphs



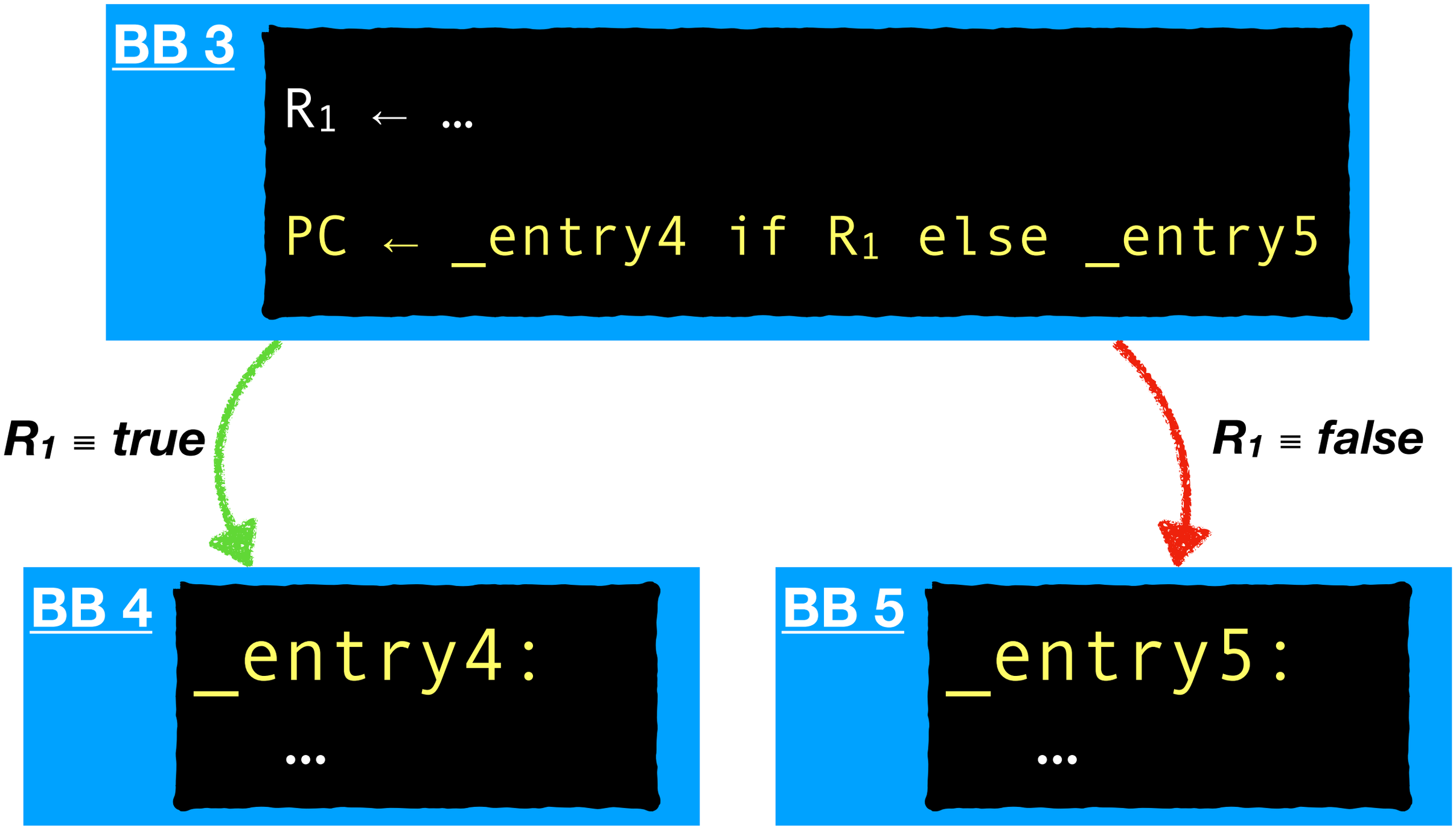
Two-sided branch opcode

Assume a branch (*br*) opcode that specifies both an *if-true* branching target, as well as *if-false* one

(**br** , R_{result} , T_{true} , T_{false}) PC ← T_{true} if R_{result} else T_{false}

