CS 335: Code Generation

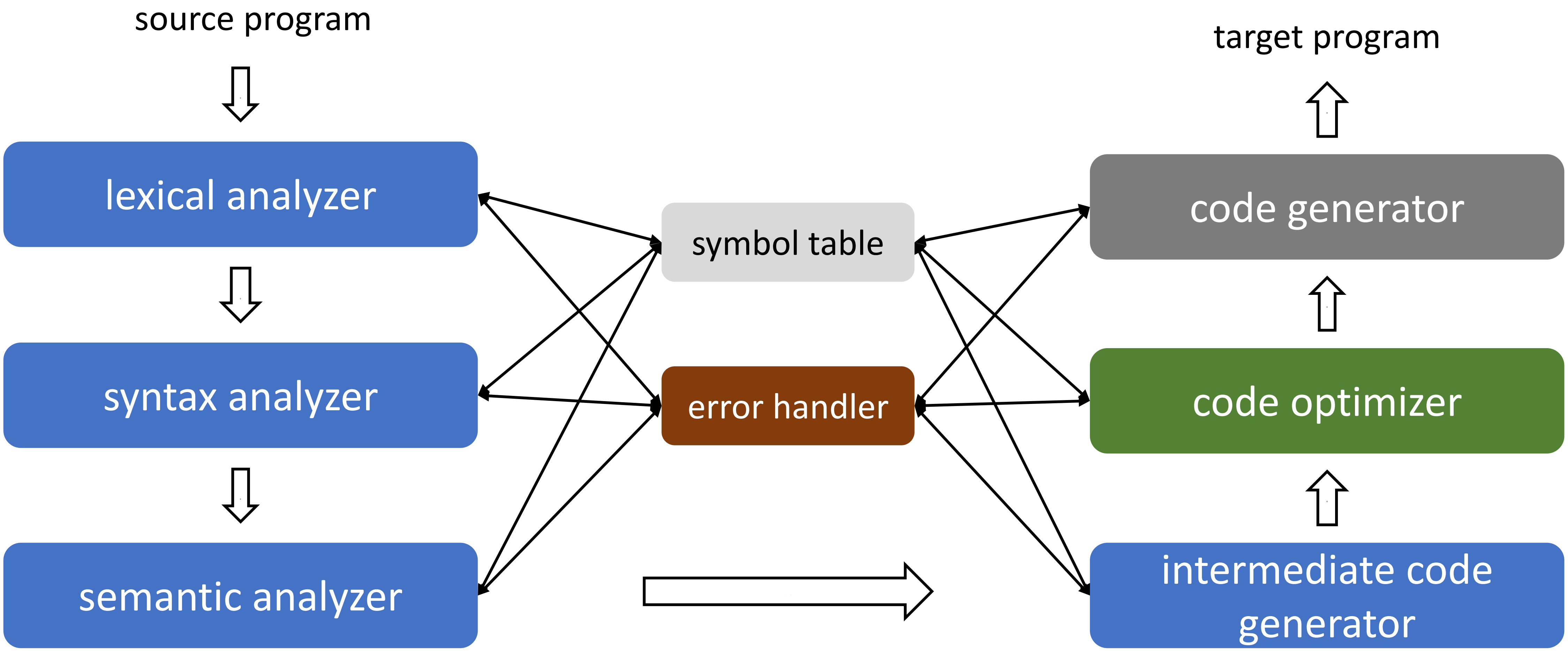
Swarnendu Biswas

Semester 2022-2023-II

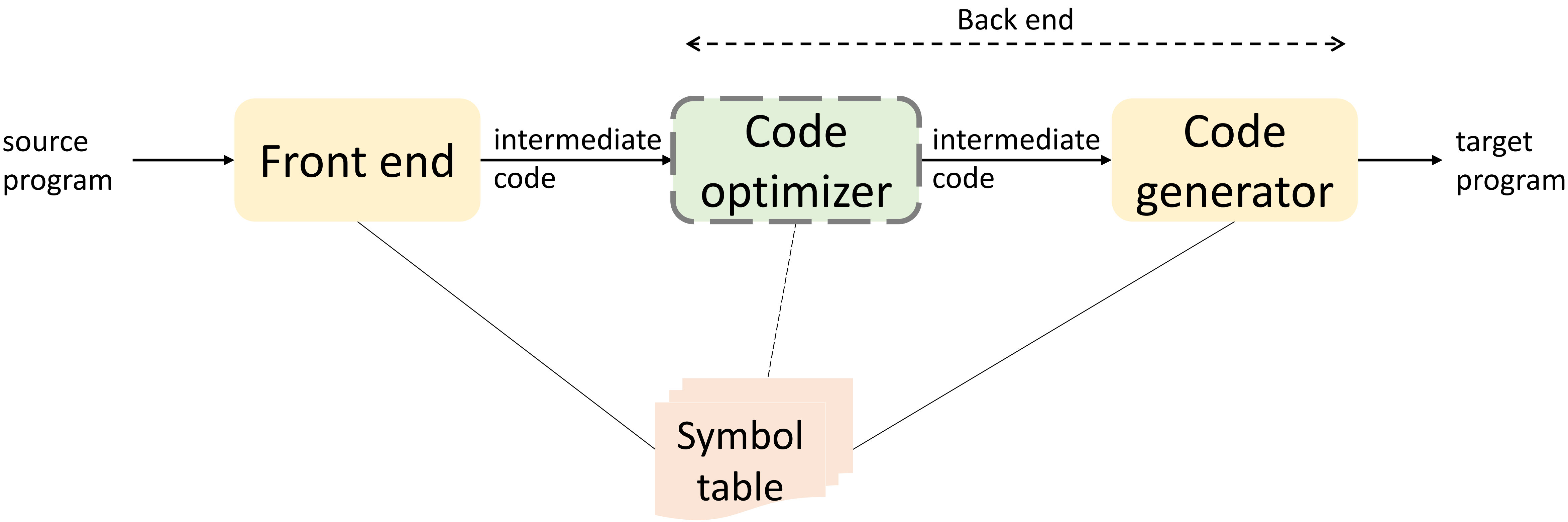
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

An Overview of Compilation



Code Generation



Code Generation

i. Generated output code must be correct

ii. Generated code must be of “good” quality
   - Should make efficient use of resources on the target machine
   - Notion of good can be vary

iii. Code generation should be efficient

- Generating optimal code is undecidable, compilers make use of well-designed heuristics

Code Generation

- **Input**
  - Intermediate representation (IR) generated by the front end
    - Linear IRs like 3AC or stack machine representations
    - Graphical IRs also work
  - Symbol table information
- **Assumptions**
  - Code generation does not bother with any error checking
  - Code generation assumes that types in the IR can be operated on by target machine instructions
    - For example, bits, integers, and floats

Code Generation

- **Output**
  - Absolute machine code
    - Generated **addresses are fixed** and works when loaded at fixed locations in memory
    - Efficient to execute, now primarily used in embedded systems
  - Relocatable machine code
    - Code can be broken down into separate sections and loaded anywhere in memory that meets size requirements
    - Allows for separate compilation, but requires a **separate linking and loading** phase
  - Assembly language
    - Simplifies code generation, but **requires assembling** the generated code

Steps in Code Generation

- Compiler backend performs three steps to translate IR to executable code
  - Instruction selection – Choose appropriate target machine instructions while generating code
  - Register allocation – Decide what values to keep in which registers
  - Instruction scheduling – Decide in what order to schedule the execution of instructions
- Manage memory during execution

Instruction Selection

- Complexity arises because each IR instruction can be translated in several ways, combinatorial problem

`a = a + 1`	`1. LD R0, a`
	`   ADD R0, R0, #1`
	`   ST a, R0`

	`2. INC a`

- Target ISA influences instruction selection
  - Scalar RISC machine – simple mapping from IR to assembly
  - CISC machine – may need to fuse multiple IR operations for effectively using CISC instructions
  - Stack machine – need to translate implicit names and destructive instructions to assembly

Instruction Selection

- Possible idea
  - Devise a target code skeleton for every 3AC IR instruction
  - Replace every 3AC instruction with the skeleton
- Need a cost model and heuristics for selection
  - Other factors are level of abstraction of the IR, speed of instructions, energy consumption, and space overhead

```
x = y + z
```

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

```
a = b + c
d = a + e
```

```
LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```

redundant

Register Allocation

- Instructions operating on register operands are more efficient
  - **Register allocation** – Choose which variables will reside in registers
  - **Register assignment** – Choose which registers to assign to each variable
- Architectures may impose restrictions on usage of registers
- Finding an optimal assignment of registers to variables is NP-complete

- Architectures such as IBM 370 may require register pairs to be used for some instructions

`MUL x, y`	• `x` is in the even register, `y` is in the odd register • Product occupies the entire even/odd register pair
`DIV x, y`	• 64-bit dividend occupies the even/odd register pair • Even register holds the remainder, odd register the quotient

Instruction Scheduling

- Order of evaluating the instructions also affect the efficiency of the target code
- Selecting the best order across inputs is a NP-complete problem

Example Target Machine

- Efficient code generation requires good understanding of the target ISA
- **Assumptions**
    - Three-address machine, byte-addressable with four-byte words
    - n general-purpose registers
    - OP dst, src_1, src_2; LD dst addr; ST dst, src; BR L; Bcondr L;

Addressing Modes

- Specifies how to **interpret the operands** of an instruction

Mode	Form	Address	Example
absolute	`M`	M	`LD R0, M`
register	`R`	R	`ADD R0, R1, R2`
indexed	`c(R)`	c + contents(R)	`LD R1, 4(R0)`
indirect register	`*R`	contents(R)	`LD R1, *R0`
indirect indexed	`*c(R)`	contents(c + contents(R))	`LD R1, *100(R0)`
literal	`#c`	c	`LD R1, #1`

Few Examples

$x = y - z$	LD $R1, y$
	LD $R2, z$
	SUB $R1, R1, R2$
	ST $x, R1$
		// R1 = y
		// R2 = z
		// R1 = R1 – R2
		// x = R1

if $x < y$ **goto** L	LD $R1, x$
	LD $R2, y$
	SUB $R1, R1, R2$
	BLTZ $R1, M$
		// R1 = x
		// R2 = y
		// R1 = R1 – R2
		// if R1 < 0 JMP M

$b = a[i]$	LD $R1, i$
	MUL $R1, R1, 8$
	LD $R2, a(R1)$
	ST $b, R2$
		// R1 = i
		// R1 = R1 * 8
		// R2 = c(a + c(R1))

$a[j] = c$	LD $R1, c$
	$LD\ R2, j$
	MUL $R2, R2, 8$
	ST $a(R2), R1$
		// R1 = c
		// R2 = j
		// R2 = R2 * 8
		// c(a + c(R2)) = R1

$x =* p$	LD $R1, p$
	$LD\ R2, 0(R1)$
	ST $x, R2$
		// R1 = p
		// R2 = c(0+c(R1)
		// x = R2

$* p = y$	LD $R1, p$
	$LD\ R2, y$
	ST $0(R1), R2$
		// R1 = p
		// R2 = y
		// c(0+c(R1) = R2

Runtime Storage Management

- Let us consider the following 3AC: `call callee`, `return`, `halt`, `action`
- Assume that the first location in the activation record (given by $staticArea$) of the callee stores the return address of the caller

	Static Allocation	
	ST $callee.staticArea, \#here + 20$	Store return address in the first slot in the callee's activation record, assume 2 opcodes and 3 constants are each of 4 bytes
	BR $callee.codeArea$	Transfer control to callee
return → address	…	

BR $*callee.staticArea$	Return transfer to caller
…	

Determine Addresses in Target Code

- Need to generate code to manage activation records at runtime

3AC

```
// code for func c
action1
call p
action2
halt
```

```
// code for func  p
action3
return
```

	Activation record for c (64 Bytes)
0:	return address
4:	`arr`
56:	`i`
60:	`j`

	Activation record for p (88 Bytes)
0:	return address
4:	`buf`
84:	`n`

Target Code for Static Allocation

text area

		// code for c
100:	ACTION$_1$	// assume takes 20 bytes
120:	ST 364, #140	// save return address 140
132:	BR 200	// call p
140:	ACTION$_2$	
160:	HALT	// Terminate, return to OS
		// code for p
200:	ACTION$_3$	
220:	BR *364	// return to address saved in location 364

stack area with activation records

		// 300-363 hold activation record for c
300:		// return address
304:		// local data for c
		// 364-451 hold activation record for p
364:	140	// return address
368:		// local data for p

Stack Allocation

Code for first procedure	
LD SP, $\#stackStart$	// initialize the stack
code	
HALT	// terminate execution

Code for procedure call	
ADD SP, SP, $\#caller.recordSize$	// increment stack pointer
ST $*$SP, $\#here + 16$	// save return address in
	// callee's frame
BR $callee.codeArea$	// jump to caller

Code for return sequence in the callee	
BR $*$0(SP)	// return to caller

Code for return sequence in the caller	
SUB SP, SP, $\#caller.recordSize$	// decrement stack pointer

3AC

```
// code for s
action$_1$
call q
action$_2$
halt

// code for p
action$_3$
return

// code for q
action$_4$
call p
action$_5$
call q
action$_6$
call q
return
```

Target Code for Stack Allocation

		// code for s
100:	LD SP, #600	// initialize the stack
108:	ACTION$_1$	// code for action$_1$
128:	ADD SP, SP, #ssize	// call sequence begins
136:	ST 0(SP), #152	// push return address
144:	BR 300	// call q
152:	SUB SP, SP, #ssize	// restore SP
160:	ACTION$_2$	
180:	HALT	
		// code for p
200:	ACTION$_3$	
220:	BR *0(SP)	// return to caller

		// code for q
300:	ACTION$_4$	// conditional jump to 456
320:	ADD SP, SP, #qsize	
328:	ST 0(SP), #344	// push return address
336:	BR 200	// call p
344:	SUB SP, SP, #qsize	
352:	ACTION$_5$	
372:	ADD SP, SP, #qsize	
380:	ST 0(SP), #396	// push return address
388:	BR 300	// call q
396:	SUB SP, SP, #qsize	
404:	ACTION$_6$	
424:	ADD SP, SP, #qsize	
432:	ST 0(SP), #448	// push return address
440:	BR 300	// call q
448:	SUB SP, SP, #qsize	
456:	BR *0(SP)	// return
600:		// stack starts here

Basic Blocks and Flow Graphs

Basic Block

- Maximal sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end
  - Entry is to the start of the BB, and exit is from the end of the BB
  - Only the start/leader instruction can be the target of a JUMP instruction
  - No jumps into the middle of the block
  - No branch instructions other than the end

$t_1 = a * a$
$t_2 = a * b$
$t_3 = 2 * t_2$
$t_4 = t_1 + t_3$
$t_5 = b * b$
$t_6 = t_4 + t_5$

Identifying Basic Blocks (BBs)

- **Input**
  - A sequence of 3AC
- **Output**
  - List of BBs with each 3AC in exactly one BB
- **Algorithm**
  - Identify the leaders which are the first statements in a BB
    1. The first statement is a leader
    2. Any statement that is the target of a conditional or unconditional goto is a leader
    3. Any statement that immediately follows a conditional or unconditional goto is a leader
  - For each leader, its BB consists of the leader and all instructions up to but not including the next leader or the end of the program

Identifying BBs

```
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j] = 0.0
for i from 1 to 10 do
  a[i,i]=1.0
```

Statements (1), (2), (3), (10), (12), and (13) are leaders

There are six BBs: (1), (2), (3)-(9), (10)-(11), (12), (13)-(17)

```
(1)  $i = 1$
(2)  $j = 1$
(3)  $t_1 = 10 \times i$
(4)  $t_2 = t_1 + j$
(5)  $t_3 = 8 \times t_2$
(6)  $t_4 = t_3 - 88$
(7)  $a[t_4] = 0.0$
(8)  $j = j + 1$
(9)  if $j \leq 10$ goto (3)
(10) $i = i + 1$
(11) if $i \leq 10$ goto (2)
(12) $i = 1$
(13) $t_5 = i - 1$
(14) $t_6 = 88 \times t_5$
(15) $a[t_6] = 1.0$
(16) $i = i + 1$
(17) if $i \leq 10$ goto (13)
```

target

follows a conditional

Next Use and Liveness

- Knowing when the value of a variable will be **used next** is important for generating good code
    - Remove variables from registers if not used
- Consider the 3AC instruction I: $x = y + z$; we say I defines x and uses y and z
- Suppose a statement I defines x. If a statement J uses x as an operand, and control can flow from I to J along a path where x is not redefined, then J uses the value of x defined at I
- A name in a BB is live at a given point if its value is used after that point
    - We say x is live at statement I

no further use

```
X = Y + Z
Z = Z * 5
t7 = Z + 1
Y = Z − t7
X = Z + Y
```

X is live at (5), X’s next use at (5) is (15)

```
(5)  X = …

    (no redefinition of X)