***Question**: why can we just* add *whatever is convenient to the IR?*

Control Flow Graphs



Boolean Conditional Expressions

```
result = (P or Q)
```

/* genCode(P) */	$R_1 \leftarrow P$
(**bf** , R_1 , **L_{right}**)	PC $\leftarrow L_{right}$ if !R_1
(**jmp** **L_{end}**)	PC $\leftarrow L_{end}$
(label **L_{right}**)	_Lright:
/* genCode(Q) */	$R_1 \leftarrow Q$
(label **L_{end}**)	_Lend:

```
result = P
if !result {
    result = Q
}
```

Basic Blocks — Boolean *or*

```
(P or Q)
```

BBleft

```
R1 ← …P…
PC ← BBtrue if R1 else BBright
```

R1 ≡ true

R1 ≡ false

BBright

```
R1 ← …Q…
PC ← BBtrue if R1 else BBfalse
```

R1 ≡ true

R1 ≡ false

BBtrue

BBfalse

Boolean Conditional Expressions

```
result = (P or Q)
```

```
genBB_BoolOrExpr(node, result, bbT, bbF):
  bbR = genBB(node.right, result, bbT, bbF)
  bbL = genBB(node.left, result, bbT, bbR)
  return bbL
```

Boolean Conditional Expressions

```
result = (P or Q)
```

```
genBB_BoolOrExpr(node, result, bbT, bbF):
  bbR = genBB(node.right, result, bbT, bbF)
  bbL = genBB(node.left, result, bbT, bbR)
  return bbL
```

```
result = (P and Q)
```

```
genBB_BoolAndExpr(node, result, bbT, bbF):
  bbR = genBB(node.right, result, bbT, bbF)
  bbL = genBB(node.left, result, bbR, bbF)
  return bbL
```

Value of Boolean *or*

```
x = (P or Q)
```

BB$_{or}$

```
R1 ← …P or Q…
PC ← BBstore if R1 else BBstore
```

R₁ ≡ true

R₁ ≡ false

BB$_{store}$

```
*x ← R1

…
```

Assignment in basic blocks

```
genBB_AssignStmt(node, nextBB):
   addrReg, valReg = tempReg(), tempReg()

   bb = newUnconditionalBB(
      code=[ emit(Store, valReg, addrReg) ],
      jump=nextBB)

   bbR = genCode(node.rhs, valReg, bb, bb)

   bbL = genBB(node.lhs, addrReg, bbR, bbR)

   return bbL
```

Control Flow Graphs

- A control flow graph of interconnected basic blocks can be assembled into a final linear list of instructions and labels

- Many opportunities for optimization at the level of the intermediate representation:

  - Block layout order can take advantage of branch fall-through instructions

  - Remove unconditional jumps by merging two basic blocks into single sequence of instructions (possibly with a label if second block was the target of >1 branches)

  - When *bbLeft=bbRight*, transform conditional branch into unconditional jump (then apply above)