(15) … = … X …

(25) X = …
```

X is dead at (15) because there is no further use

Example of Next Use and Liveness

Intermediate Code	Live/Dead			Next use		
	x	y	z	x	y	z
(1) x=y+z	L	D	D	(2)	-	-
(2) z=x*5	D		L	-		(3)
(3) y=z−7		L	L		(5)	(5)
(4) x=z+y	D	D	D	-	-	-

Determining Next Use and Liveness Information

- **Input**
  - A BB (say B) of 3AC
  - Assume symbol table shows all non-temporary variables in B as live on exit and all temporaries are dead on exit
- **Output**
  - Liveness and next use information for each statement $I\colon x = y \text{ op } z$ in B
- **Algorithm**

  i. Scan forward over B to find the last statement.
     - For each variable x used in B, create fields x.live and x.next_use in the symbol table. Initialize x.live = FALSE and x.next_use = NONE.
     - Each tuple $I\colon x = y \text{ op } z$ stores next use and liveness information. Initialize tuple.

  ii. Scan backward over B. For each statement $I\colon x = y \text{ op } z$ in B, do
     - Copy the next use and liveness information for x, y, and z from the symbol table to tuple I
     - Update x, y, and z's symbol table entries.
       - Set x.live = FALSE, x.next_use = NONE
       - Set y.live=z.live = TRUE and y.next_use = z.next_use = I

Example Computation of Next Use and Liveness Information

Intermediate Code	Symbol Table Information						Instruction Information					
	Live			Next use			Live			Next use		
	x	y	z	x	y	z	x	y	z	x	y	z
(1) x=y+z	F	F	F	N	N	N	F	F	F	N	N	N
(2) z=x*5	F	F	F	N	N	N	F	F	F	N	N	N
(3) y=z-7	F	F	F	N	N	N	F	F	F	N	N	N
(4) x=z+y	F	F	F	N	N	N	F	F	F	N	N	N

after the forward pass

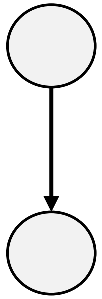
Example Computation of Next Use and Liveness Information

Intermediate Code	Symbol Table Information						Instruction Information					
	Live			Next use			Live			Next use		
	x	y	z	x	y	z	x	y	z	x	y	z
`(4) x=z+y`	F	T	T	N	(4)	(4)	F	F	F	N	N	N
`(3) y=z-7`	F	F	T	N	N	(3)	F	T	T	N	(4)	(4)
`(2) z=x*5`	T	F	F	(2)	N	N	F	F	T	N	N	(3)
`(1) x=y+z`	F	T	T	N	(1)	(1)	T	F	F	(2)	N	N

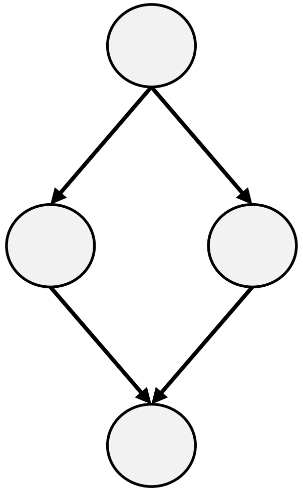
after the backward pass

Control Flow Graph (CFG)

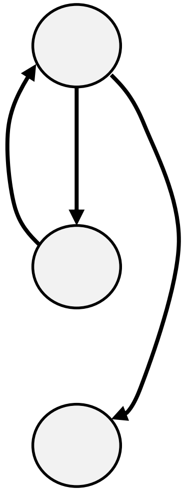
- Graphical representation of control flow during execution
  - Each node represents a statement or a BB
  - An entry and an exit node are often added to a CFG for a function
  - An edge represents possible transfer of control between nodes
- Used for compiler optimizations and static analysis (e.g., instruction scheduling and global register allocation)



straight-line code



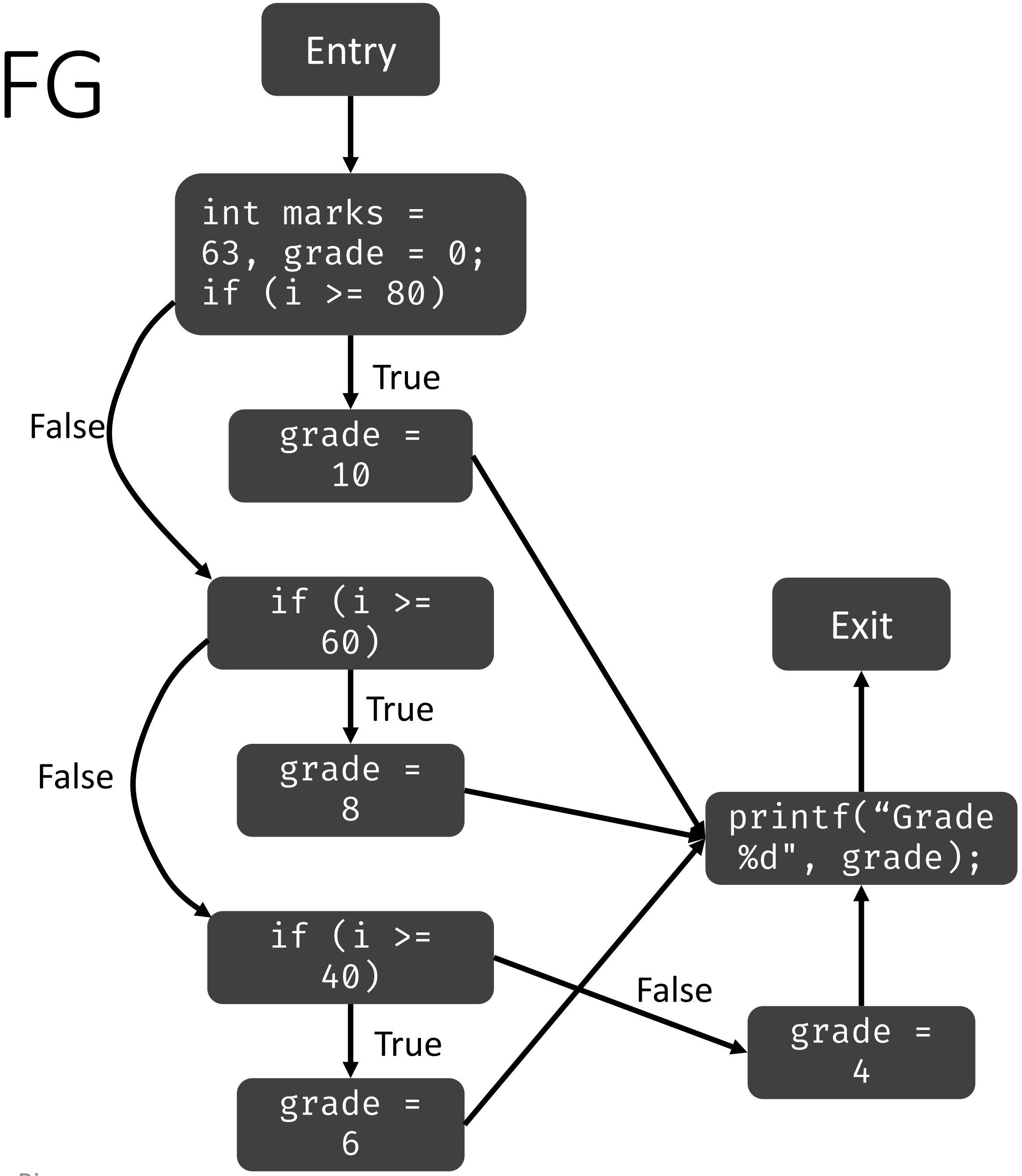
predicate



loop iteration

Example of BBs and a CFG

```
int main() {
    int marks = 63, grade = 0;
    if (marks >= 80)
       grade = 10;
    else if (marks >= 60)
       grade = 8;
    else if (marks >= 40)
       grade = 6;
    else
       grade = 4;
    printf(“Grade %d", grade);
    return 0;
}
```



Entry

int marks = 63, grade = 0; if (i >= 80)

True

False

grade = 10

if (i >= 60)

True

False

grade = 8

if (i >= 40)

True

False

grade = 6

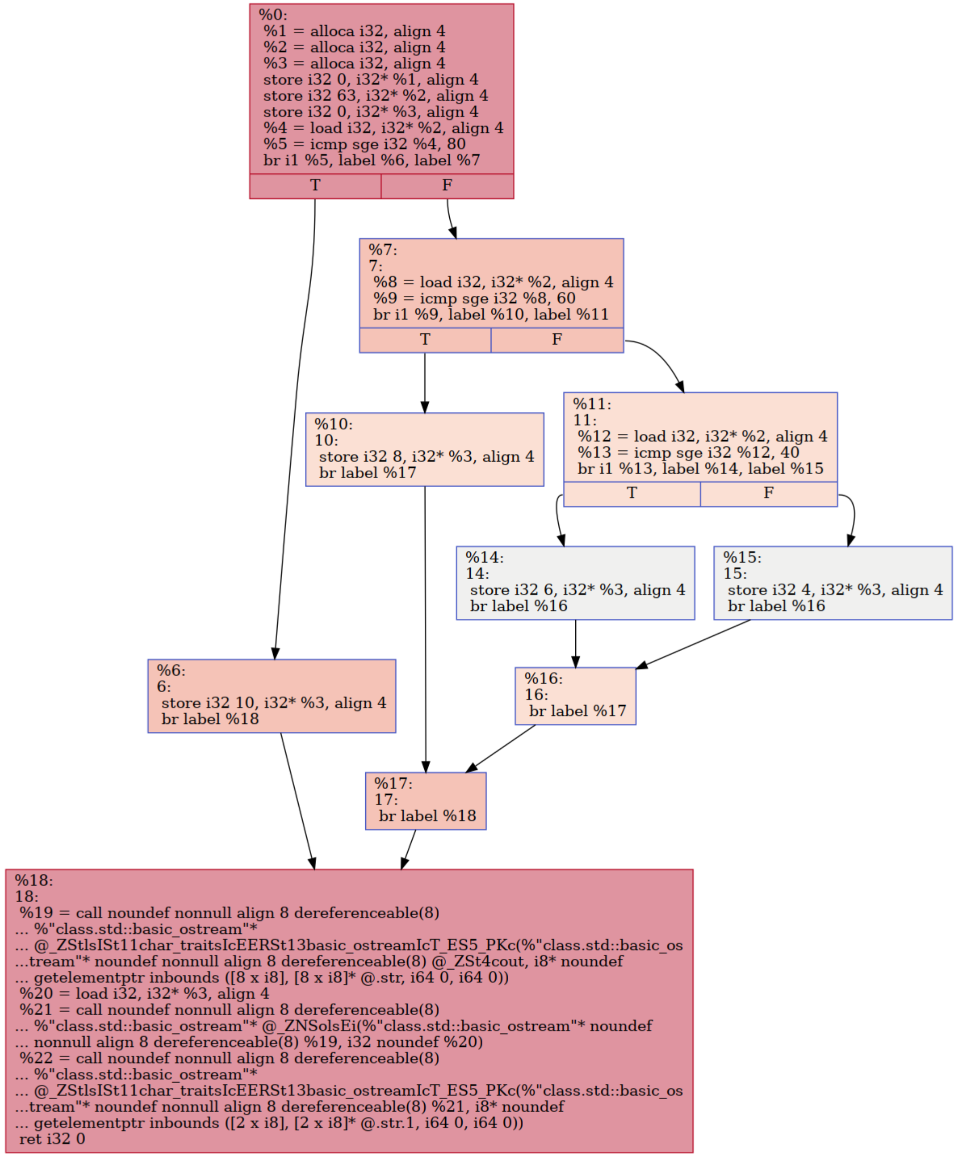
grade = 4

printf(“Grade %d", grade);

Exit

Example CFG Generated with LLVM

```
int main() {
    int marks = 63, grade = 0;
    if (marks >= 80)
      grade = 10;
    else if (marks >= 60)
      grade = 8;
    else if (marks >= 40)
      grade = 6;
    else
      grade = 4;
    printf(“Grade %d", grade);
    return 0;
}
```



```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 63, i32* %2, align 4
 store i32 0, i32* %3, align 4
 %4 = load i32, i32* %2, align 4
 %5 = icmp sge i32 %4, 80
 br i1 %5, label %6, label %7
T | F
```

```
%7:
7:
 %8 = load i32, i32* %2, align 4
 %9 = icmp sge i32 %8, 60
 br i1 %9, label %10, label %11
T | F
```

```
%10:
10:
 store i32 8, i32* %3, align 4
 br label %17
```

```
%11:
11:
 %12 = load i32, i32* %2, align 4
 %13 = icmp sge i32 %12, 40
 br i1 %13, label %14, label %15
T | F
```

```
%14:
14:
 store i32 6, i32* %3, align 4
 br label %16
```

```
%15:
15:
 store i32 4, i32* %3, align 4
 br label %16
```

```
%6:
6:
 store i32 10, i32* %3, align 4
 br label %18
```

```
%16:
16:
 br label %17
```

```
%17:
17:
 br label %18
```

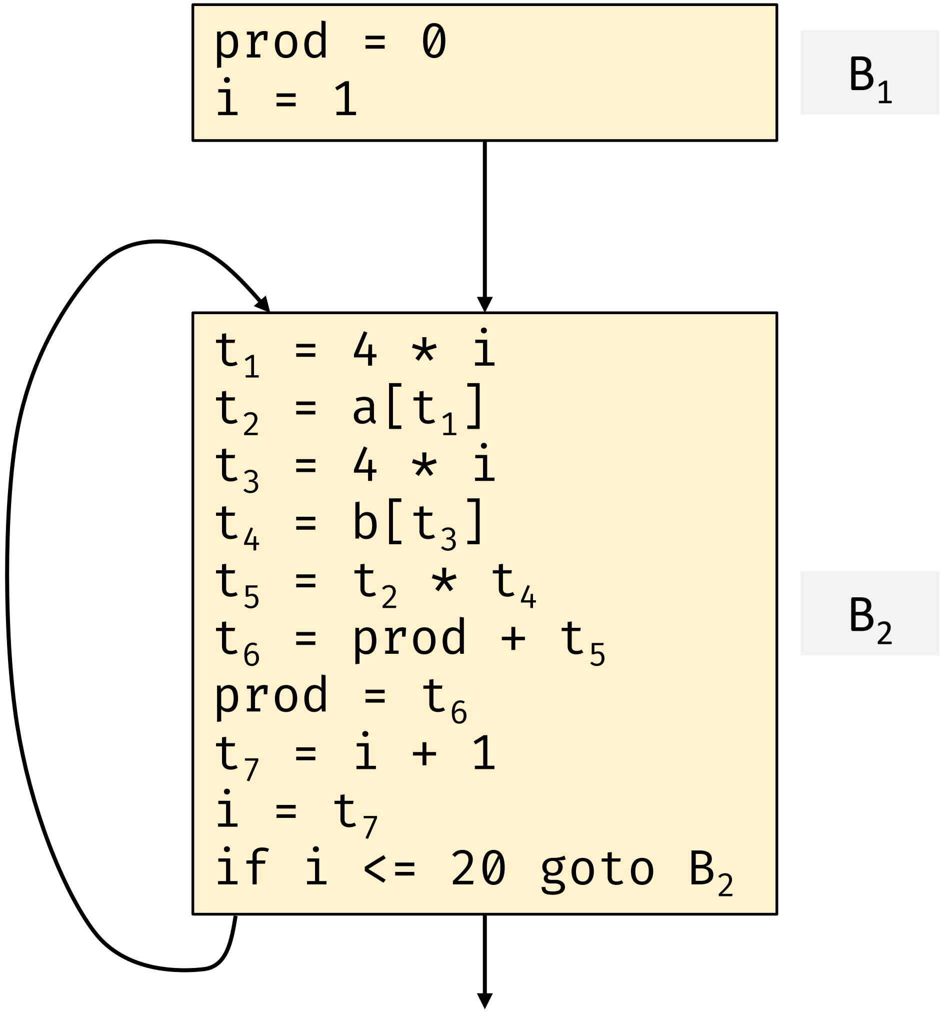
```
%18:
18:
 %19 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"*
... @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_os
...tream"* noundef nonnull align 8 dereferenceable(8) @_ZSt4cout, i8* noundef
... getelementptr inbounds ([8 x i8], [8 x i8]* @.str, i64 0, i64 0))
 %20 = load i32, i32* %3, align 4
 %21 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"* @_ZNSolsEi(%"class.std::basic_ostream"* noundef
... nonnull align 8 dereferenceable(8) %19, i32 noundef %20)
 %22 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"*
... @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_os
...tream"* noundef nonnull align 8 dereferenceable(8) %21, i8* noundef
... getelementptr inbounds ([2 x i8], [2 x i8]* @.str.1, i64 0, i64 0))
 ret i32 0
```

CFG for 'main' function

[Control Flow Graph generator for code in C++]

Control Flow Graph (CFG)

1.	prod = 0
2.	i = 1
3.	t_1 = 4 * i
4.	t_2 = a[t1]
5.	t_3 = 4 * i
6.	t_4 = b[t_3]
7.	t_5 = t_2 * t_4
8.	t_6 = prod + t_5
9.	prod = t_6
10.	t_7 = i + 1
11.	i = t_7
12.	if i <= 20 goto (3)

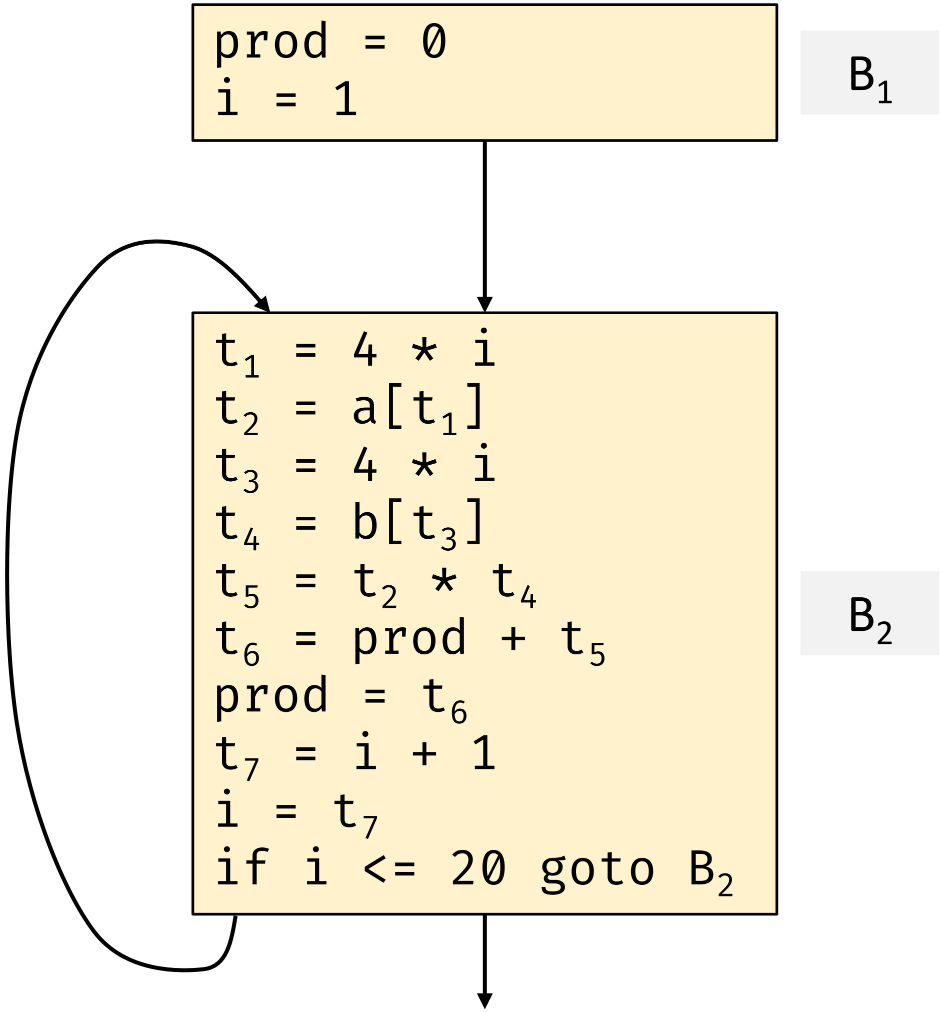


prod = 0
i = 1

t_1 = 4 * i
t_2 = a[t_1]
t_3 = 4 * i
t_4 = b[t_3]
t_5 = t_2 * t_4
t_6 = prod + t_5
prod = t_6
t_7 = i + 1
i = t_7
if i <= 20 goto B_2

Loops in a CFG

- A set of CFG nodes L form a loop if that L contains a node e called loop entry such that
  - e is not the Entry node,
  - No node in L besides e has a predecessor outside L
  - Every node in L has a nonempty path to e that is completely within L



prod = 0
i = 1

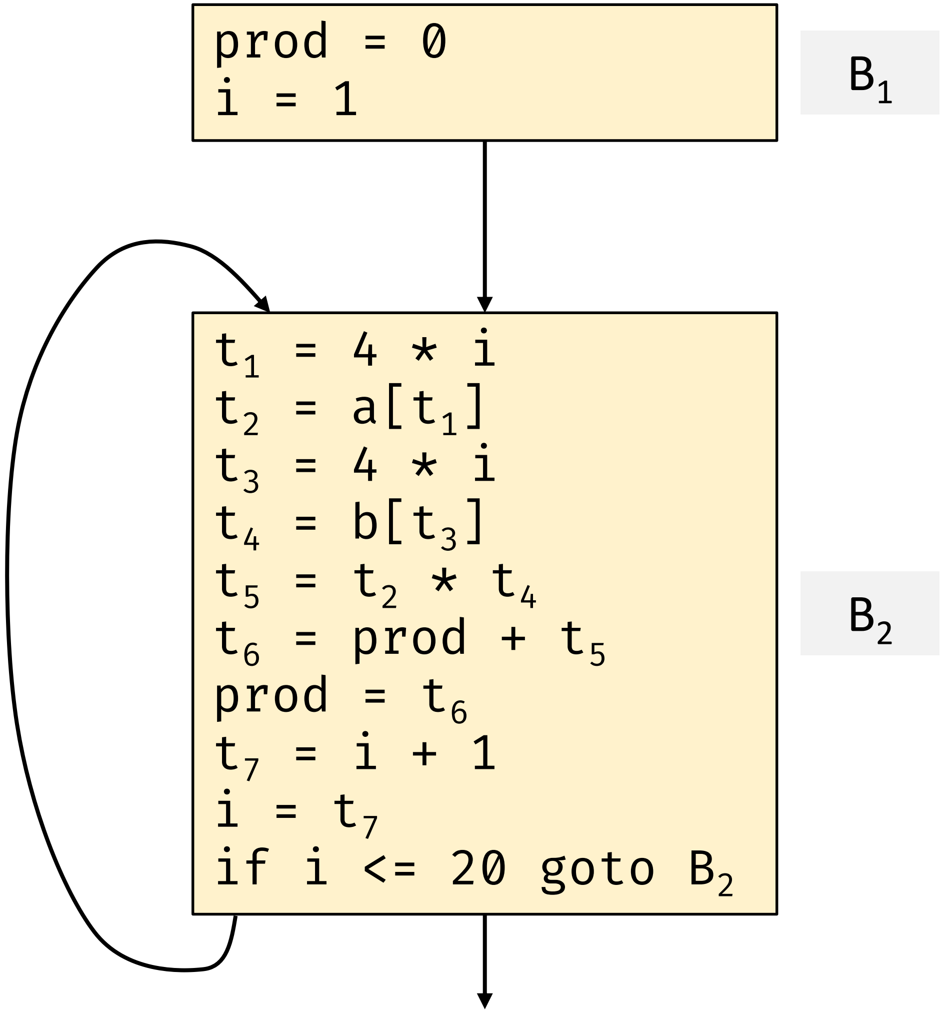
B_1

t_1 = 4 * i
t_2 = a[t_1]
t_3 = 4 * i
t_4 = b[t_3]
t_5 = t_2 * t_4
t_6 = prod + t_5
prod = t_6
t_7 = i + 1
i = t_7
if i <= 20 goto B_2

B_2

Loops in a CFG

- There is a unique entry
  - Only way to reach a node in L from outside the loop is through e
- All nodes in the group are strongly connected
  - There is a path from any node in the loop to any other loop
  - Path is wholly-contained within the loop

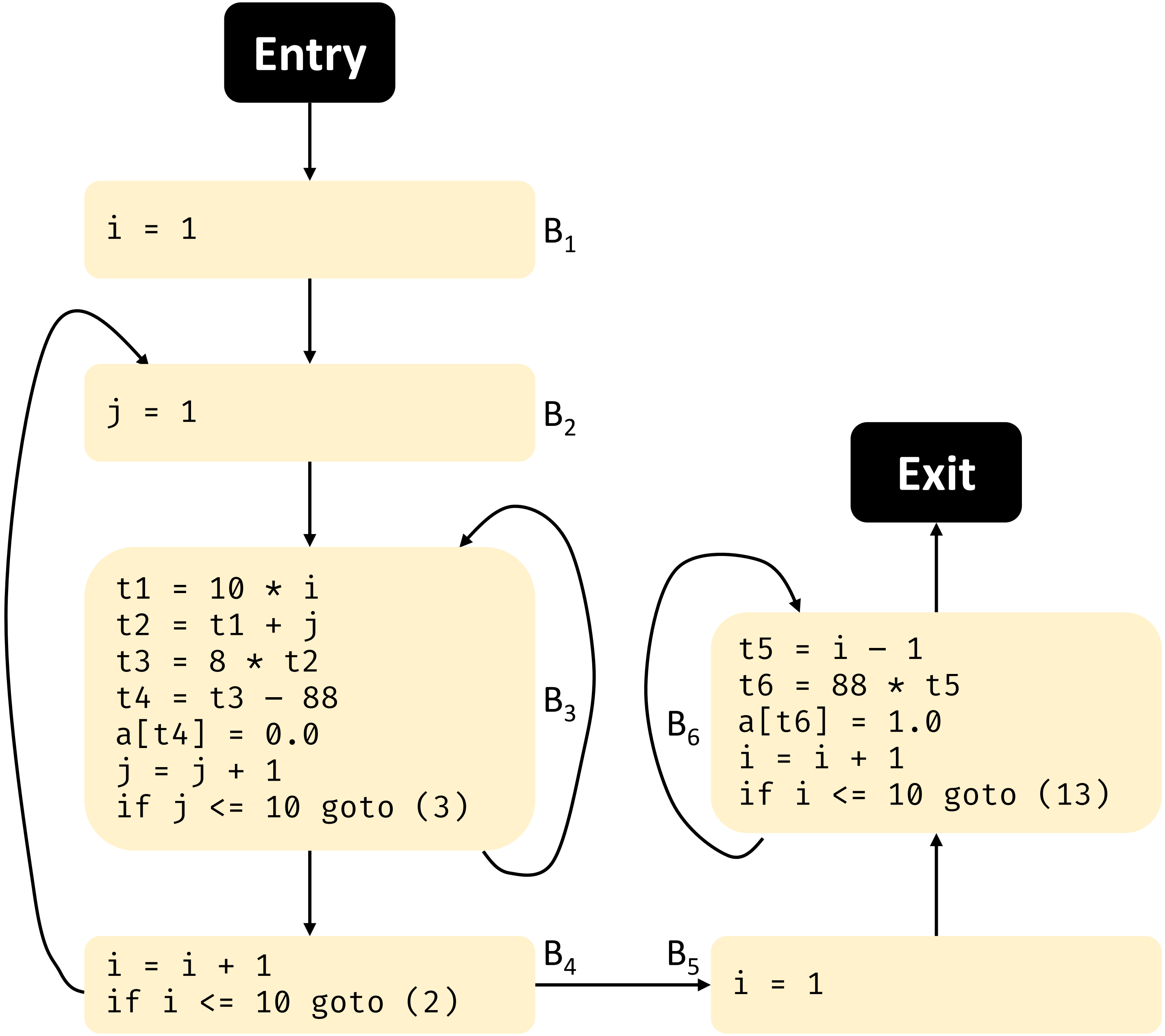


```
prod = 0
i = 1
```

```
t1 = 4 * i
t2 = a[t1]
t3 = 4 * i
t4 = b[t3]
t5 = t2 * t4
t6 = prod + t5
prod = t6
t7 = i + 1
i = t7
if i <= 20 goto B2
```

Example CFG

```
1) i = 1  // Leader
2) j= 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Optimizing BBs

Local optimizations

Optimization of BBs

- Code optimizations can lead to substantial improvement in running time and/or energy consumption
- Global optimization analyzes control and data flow **among BBs**
  - E.g., performs control flow, data flow, and data dependence analysis
- Local or intra-BB optimizations can also provide significant improvements
- DAG is a useful data structure for implementing transformations on BBs
  - Allows detecting common sub-expressions

Intra-Block Transformations

- Expressions are values of names that are live on exit from a BB
- Two BBs are equivalent if they compute the same set of expressions

- Local transformations on BBs to improve code quality
  - Structure-preserving and algebraic transformations
  - Should not change the set of expressions computed by a block

Structure-Preserving Transformations

i. Common subexpression elimination
   - Instructions compute a value that has been computed

```
a = b + c
b = a – d
c = b + c
d = a - d
```

```
a = b + c
b = a – d
c = b + c
d = b
```

ii. Dead code elimination
   - Remove Instructions that define variables that are never used

iii. Renaming temporary variables
   - Can always transform a BB into an equivalent block where each statement that defines a temporary uses a new name
     - Such a BB is called a normal-form block

iv. Reordering of dependence-free statements
   - Normal-form blocks permits statement interchanges without affecting the value of the block
   - May improve latency of accesses and register usage

$t_1 = b + c$

$t_2 = x + y$

Algebraic Transformations

- Apply algebraic laws to simplify computation

Strength Reduction	
Expensive	Cheaper
x^2	$x \times x$
$2 \times x$	$x + x$
$x \div 2$	$x \gg 1$

$$x + 0 = 0 + x = x$$
$$x \times 1 = 1 \times x = x$$
$$x - 0 = x$$
$$x \div 1 = 1$$

- Constant folding – evaluate constants during compilation
  - E.g., $i = 2 * 3.14 * 300 * 300;$
- Relational operators can generate common sub-expressions (e.g., $x > y$ and $x - y$)

DAG Representation of BBs

Many optimizations are easier to perform on a DAG representation of BBs

(1) t_1 = 4 * i

(2) t_2 = a[t_1]

(3) t_3 = 4 * i

(4) t_4 = b[t_3]

(5) t_5 = t2 * t_4

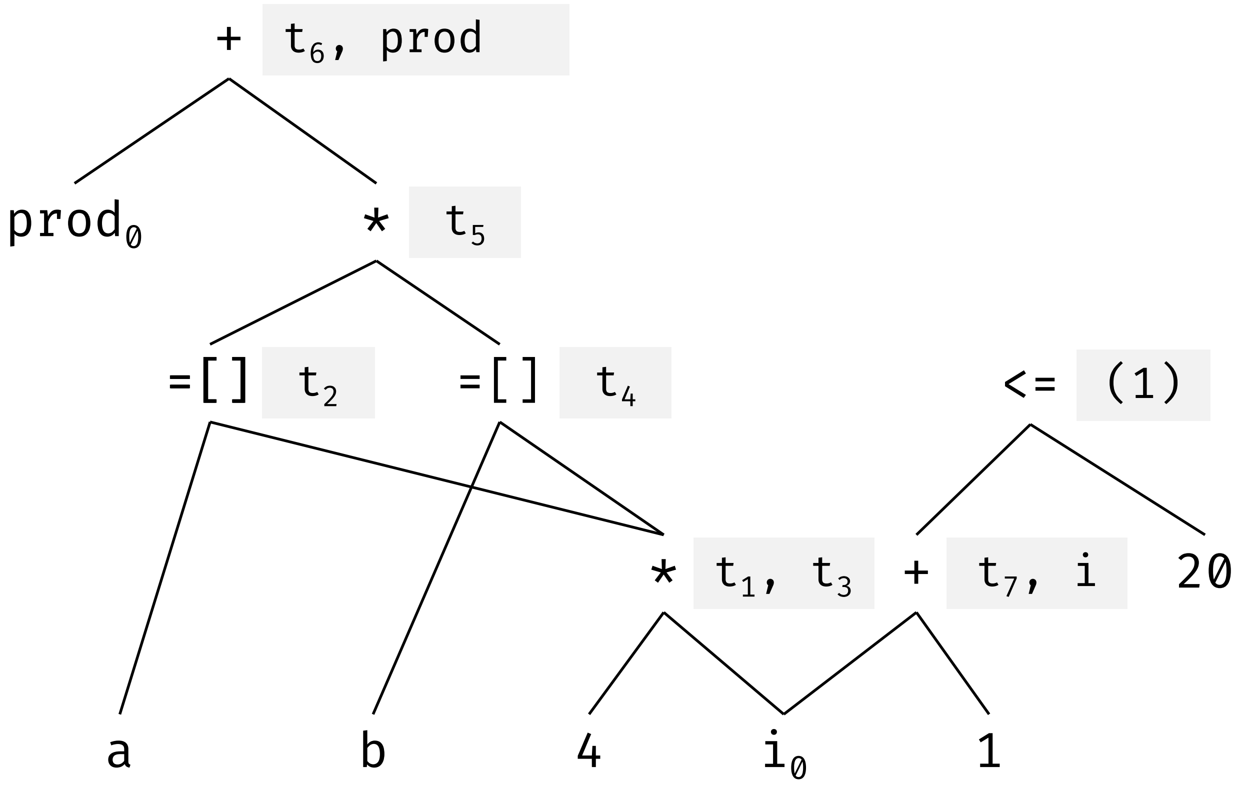
(6) t_6 = prod + t_5

(7) prod = t_6

(8) t_7 = i + 1

(9) i = t_7

(10) if i <= 20 goto (1)



Representing BBs with DAGs

- Rules on the DAG structure
  - Leave nodes are labeled with variable names or constants
  - Initial values for each variable is represented by a node
  - A node N is associated with each statement s in a BB
    - Children of N correspond to statements that last define the operands used in s
  - Inner nodes are labeled by an operator symbol
    - Node N is labeled by the operator applied at s
  - Nodes optionally have a sequence of identifiers for labels
  - Output nodes are those variables that are live on exit
- Each BB node in a CFG can be represented with a DAG

Constructing a DAG

- **Input**
  - A basic block (BB)
- **Output**
  - A DAG for the BB with the following information
    - a label for each node (id for leaf nodes and operator symbols for interior nodes)
    - a list of identifiers (not constants) for each node
- **Assumptions**
  - Three kinds of 3AC: (i) $x = y \text{ op } z$, (ii) $x = \text{op } y$, and (iii) $x = y$
  - Relational operators like "$\text{if } i \leq 20 \text{ goto } (1)$" are treated like case (i)

Constructing a DAG

- **Steps**
  - For each statement in the BB,
    1. If $node(y)$ is undefined, create a leaf labeled y and set $node(y)$ to the new node
    2. For case (i), check if there is a node in the DAG labeled op with left child $node(y)$ and right child $node(z)$. If not, then create a node (denoted by n).
    3. For case (ii), check if there is a node labeled op with $node(y)$ as the only child. If not, then create a node (denoted by n).
    4. Delete x from the list of identifiers for $node(x)$. Append x to the list of identifiers for the node and set $node(x)$ to n.