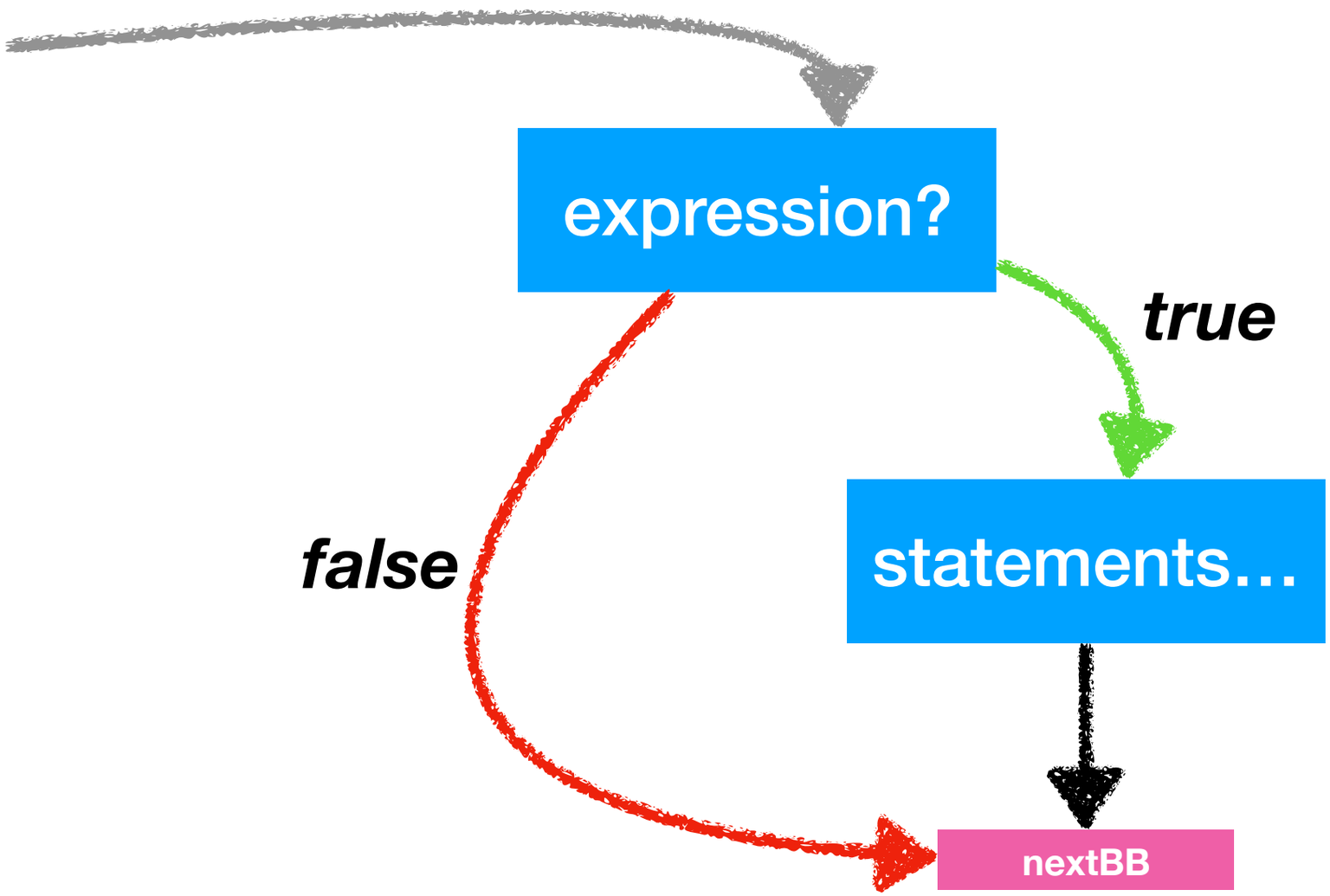
  - Delete instructions that produce values that are never used (requires more elaborate *usage analysis* to prove)

  - Propagate constant values, unroll short loops, hoist out common subexpressions, convert expensive instructions to cheaper ones

Control flow graph examples

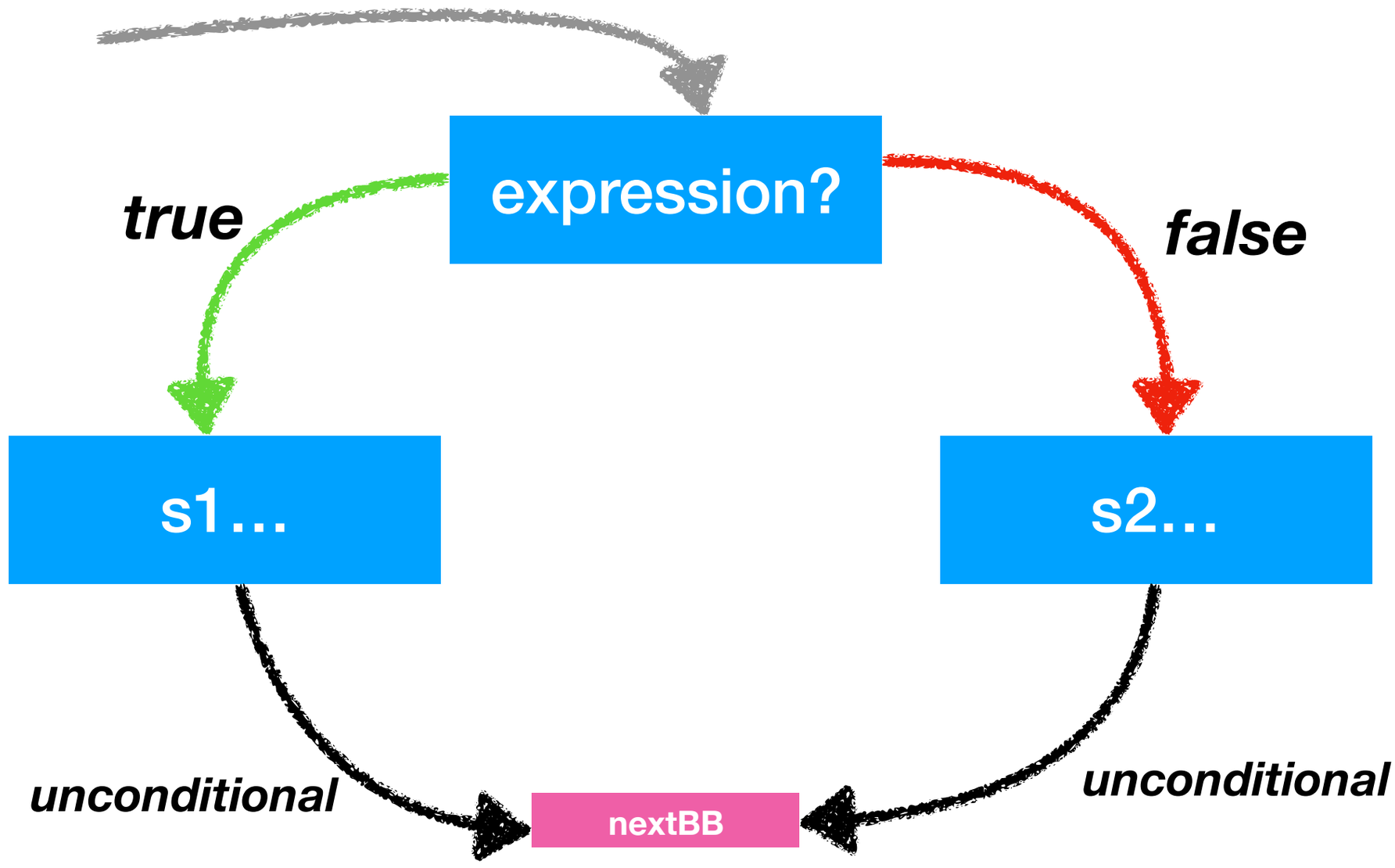
CFG Examples

```
if expression? { statements… }
```



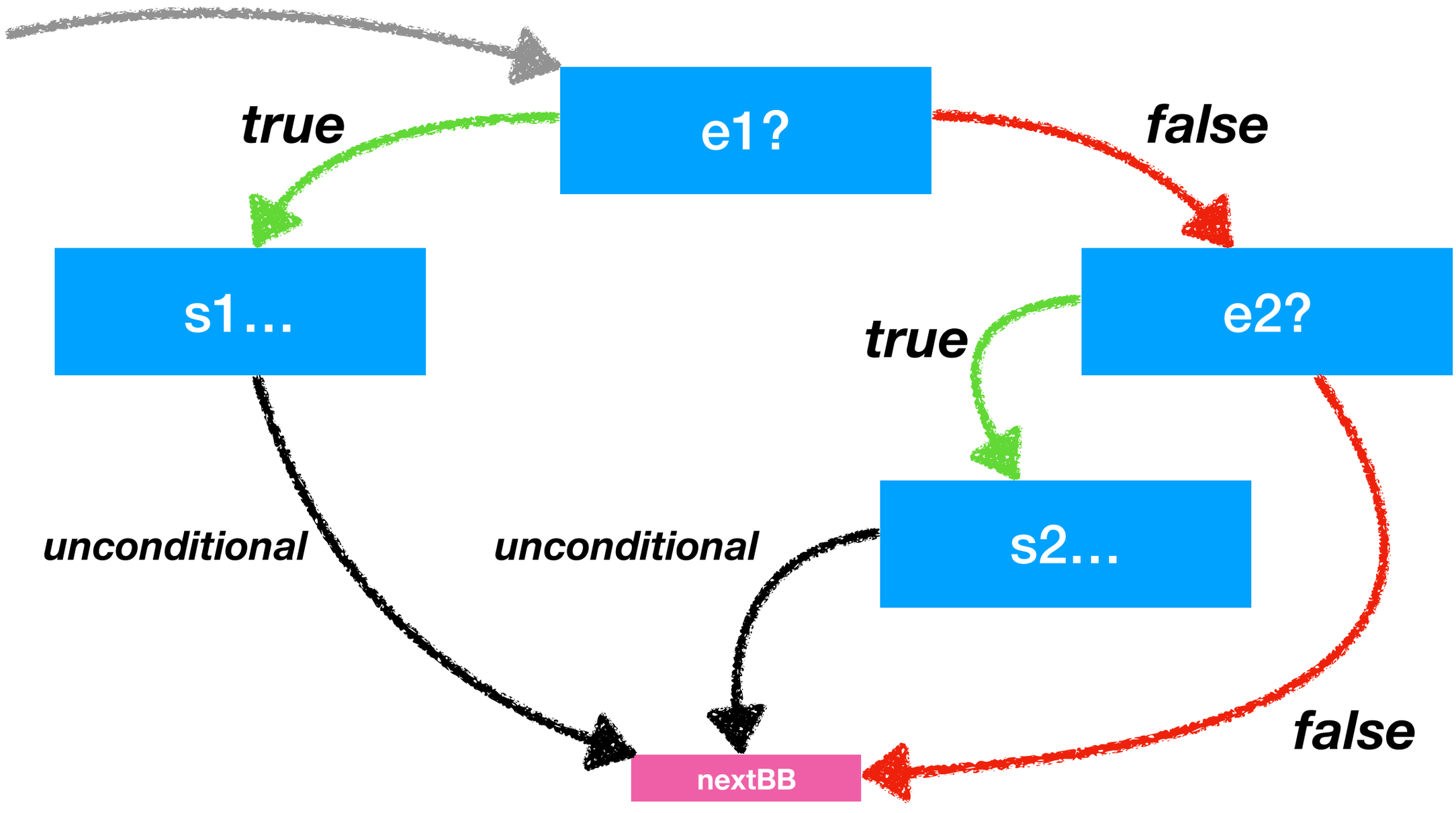
CFG Examples

```
if expression? { s1… } else { s2… }
```