DAG Representation of BBs

(1) t_1 = 4 * i

(2) t_2 = a[t_1]

(3) t_3 = 4 * i

(4) t_4 = b[t_3]

(5) t_5 = t2 * t_4

(6) t_6 = prod + t_5

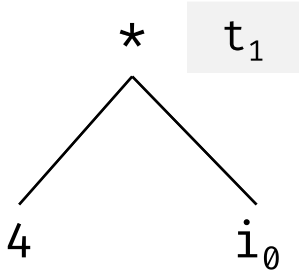
(7) prod = t_6

(8) t_7 = i + 1

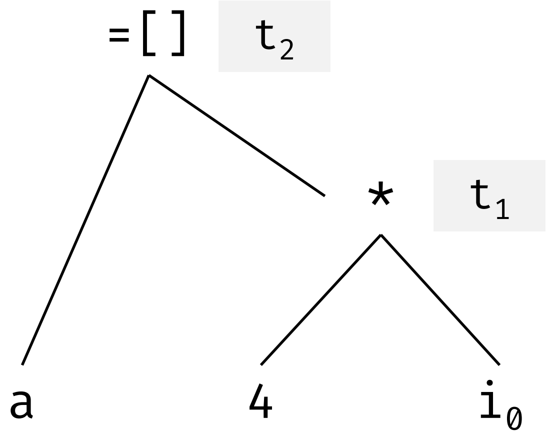
(9) i = t_7

(10) if i <= 20 goto (1)

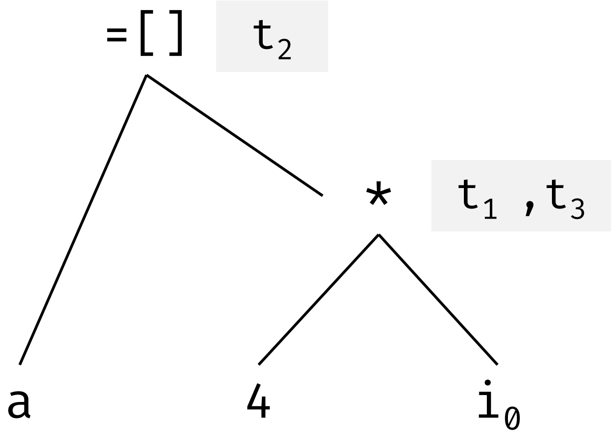
1



2

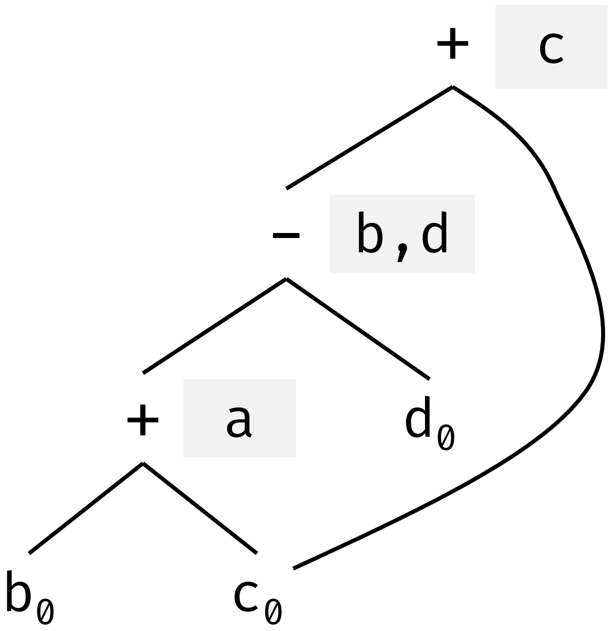


3



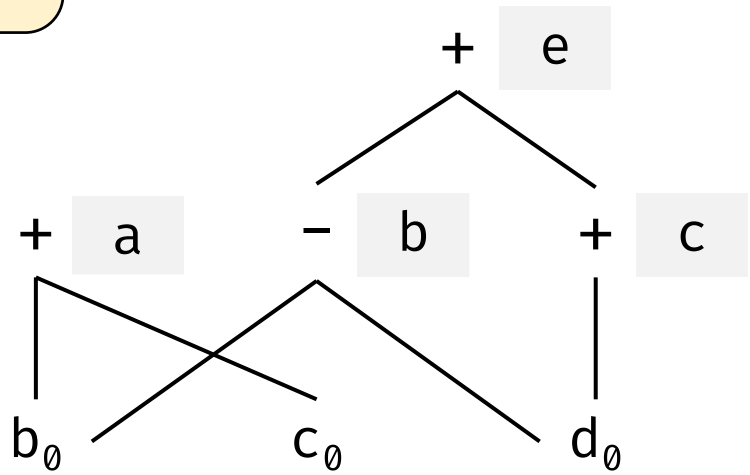
Local Common Subexpressions

```
a = b + c
b = a - d
c = b + c
d = a - d
```



```
a = b + c
b = b - d
c = c + d
e = b + c
```

DAG fails to capture that the 1st and 4th statements compute the same values

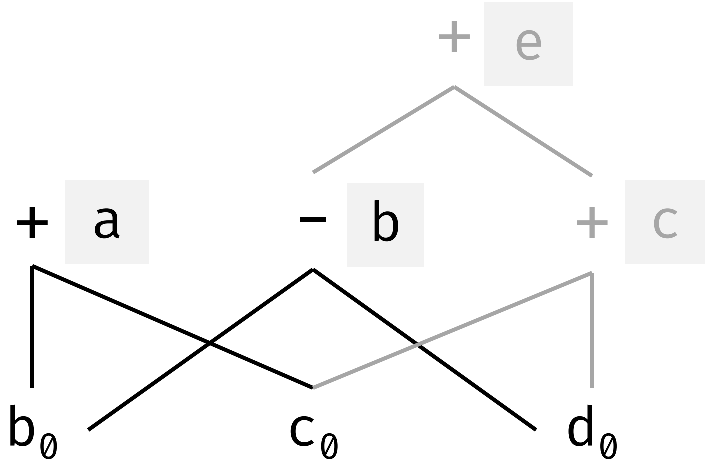


Dead Code Elimination

- Delete a root node from the DAG if it has no live variables
  - Repeat till there are no such nodes

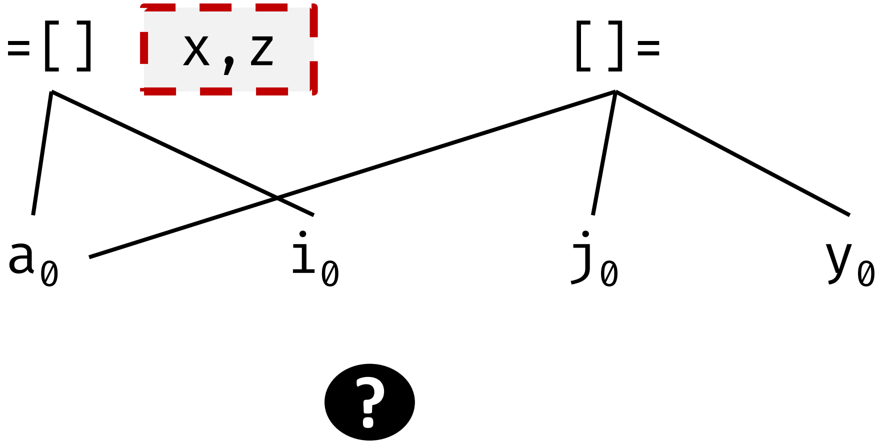
- Assume only a and b are live on exit

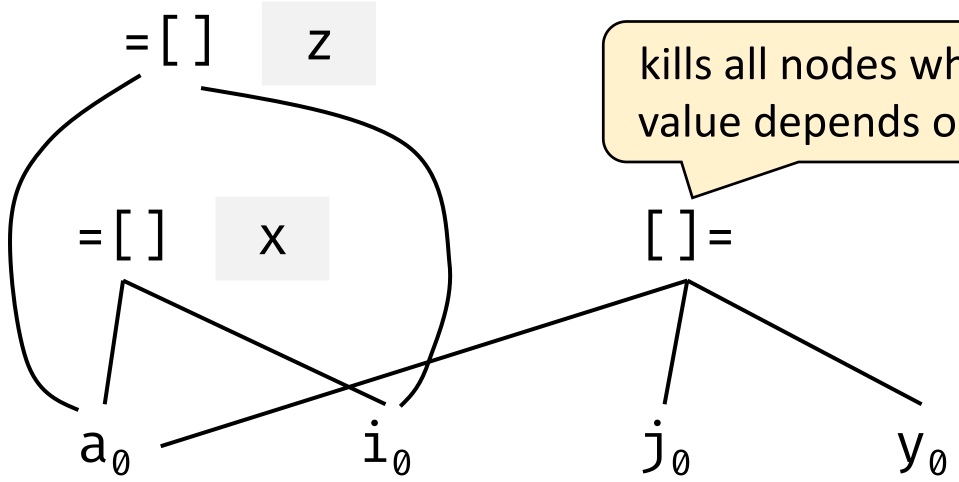
```
a = b + c
b = b - d
c = c + d
e = b + c
```



Representing Array References

```
x = a[i]
a[j] = y
z = a[i]
```

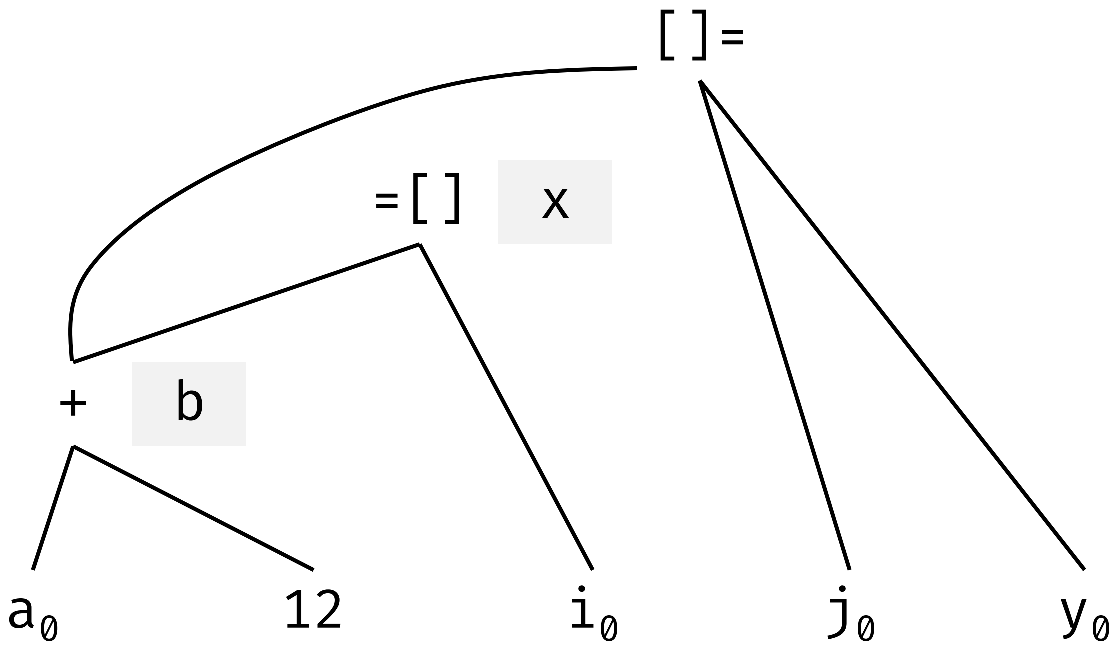




kills all nodes whose value depends on a_0

Consider Other Sources of Possible Aliasing

```
b = a + 12
x = b[i]
b[j] = y
```



`x = *p` // Use of every possible variable
`*q = y` // Possible assignment to every variable

- `=*` must include all nodes for optimization analysis
- `*=` kills all other nodes
- Possible to use pointer analysis to be more precise

- Assume that procedures use variables attached to a node and kills that node

Code Generation Algorithm

Single Basic Blocks

Code Generation Strategy

- **Goal**: Generate target code for a sequence of 3AC within a BB
- **Assumptions**
  - Every 3AC operator has an equivalent operator in the target language
  - Computed values can reside in registers and only needs to be saved when
    1. The register is required for another computation, or
    2. Just before a procedure call, jump, or a labelled statement
       - Implies every register must be saved before the end of a BB
- **Steps**: For each 3AC,
  - Identify variables that need to be loaded into registers
  - Load the variables into registers
  - Generate code for the instruction
  - Generate a store if the result needs to be saved into memory

Challenges in Code Generation

Different Possibilities		
a = b + c	ADD Rj, Ri	b is in Ri, c is in Rj, b is no longer live on exit
	ADD c, Ri	b is in Ri, b is no longer live on exit
	MOV c, Rj ADD Rj, Ri	b is in Ri, b is no longer live on exit

Usually there will be numerous cases to consider

- An efficient choice depends on a number of factors (e.g., frequency of use of b and c later)
- Properties of the operator (e.g., commutativity) can add to the complexity

A Simple Code Generator

- Treat each IR quadruple as a “macro”
- Replace the macro with pre-existing code templates

```
a = b + c
```

→

```
LD R1, b
LD R2, c
ADD R1, R2
ST A, R1
```

or

```
LD R1, b
ADD R1, c
ST A, R1
```

- Simple to implement but makes inefficient use of registers
- **Goal**: Track values in registers and reuse them

Register and Address Descriptors

Register descriptor

- Keeps track of **what name is stored in each register**, consulted whenever a new register is needed
- Each register holds the value of zero or more names at any time during execution

Address descriptor

- Keeps track of the **location(s) where the current value of a name** can be found at runtime
  - Location can be a register, a stack location, a memory address, or some combination of these (data can get copied)
- Information can be stored in the symbol table

Code Generation Algorithm

- For each 3AC instruction I of the form $x = y \text{ op } z$,
  - Invoke function $getreg(I)$ to select registers R_x, R_y, and R_z
  - If y is not in R_y according to the address descriptor, then generate instruction $\text{LD } R_y, y'$
    - y' is one of the memory locations for y
  - Perform the same steps for z
  - Generate the instruction $\text{OP } R_x, R_y, R_z$
- For a 3AC copy instruction $x = y$,
  - If y is not in R_y according to the address descriptor, then generate instruction $\text{LD } R_y, y'$
  - Adjust the register descriptor for R_y to include x

Managing Register and Address Descriptors

- For an instruction $\mathrm{LD}\ R, x$,
  - Change the register descriptor for R so it holds only x
  - Change the address descriptor for x by adding register R as an additional location
- For instruction $\mathrm{ST}\ x, R$, change the address descriptor for x to include its own memory location

Managing Register and Address Descriptors

- For an instruction such as $\mathrm{ADD}\ R_x, R_y, R_z$, implementing a 3AC $x = y + z$,
  - Change the register descriptor for R_x so that it holds only x
  - Change the address descriptor for x so that its only location is R_x
    - The memory location for x is no longer in the address descriptor for x
  - Remove R_x from the address descriptor of any variable other than x
- For a copy instruction $x = y$, remember to
  - Process the load from y into a register (if needed)
  - Add x to the register descriptor for R_y
  - Change the address descriptor for x so that its only location is R_y