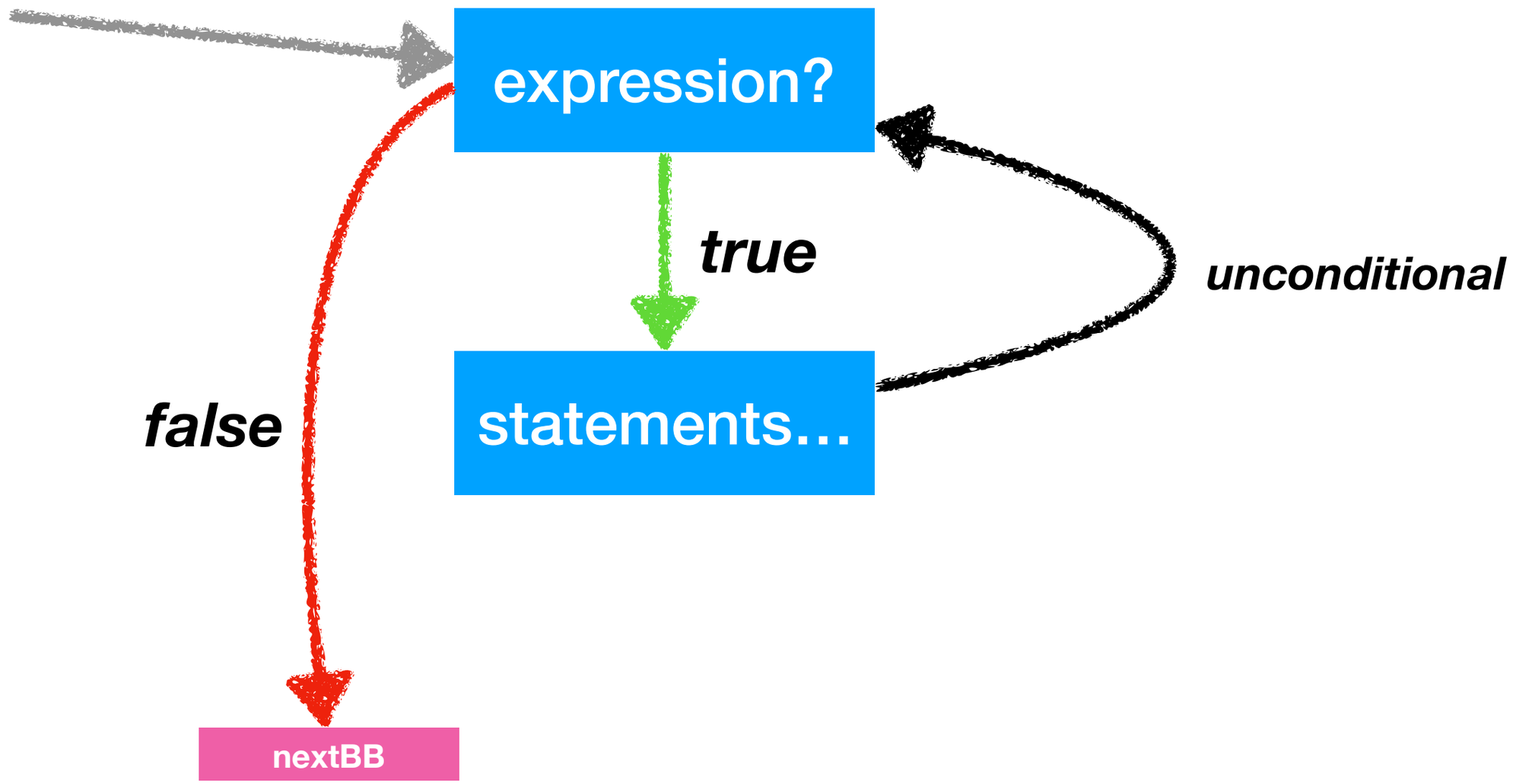
CFG Examples

```
if e1? { s1… } else if e2? { s2… }
```



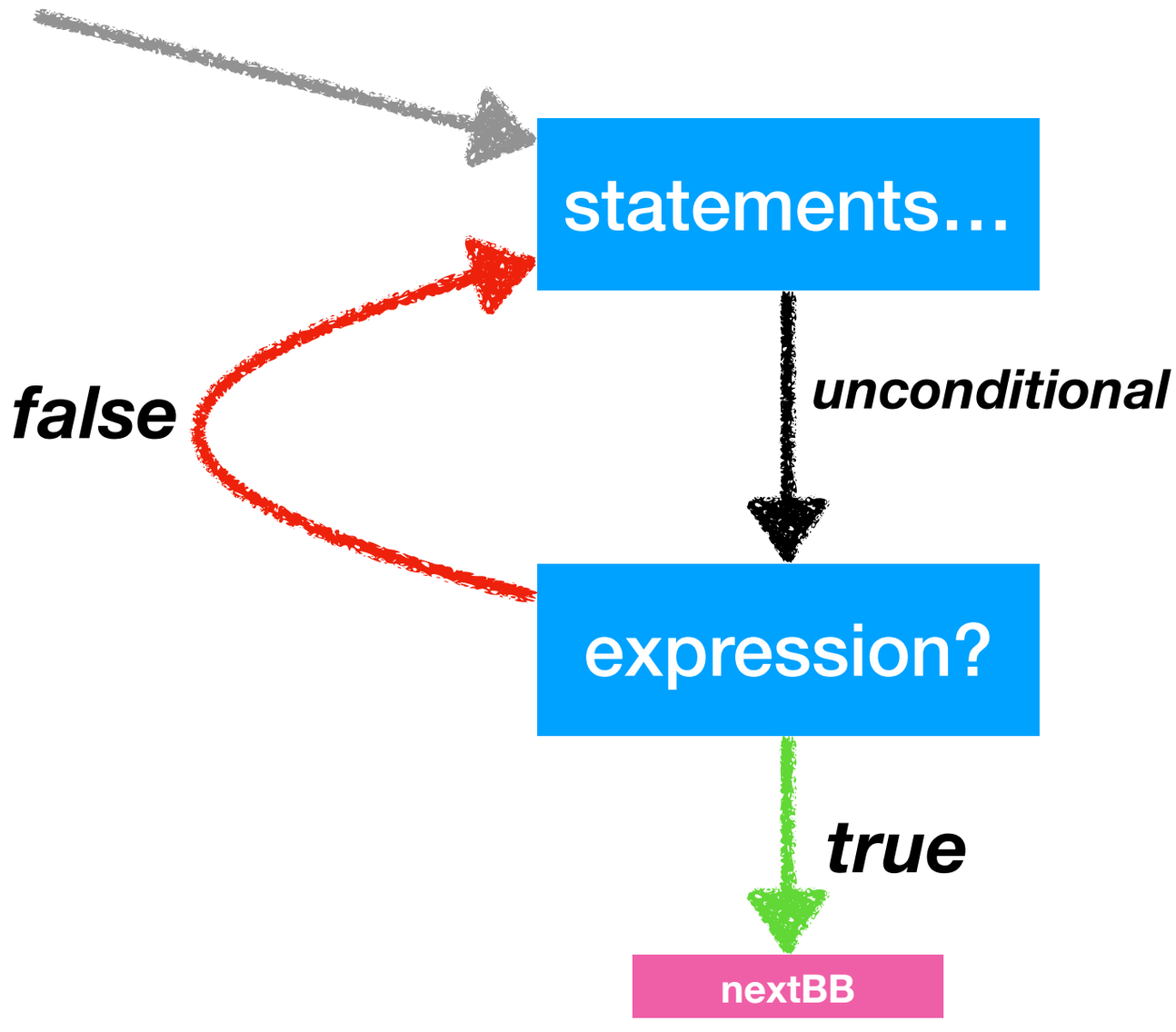
CFG Examples

```
while expression? { statements… }
```