Usage of Registers

- Leave the computed result in a register for as long as possible
- Store the result only at the end of a BB or when the register is needed for another computation
    - A variable is live at a point if it is used (possibly in later BBs) later, requires global dataflow analysis
    - On exit from a BB, store only live variables which are not already in their memory locations (use address descriptors to identify)
    - If liveness information is not available, then assume that all variables are live at all times

Defining Function $getreg()$

- **Input**
    - 3AC I: $x = y \text{ op } z$
- **Output**
    - Returns registers to hold the value of x, y, and z

- We assume that there is no global register allocation

$getreg()$: Choosing R_y for y

i. If y is in a register, then return the register containing y as R_y

ii. If y is not in a register, but there is an empty register available, then pick one such register as R_y

iii. If y is not in a register and there are no empty registers, then
   - Let R be a candidate register and suppose v is one of the variables stored in R
     - Heuristic for candidate selection can be based on farthest references or fewest next use
     - If the address descriptor for v says that v is somewhere else beside R, then choose R
     - If v is x, and x is not an operand of I (i.e., $x \neq z$), then choose R
     - If v is not used later, then choose R
     - Else, generate $\mathrm{ST}\ v, R$ (called a register spill)
   - R may hold several variables, so we need to repeat the previous steps for each variable
   - Compute the number of store instructions generated for R (i.e., score) for each variable
   - Pick the register with the lowest score

iv. Selecting R_z for z is similar

$getreg()$: Choosing R_x for x

- Consider selection of a register R_x for x. In addition to the previous checks, try the following.
  - A register that holds only x is always an acceptable choice for R_x
  - If y is not used after instruction I, and R_y holds only y after being loaded, then R_y can also be used for R_x
  - Perform similar checks with R_z if required

- If I is a copy instruction, then always choose R_y

Code Generation Example

3AC	Generated Code	Register Descriptor			Address Descriptor						
		R1	R2	R3	a	b	c	d	t	u	v
					a	b	c	d			
$t = a - b$	LD $R1, a$ LD $R2, b$ SUB $R2, R1, R2$										
		a	t		$a, R1$	b	c	d	$R2$		
$u = a - c$	LD $R3, c$ SUB $R1$ $R1, R3$										
		u	t	c	a	b	$c, R3$	d	$R2$	$R1$	
$v = t + u$	ADD $R3, R2, R1$										
		u	t	v	a	b	c	d	$R2$	$R1$	$R3$
$a = d$	LD $R2, d$										
		u	a, d	v	$R2$	b	c	$d, R2$		$R1$	$R3$

R2 is reused since there is no next use of b

in memory, live at the end of BB

temporaries, not live at the end of BB

Code Generation Example

3AC	Generated Code	Register Descriptor			Address Descriptor						
		R1	R2	R3	a	b	c	d	t	u	v
		u	a, d	v	$R2$	b	c	$d, R2$		$R1$	$R3$
$d = v + u$	ADD $R1, R3, R1$										
		d	a	v	$R2$	b	c	$R1$			$R3$
exit	ST $a, R2$ ST $d, R1$										
		d	a	v	$a, R2$	b	c	$d, R1$			$R3$

Code Sequences for Indexed and Pointer Assignments

3AC	$\boldsymbol{i}$ in register $\boldsymbol{Ri}$	$\boldsymbol{i}$ in memory $\boldsymbol{Mi}$	$\boldsymbol{i}$ in Stack
$a = b[i]$	MOV $b(Ri), R$	MOV Mi, R MOV $b(R), R$	MOV $Si(A), R$ MOV $b(R), R$
$a[i] = b$	MOV $b, a(Ri)$	MOV Mi, R MOV $b, a(R)$	MOV $Si(A), R$ MOV $b, a(R)$

3AC	$\boldsymbol{p}$ in register $\boldsymbol{Rp}$	$\boldsymbol{p}$ in memory $\boldsymbol{Mp}$	$\boldsymbol{p}$ in Stack
$a = *p$	MOV $*Rp, a$	MOV Mp, R MOV $*R, R$	MOV $Sp(A), R$ MOV $*R, R$
$*p = b$	MOV $a, *Rp$	MOV Mp, R MOV $a, *R$	MOV a, R MOV $R, *Sp(A)$

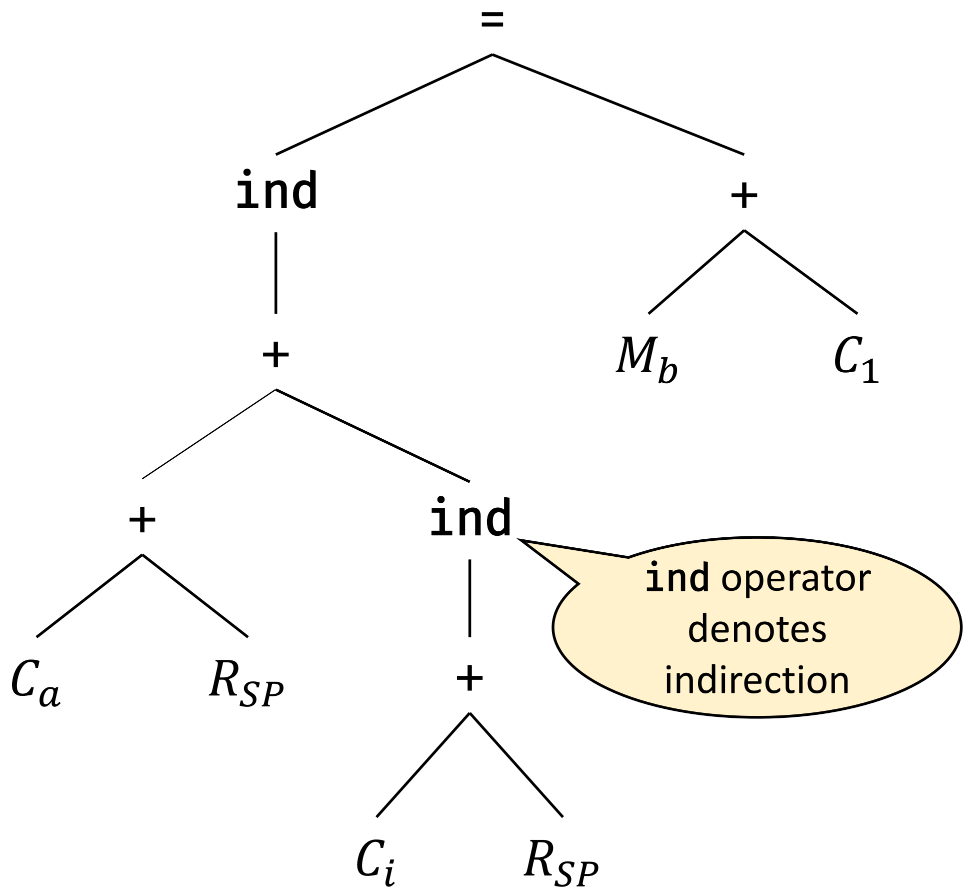
Instruction Selection by Tree Rewriting

Tree Representation

- Consider the statement

$$a[i] = b + 1$$

  - Assume b is in memory location M_b
  - Array of chars a is a local and is stored on the stack
  - SP points to the beginning of the current activation record
  - Addresses of locals a and i are given as constant offsets C_a and C_i from the SP



`ind` operator denotes indirection

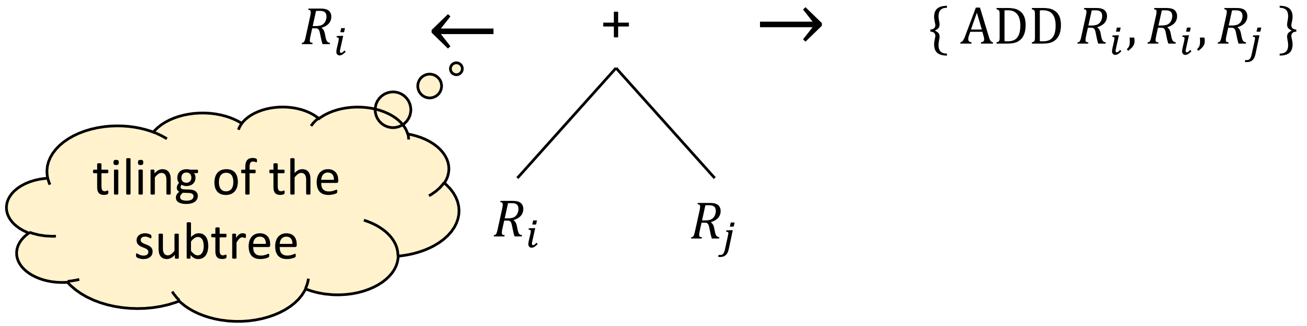
Tree Rewriting

- Target code can be generated by applying a sequence of tree-rewriting rules to reduce the input tree to a single node
- Each rewrite rule is of the form

$$replacement \leftarrow template \,\{\, action \,\}$$

where $replacement$ is a single node, $template$ is a tree, and $action$ is a code fragment like in a SDT

- A set of tree rewriting rules is called a tree-translation scheme

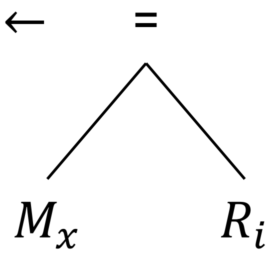


Tree Rewriting Rules

1 $R_i \leftarrow C_a$ $\{ \text{LD } R_i, \#a \}$

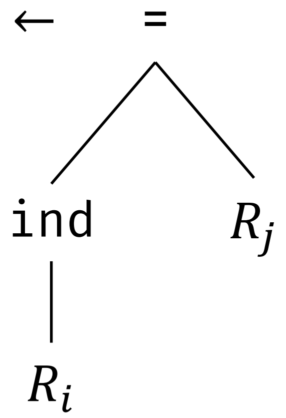
2 $R_i \leftarrow M_x$ $\{ \text{LD } R_i, x \}$

3 $M \leftarrow$



$\{ \text{ST } x, R_i \}$

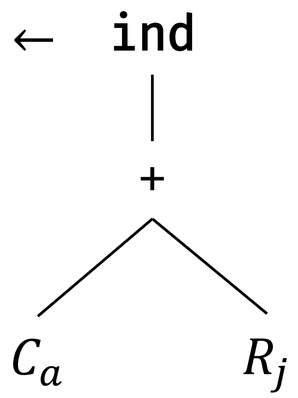
4 $M \leftarrow$



$\{ \text{ST } *R_i, R_i \}$

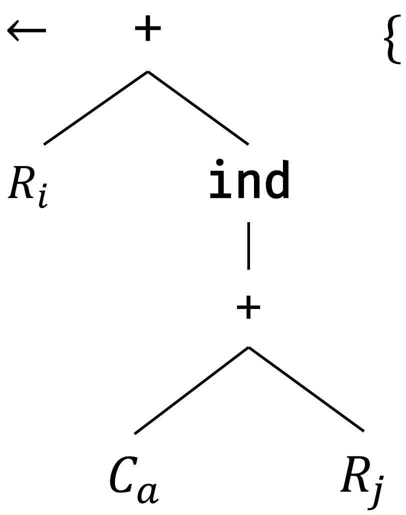
Tree Rewriting Rules

5 $R_i \leftarrow$



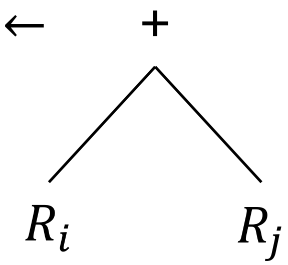
$\{ \text{LD } R_i, a(R_j) \}$

6 $R_i \leftarrow$



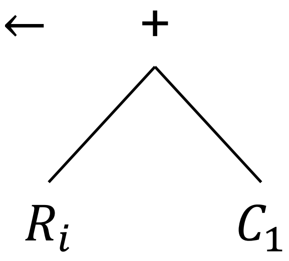
$\{ \text{ADD } R_i, R_i, a\left(R_j\right) \}$

7 $R_i \leftarrow$



$\{ \text{ADD } R_i, R_i, R_j \}$

8 $R_i \leftarrow$

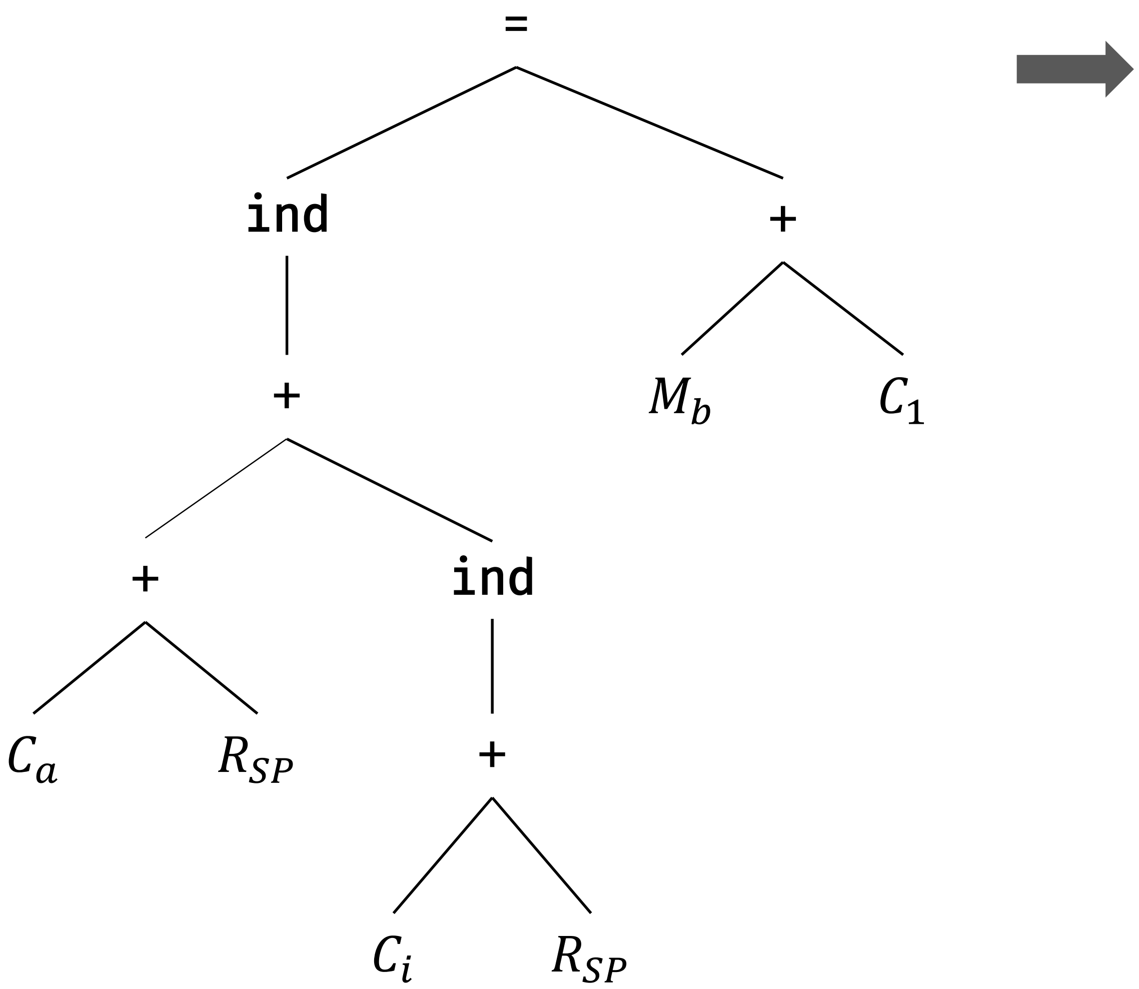


$\{ \text{INC } R_i \}$

Code Generation by Tiling an Input Tree

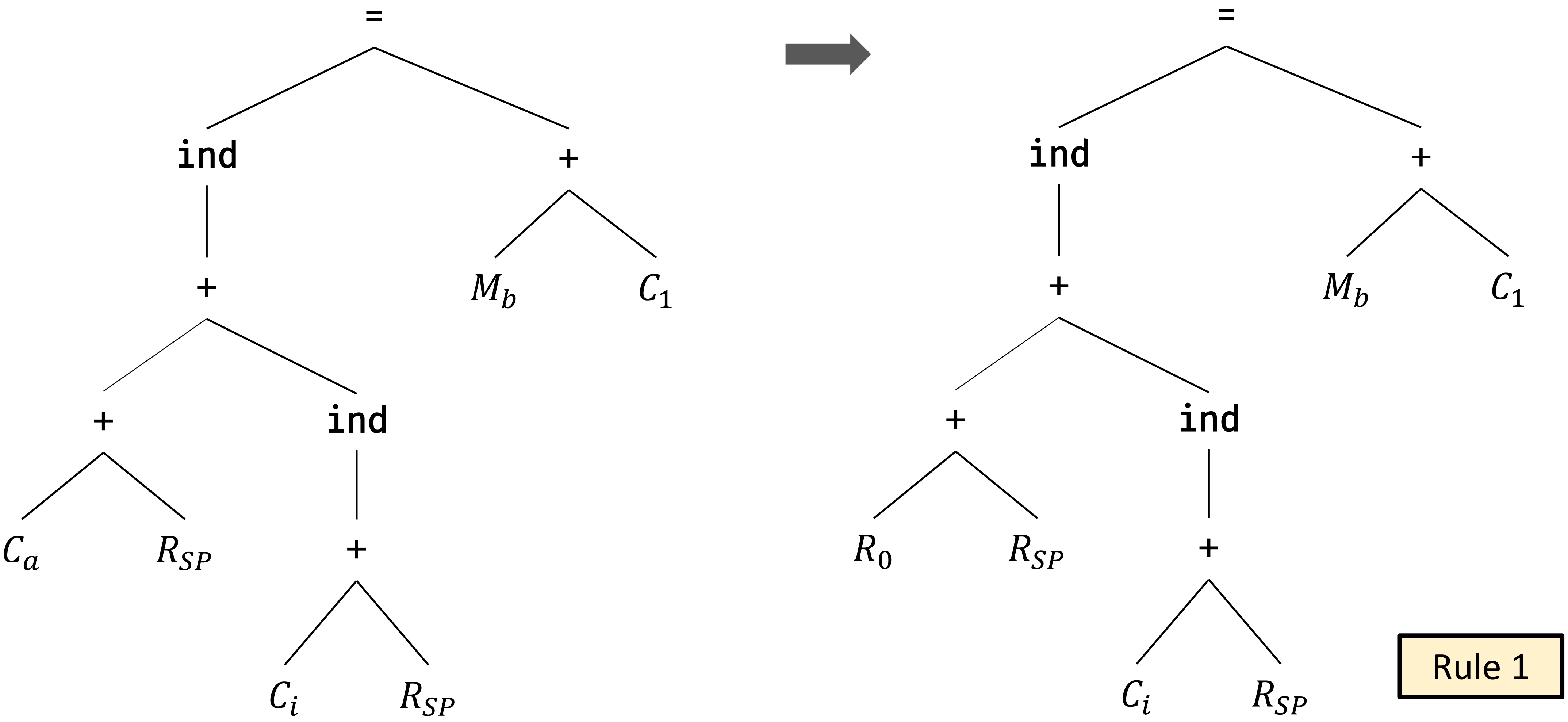
- High-level steps in a tree-translation scheme
  - Given an input tree, the templates in the tree-rewriting rules are applied to tile its subtrees
  - If a template matches, replace the matching subtree with the replacement node of the rule
    - Execute the action associated with the rule
    - If the action contains a sequence of instructions, the instructions are emitted
  - Repeat the above steps until the tree is reduced to a single node, or until no more templates match
- Output of the tree-translation scheme is the instruction sequence generated as the input tree is reduced to a single node

Example of Code Generation with Tree Rewriting

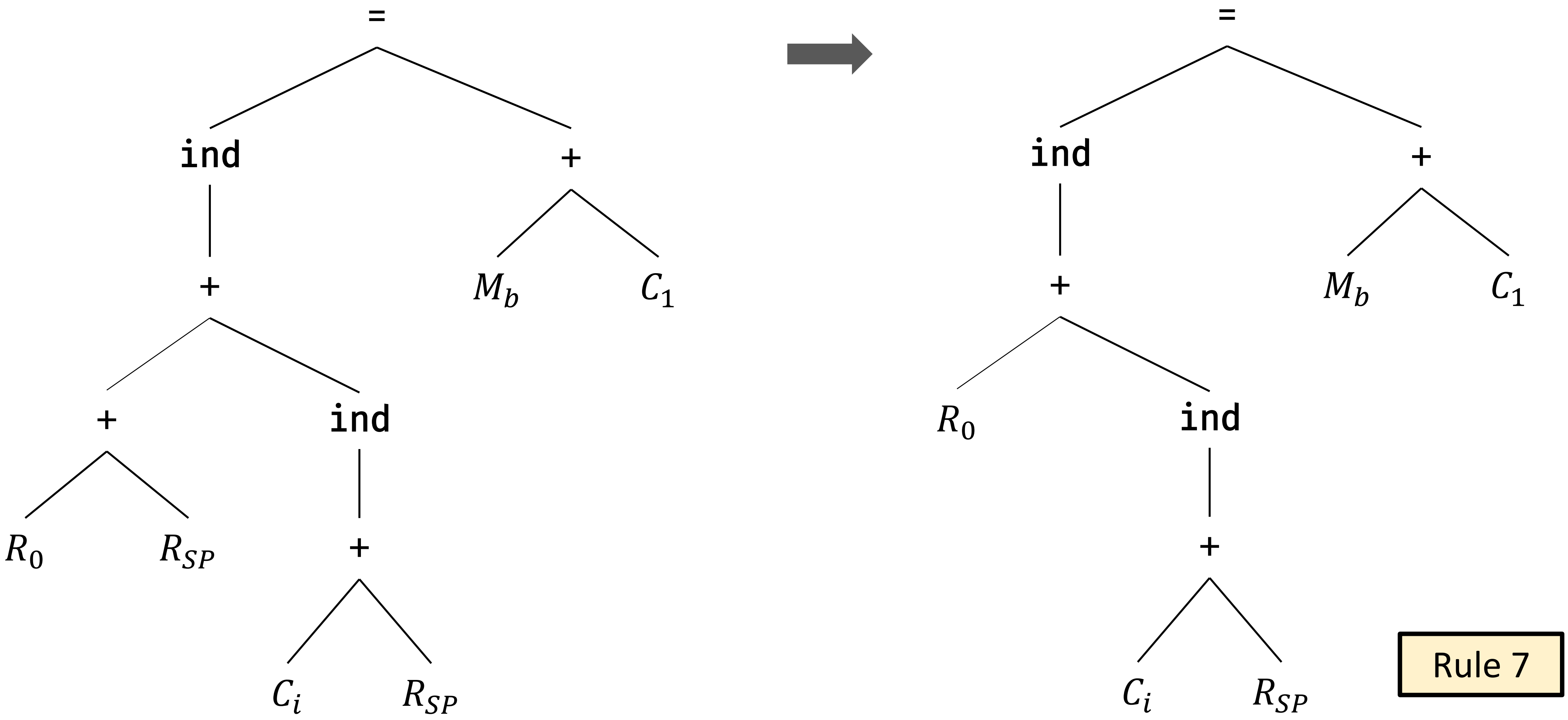


??

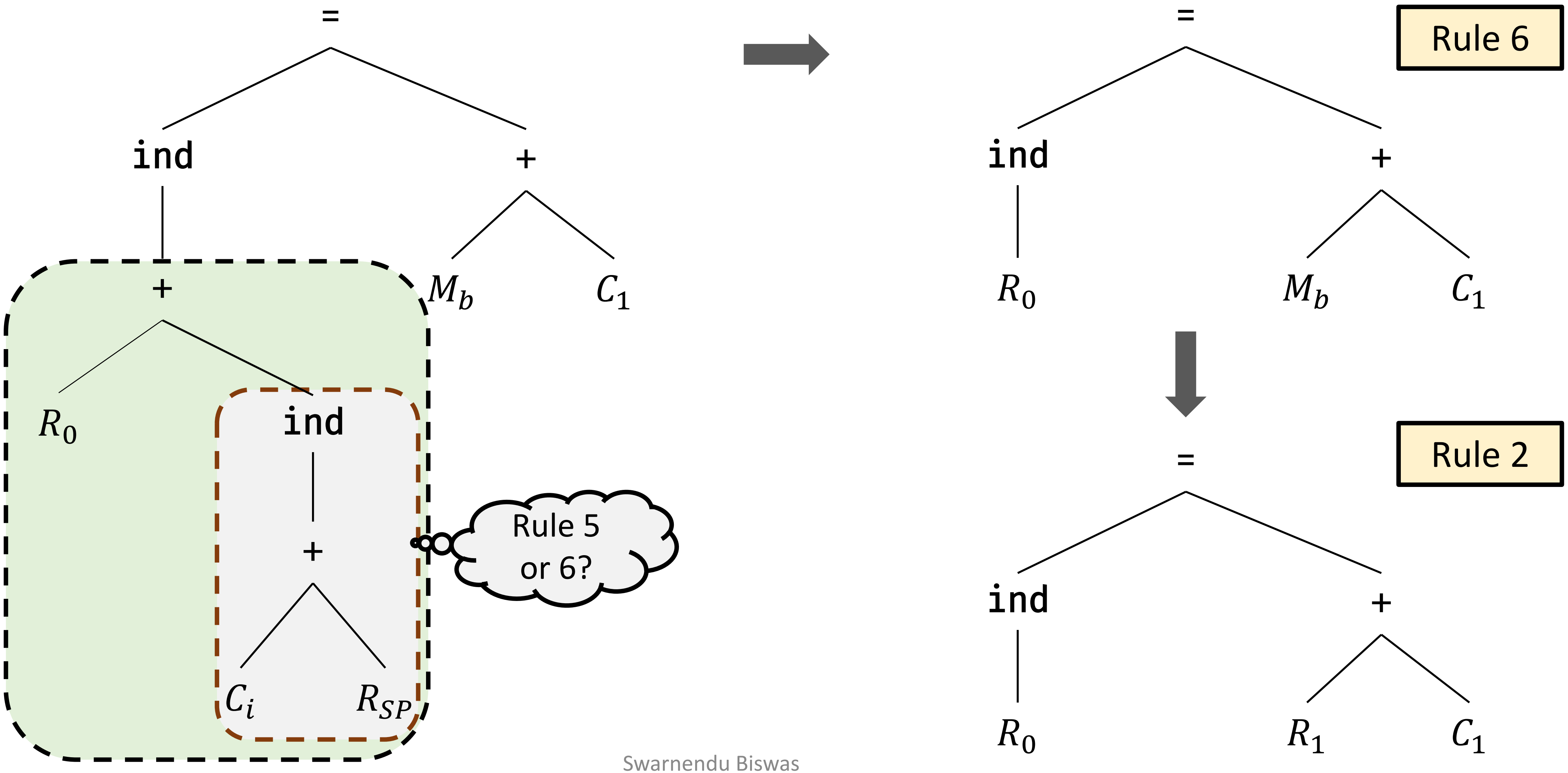
Example of Code Generation with Tree Rewriting



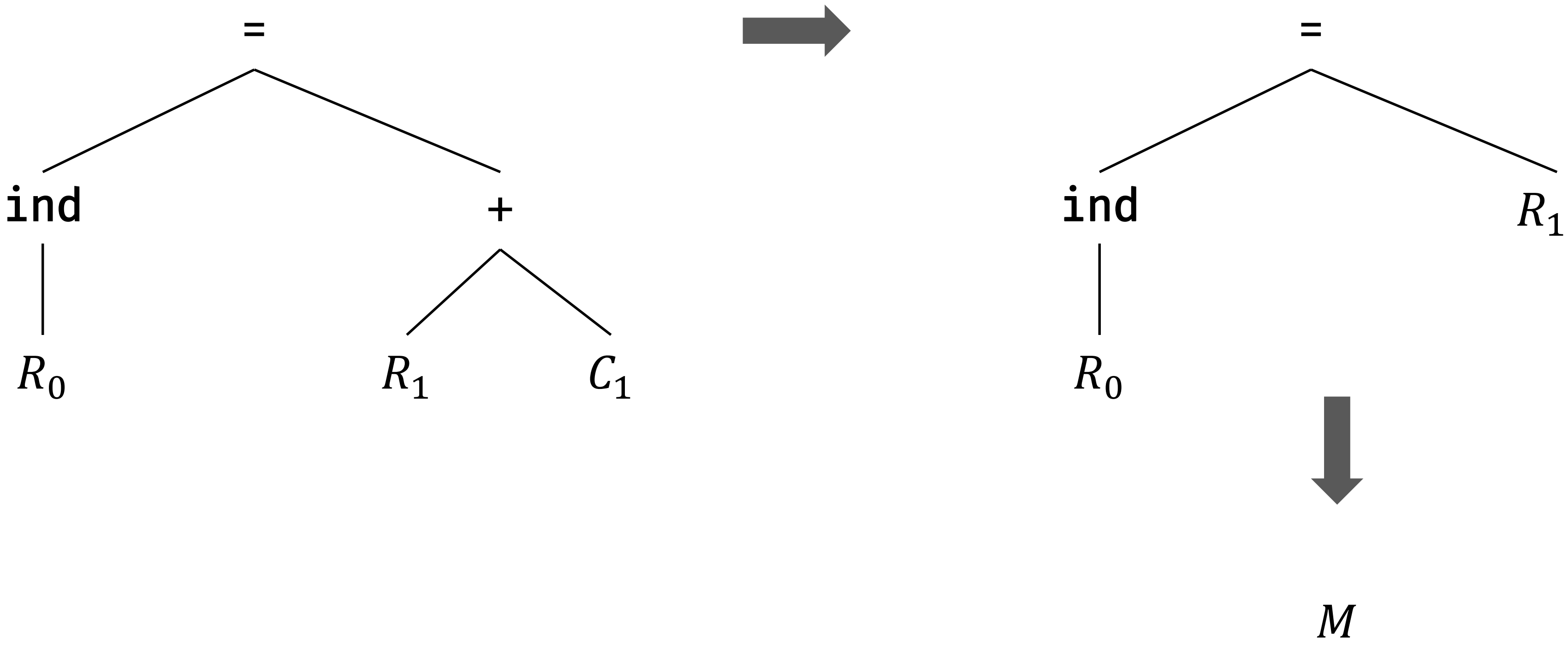
Example of Code Generation with Tree Rewriting



Example of Code Generation with Tree Rewriting



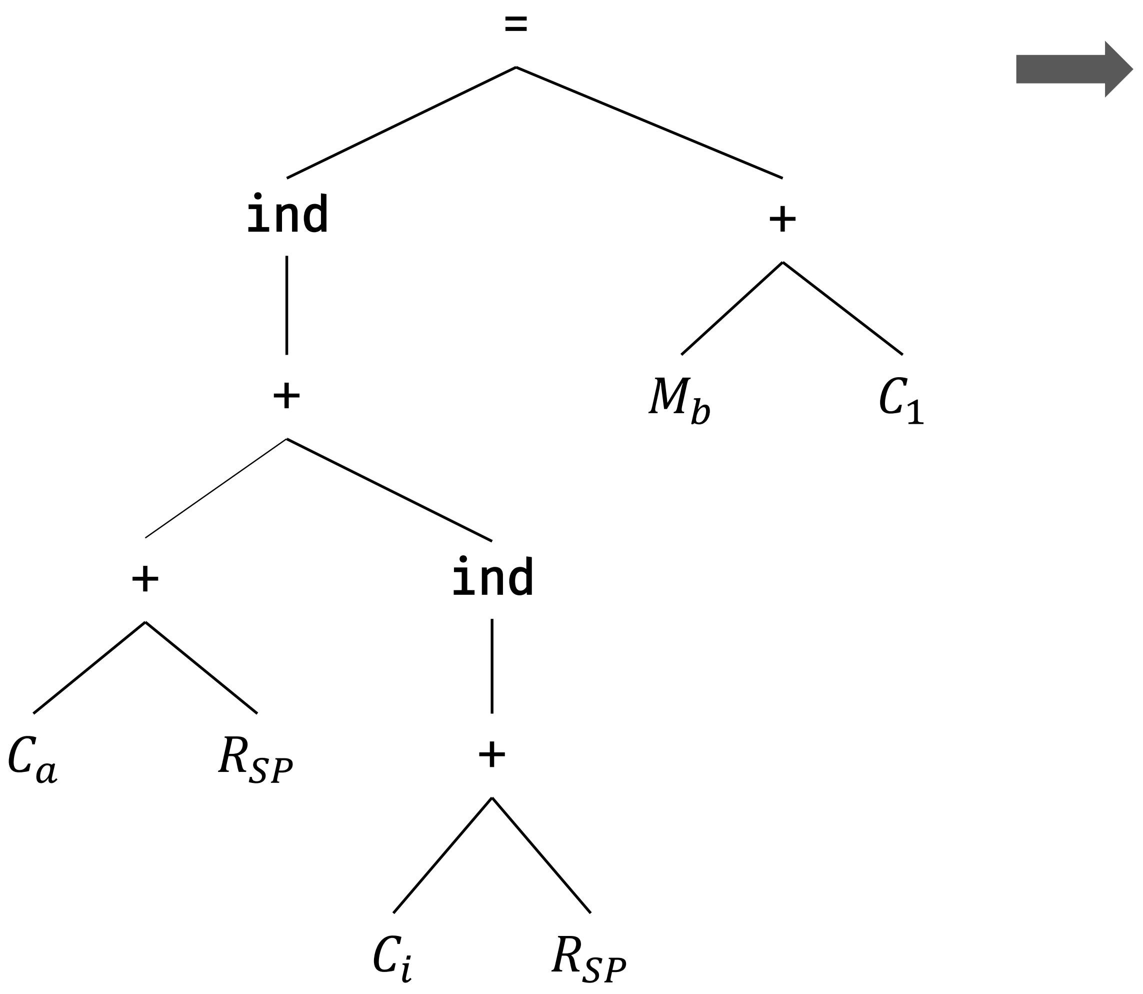
Example of Code Generation with Tree Rewriting