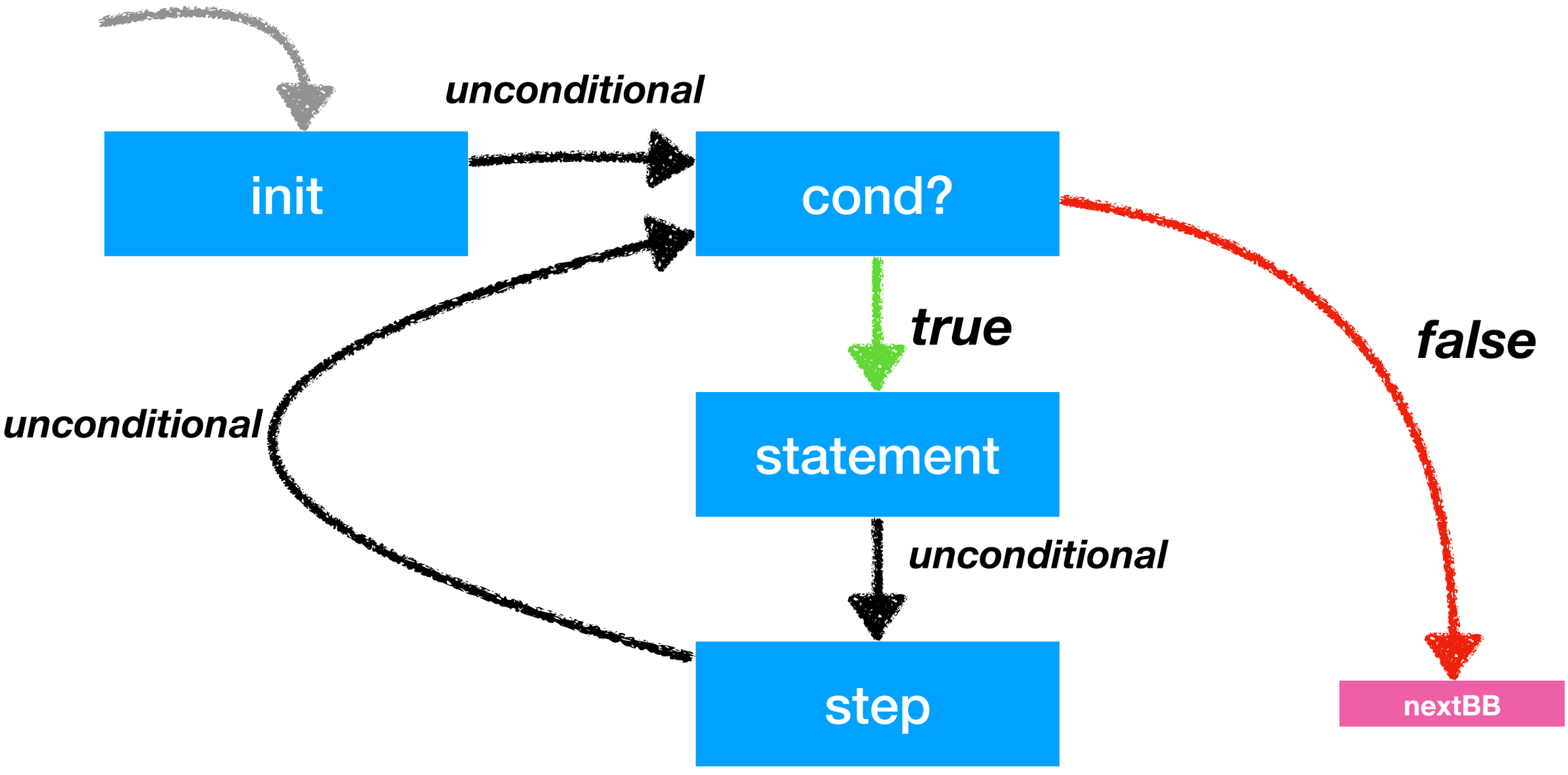
CFG Examples

```
repeat { statements… } until expression?
```