Rule 8

Rule 4

Example of Code Generation with Tree Rewriting



LD $R_0, \#a$
ADD R_0, R_0, SP
ADD $R_0, R_0, i(\mathrm{SP})$
LD R_1, b
INC R_1
ST $*R_0, R_1$

Considerations during Tree Reduction

i. Performance of tree matching impacts the efficiency of code generation at compile time

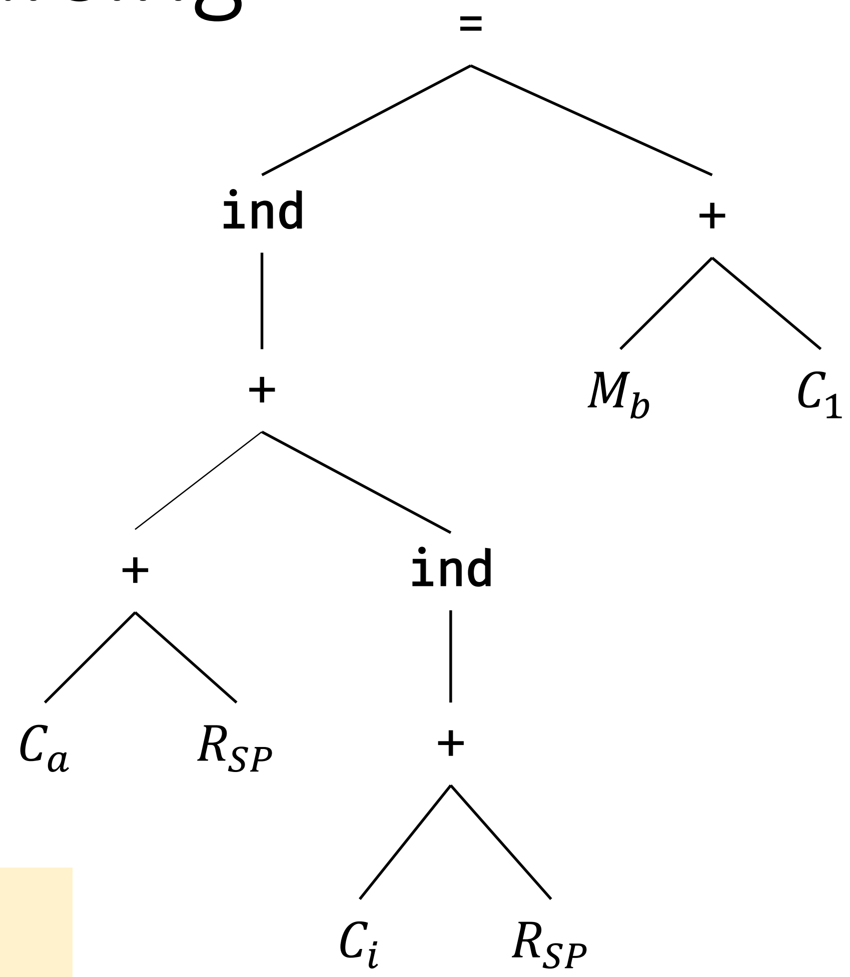
ii. Multiple templates may match during code generation

iii. Different match sequences of templates will lead to different code being generated which can impact efficiency

iv. If no template matches, then the code-generation process blocks
   - Assume each operator in the intermediate code can be implemented by one or more target-machine instructions
   - Assume there are sufficient registers to compute each tree node by itself

Pattern Matching with LR Parsing

- Idea
  - Convert the input tree to a string using prefix form for comparison
  - Use a parsing mechanism for pattern matching
  - Come up with a syntax-directed translation (SDT) as an alternate for tree rewriting rules



Prefix representation

$$= \ \mathbf{ind} \ + \ + C_a R_{SP} \ \mathbf{ind} + C_i R_{SP} + M_b C_1$$

SDT for Tree Rewriting

- Terminal **m** represents a memory location
- Terminal **sp** represents register SP
- Terminal **c** represents a constant

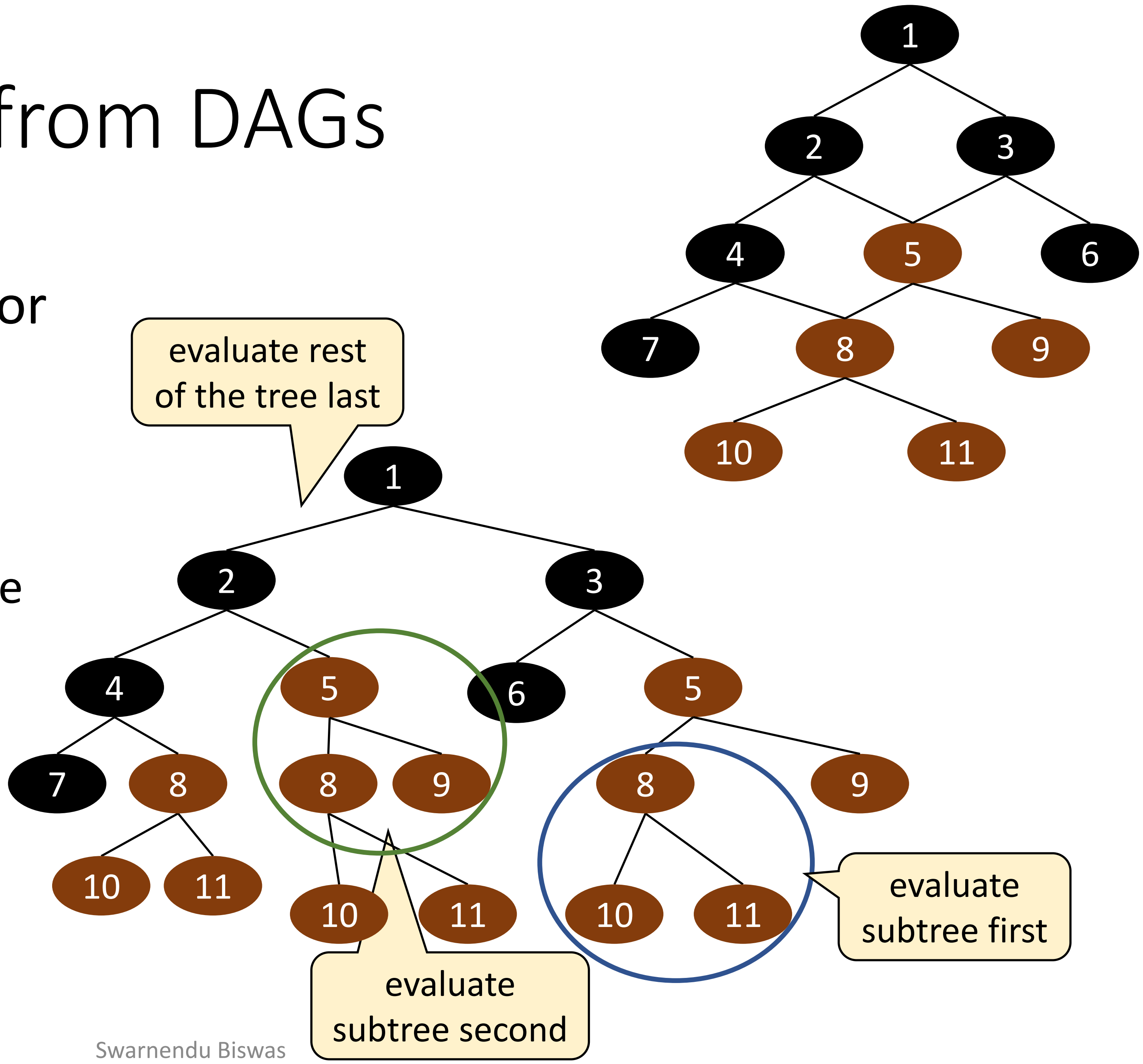
- Design a code generator for a different architecture by rewriting the grammar
- Resolve conflicts using estimates of instruction costs, favoring larger reductions, and favoring shifts over reductions

Production	**Semantic Action**
$R_i \rightarrow$ **c**$_a$	{ LD $R_i, \#a$ }
$R_i \rightarrow M_x$	{ LD R_i, x }
$M \rightarrow= M_x R_i$	{ ST x, R_i }
$M \rightarrow=$ **ind** $R_i R_j$	{ ST $*R_i, R_j$ }
$R_i \rightarrow$ **ind** $+$ **c**$_a R_j$	{ LD $R_i, a\left(R_j\right)$ }
$R_i \rightarrow +R_i$ **ind** $+$ **c**$_a R_j$	{ ADD $R_i, R_i, a\left(R_j\right)$ }
$R_i \rightarrow + R_i R_j$	{ ADD R_i, R_i, R_j }
$R_i \rightarrow + R_i$ **c**$_1$	{ INC R_i }
$R \rightarrow$ **sp**	
$M \rightarrow$ **m**	

Dynamic Programming Based Optimal Code Generation

Code Generation from DAGs

- Optimal code generation for DAGs is NP-complete
- So, DAGs are divided into trees and processed
  - An alternative is to replicate shared trees in DAGs but it increases the code size



evaluate rest of the tree last

evaluate subtree second

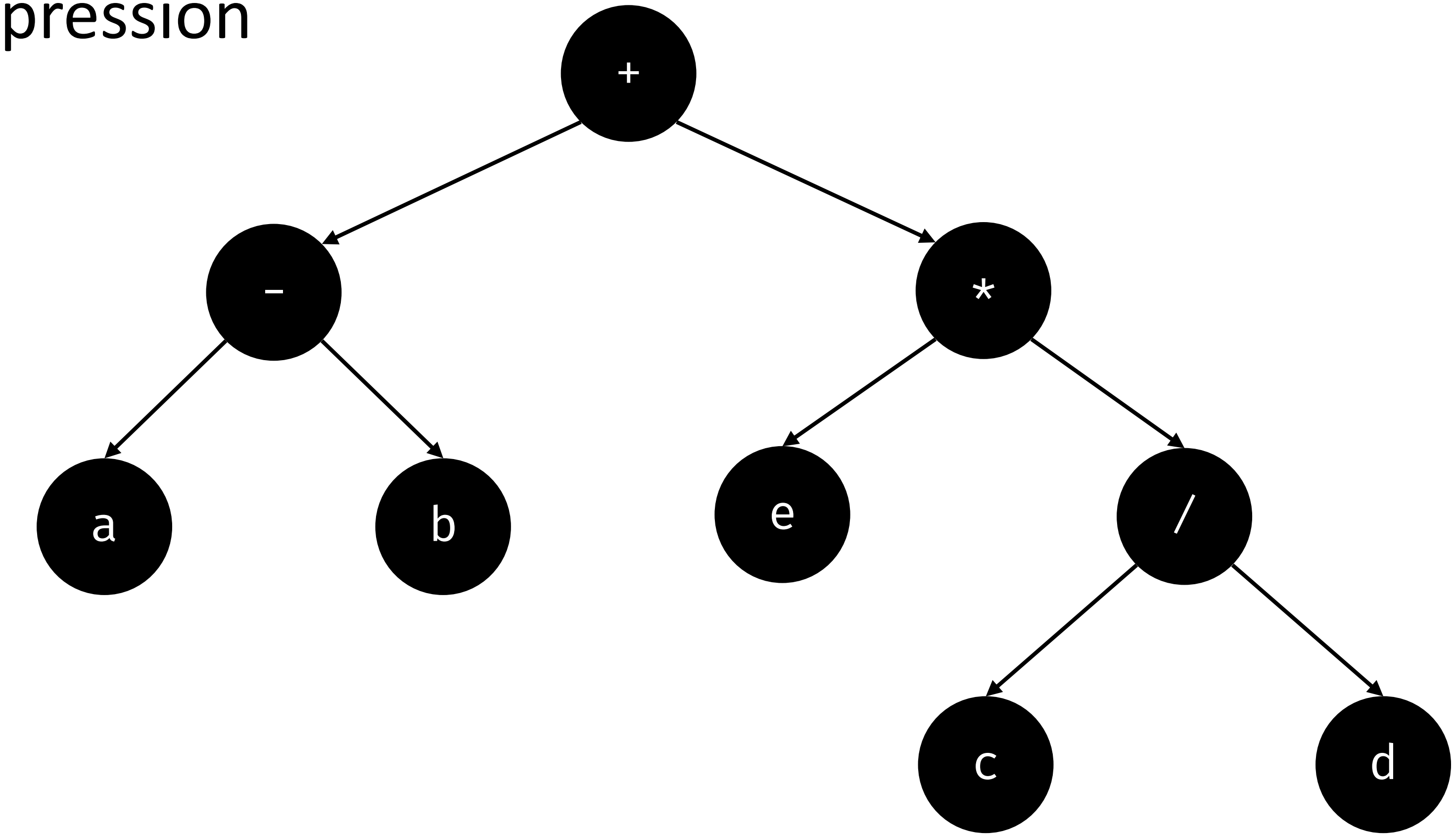
evaluate subtree first

Expression Trees

- A syntax tree for an expression

```
(a-b)+e*(c/d)
```

```
t1 = a − b
t2 = c / d
t3 = e * t2
t4 = t1 + t3
```



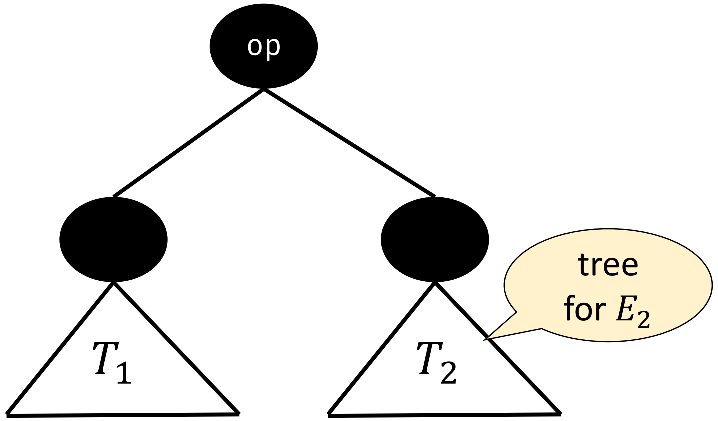
Dynamic Programming Based Optimal Code Generation

- Generates optimal code from an **expression tree** for a BB
  - Optimality is in terms of number of instructions generated
- Machine model – register machines with complex instruction sets
  - Assume there are r interchangeable registers R_0, …, R_{r-1}
  - Instructions are of form: $R_i = E$
    - If E involves registers, then R_i must be one of them (i.e., 2-address instructions)
    - Variants: $R_i = M_j$, $R_i = R_i \text{ op } R_j$, $R_i = R_i \text{ op } M_j$
    - Other variants: $R_i = R_j$, $M_i = R_j$

Contiguous Evaluation

- The optimality criterion requires contiguous evaluation of an expression tree
    - No higher costs and no more registers
- A program P evaluates a tree T **contiguously** if
    - it first evaluates those subtrees of T that need to be computed into memory,
    - it then evaluates T_1, $T2$, and then root, in order, or T_2, T_1, and then root, in order
- Evaluating part of T_1 leaving the result in a register, evaluating T_2, and then evaluating rest of T_1 is not contiguous evaluation