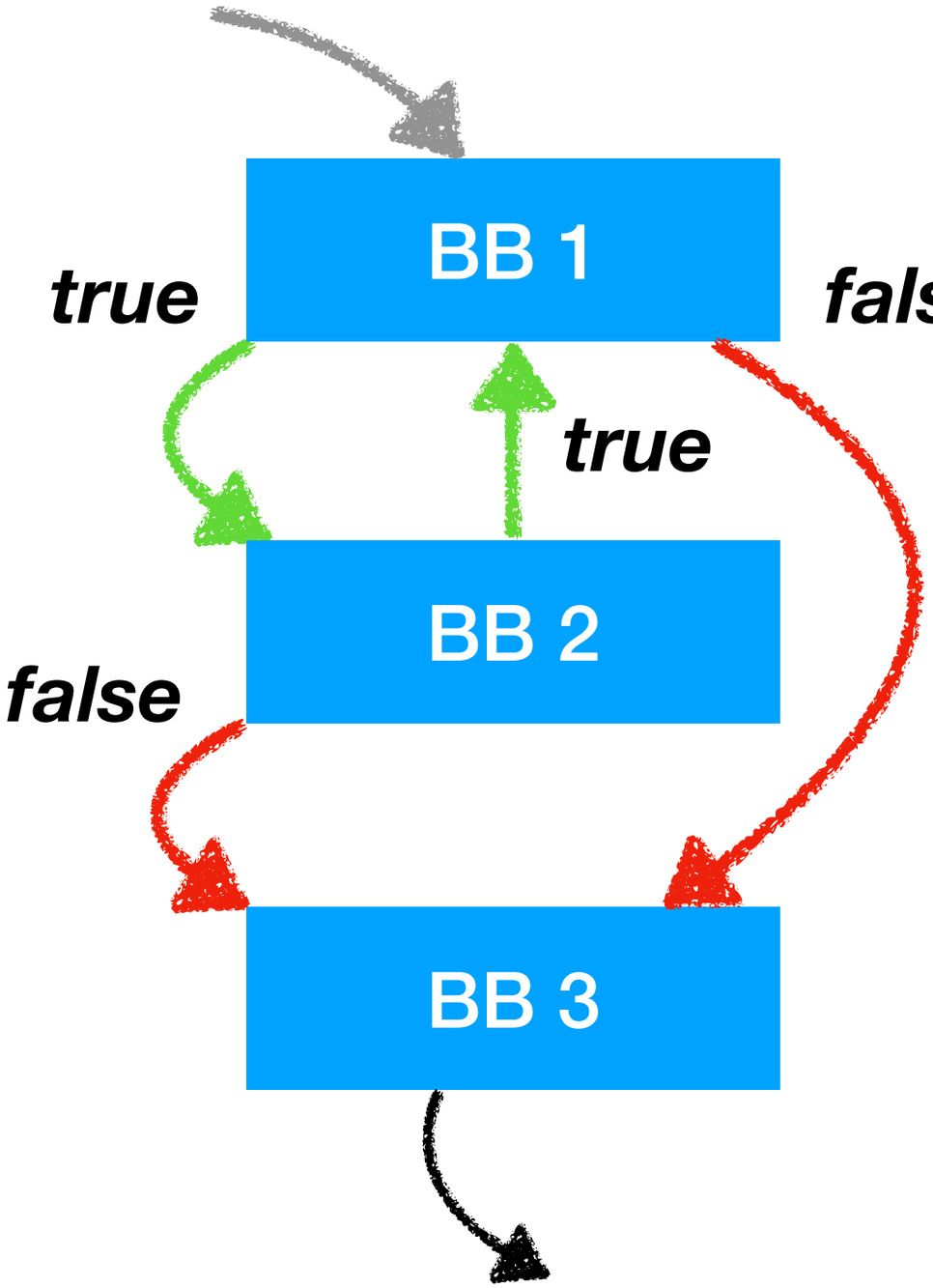
CFG Examples

```
for (init; cond?; step) statement
```



Contains both forwards *and* backwards *jumps*

Mutually recursive basic blocks




```
genBB_Rec(…):
   bb3 = genBB(…)

   # Uh oh…
   bb1 = genBB(…, bb2, bb3)
   bb2 = genBB(…, bb1, bb3)

   return bb3
```

Mutually recursive basic blocks

```
genBB_Rec(…):
   bb3 = genBB(…)

   bb2ref = bbRef()
   bb1ref = bbRef()

   bb1 = genBB(…, bb2ref, bb3.ref)
   bb2 = genBB(…, bb1ref, bb3)

   bb2ref.ref = bb2
   bb1ref.ref = bb1

   return bb3
```

```
class BB:
   __init__:
      self.ref = self
```