Assume E is $E_1 + E_2$



syntax tree for E

Dynamic Programming Algorithm

- **Assumption**: target has r registers

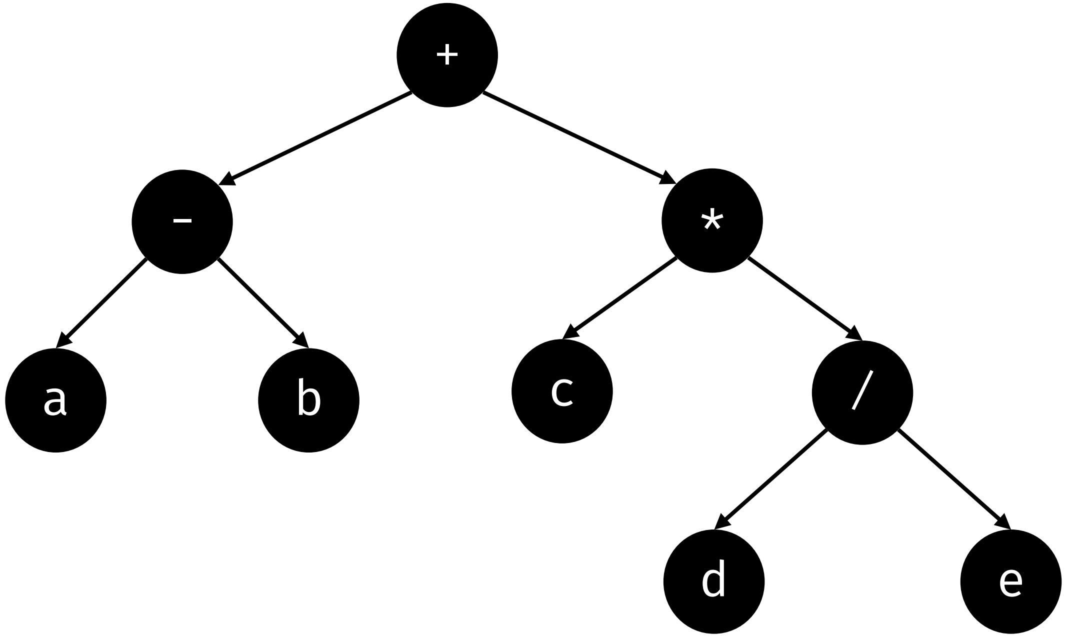
1. Compute bottom-up for each node n of the expression tree T an array C of costs
   - $C[i]$ is the minimum cost of computing the subtree S rooted at n into a register, assuming i registers are available for the computation, for $1 < i < r$
   - The cost of computing a node n includes the count of loads and stores necessary to evaluate S in the given number of registers plus the cost of computing the operator at the root of S
2. Traverse T, using the cost vectors to determine which subtrees of T must be computed into memory
3. Traverse each tree using the cost vectors and associated instructions to generate the final target code
   - Code for the subtrees computed into memory locations is generated first, then code for other subtrees, and then code for the root

Example

- Consider a machine having two registers R_0 and R_1
- Assume the available instructions are

LD R_i, M_j
op R_i, R_i, R_j
op R_i, R_i, M_j
LD R_i, R_j
ST M_i, R_j

- Furthermore, assume all instructions are of unit cost
  - Can be extended to cases where instructions have varying costs



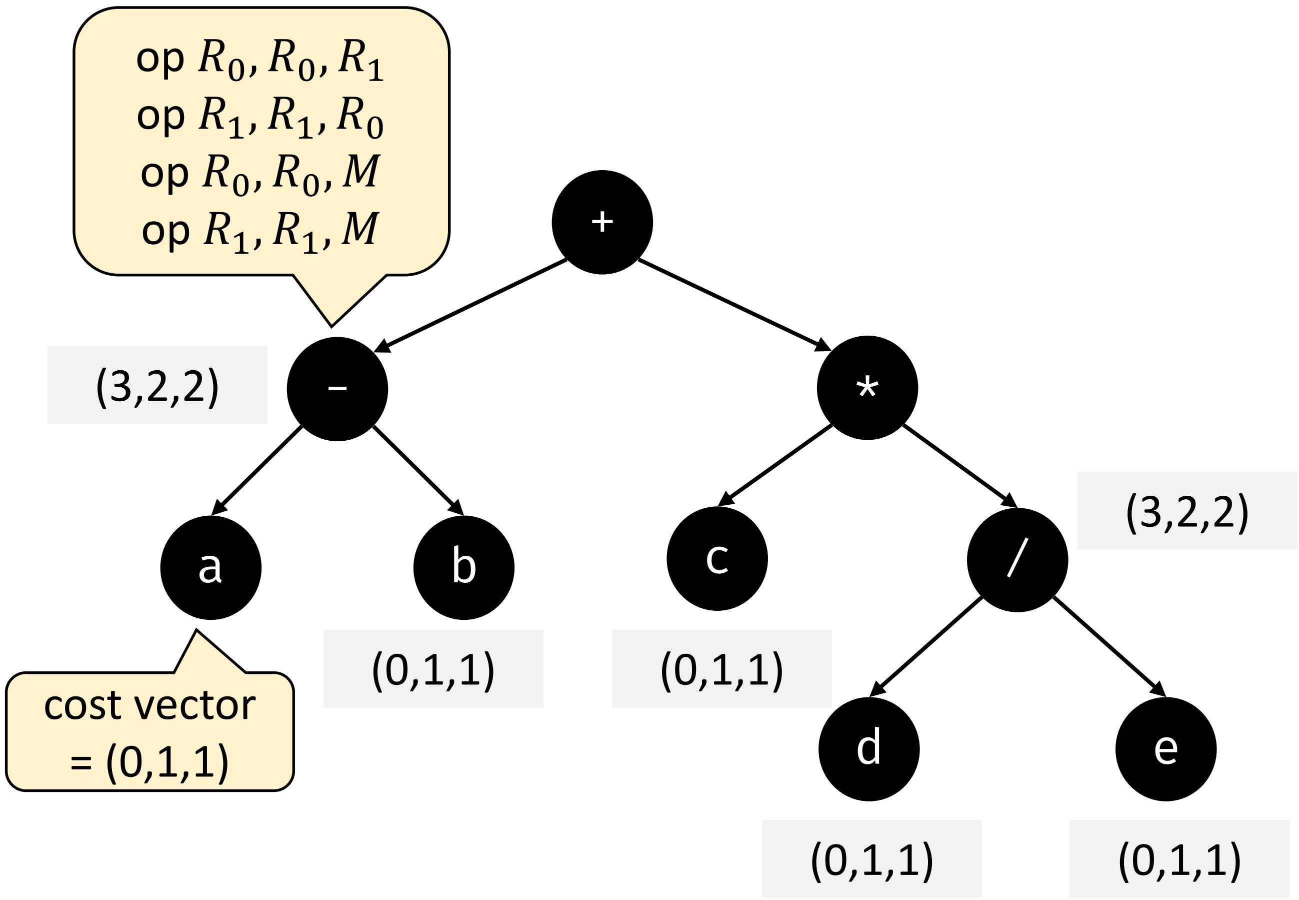
Expression tree

Expression Tree with Cost Vectors

$C_a[0] = 0$	Cost of computing a in memory
$C_a[1] = 1$	Cost of computing a in a register
$C_a[2] = 1$	Cost of computing a in a register, with two registers available

LD R_0, a
ADD R_0, R_0, b

- $C_{-}[1] = C_a[1] + C_b[0] + 1 = 1 + 0 + 1 = 2$
- $$C_{-}[2] = \min \begin{pmatrix} C_a[2] + C_b[1] + 1, \\ C_a[2] + C_b[0] + 1, \\ C_a[1] + C_b[2] + 1, \\ C_a[1] + C_b[1] + 1, \\ C_a[1] + C_b[0] + 1 \end{pmatrix}$$
  $$= \min(3,2,3,3,2) = 2$$
- $C_{-}[0] = C_{-}[2] + 1 = 3$



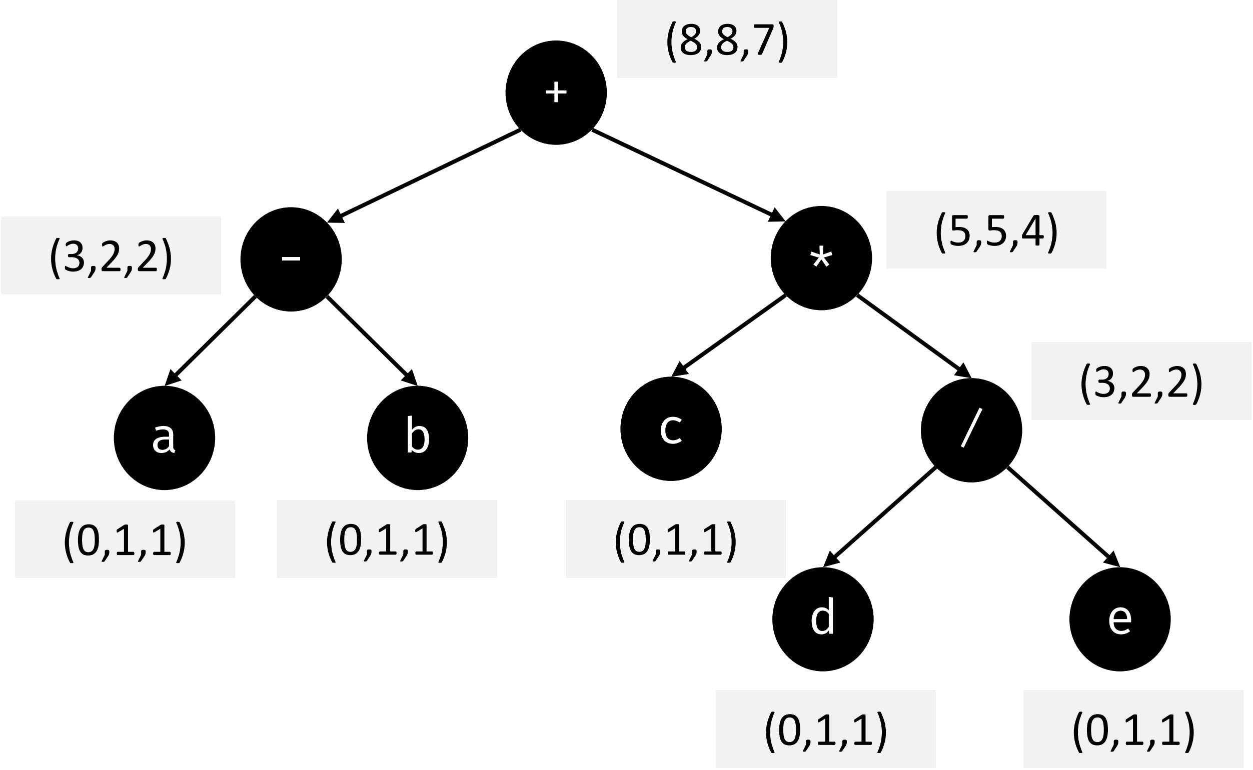
op R_0, R_0, R_1
op R_1, R_1, R_0
op R_0, R_0, M
op R_1, R_1, M

cost vector = (0,1,1)

Expression Tree with Cost Vectors

- $C_*[1] = C_c[1] + C_/[0] + 1 = 1 + 3 + 1 = 5$
- $C_*[2] = \min\begin{pmatrix} C_c[2] + C_/[1] + 1, \\ C_c[2] + C_/[0] + 1, \\ C_c[1] + C_/[2] + 1, \\ C_c[1] + C_/[1] + 1, \\ C_c[1] + C_/[0] + 1 \end{pmatrix}$
  $= \min(4,5,4,4,5) = 4$
- $C_*[0] = C_*[2] + 1 = 5$

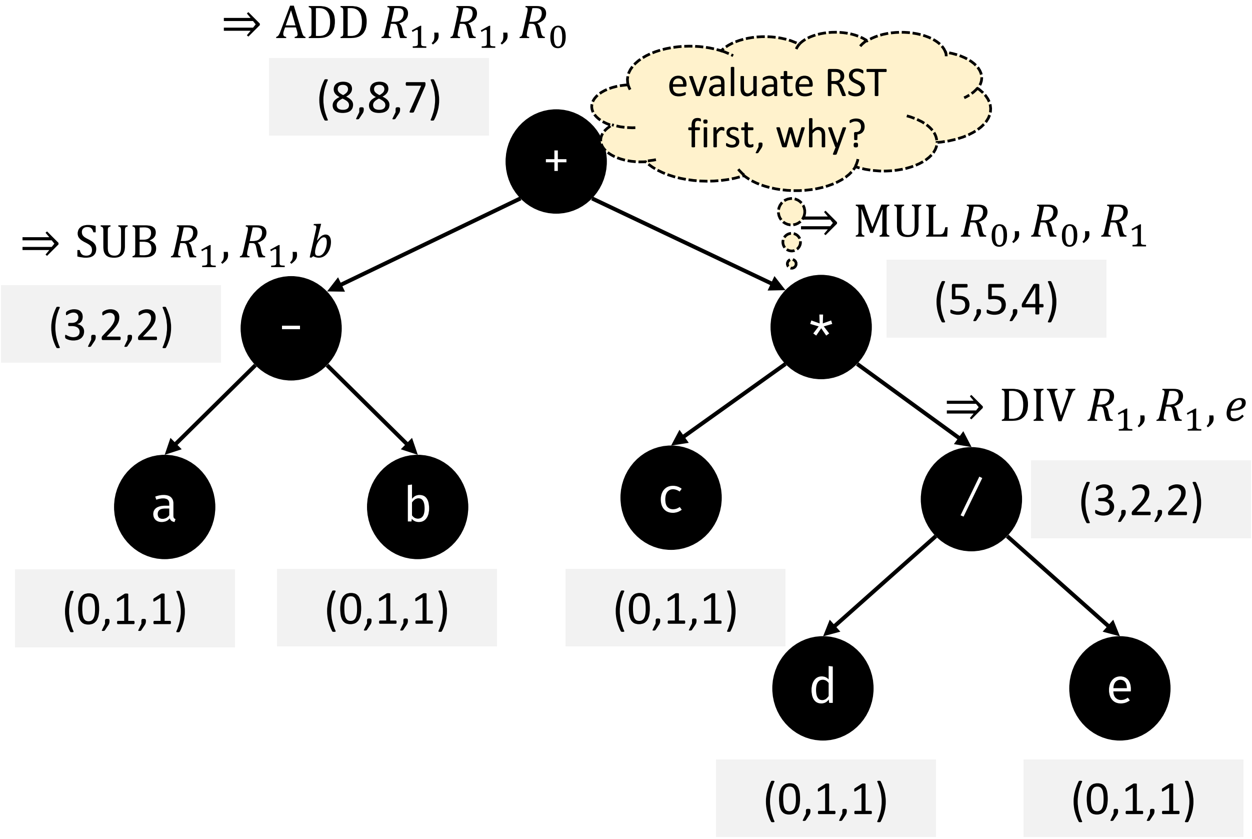
- $C_+[1] = C_-[1] + C_*[0] + 1 = 2 + 5 + 1 = 8$
- $C_+[2] = \min\begin{pmatrix} C_-[2] + C_*[1] + 1, \\ C_-[2] + C_*[0] + 1, \\ C_-[1] + C_*[2] + 1, \\ C_-[1] + C_*[1] + 1, \\ C_-[1] + C_*[0] + 1 \end{pmatrix}$
  $= \min(8,8,7,8,8) = 7$
- $C_+[0] = C_+[2] + 1 = 8$



Tree Traversal to Generate Code

LD R_0, c
LD R_1, d
DIV R_1, R_1, e
MUL R_0, R_0, R_1
LD R_1, a
SUB R_1, R_1, b
ADD R_1, R_1, R_0

- Min cost at node $+$ is 7, which implies right subtree (RST) is computed with 2 registers in R_0 and left subtree (LST) is computed with 1 register into R_1
- For node $*$, compute RST with one register in R_1 and LST in R_0
- For node c, emit LD R_0, c
- For node $/$, compute RST in memory and compute LST in R_1
- For node d, emit LD R_1, d
- For node $-$, compute RST in memory and compute LST in R_1
- For node a, emit LD R_1, a



Instruction Selection via Peephole Optimization

Peephole Optimization

- **Insight**: Find local optimizations by examining short sequences of adjacent operations
  - The sliding window, or the peephole, moves over code
  - Code in a peephole need not be contiguous
  - Goal is to identify code patterns that can be improved
  - Rewrite code patterns with improved sequence

1

LD a, R_0
ST R_0, a

→

LD a, R_0

2

ST a, R_0
LD R_3, a

→

ST a, R_0
MOV R_3, R_0

3

ADD $R_7, R_0, 0$
MUL R_{10}, R_4, R_7

→

MUL R_{10}, R_4, R_0

Examples of Peephole Optimizations

- Eliminate redundant instructions
- Eliminate unreachable code
- Eliminate jump over jumps
- Algebraic simplification
- Strength reduction
- Use of machine idioms

```
…
LD R0, x
{no modifications
to x or R0}
ST R0, x
…
```

BB

```
    …
    if print == 1
      goto L1
    goto L2
L1: print …
L2: …
```

→

```
    …
    if print != 1
      goto L2
    print …
L2: …
```

```
    …
    goto L1
    …
L1: goto L2
    …
```

→

```
    …
    goto L2
    …
L1: goto L2
    …
```

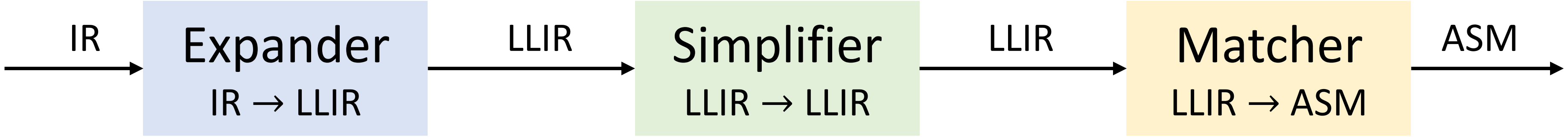
no jumps to L1 →

```
    …
    goto L2
    …
    …
    …
```

BB beginning at L1: … can be removed if it is preceded by an unconditional jump

Peephole Optimization based Code Generation

- A naïve optimization strategy can use exhaustive search to match the patterns and rewrite code
  - Can work if number of patterns and the window size are small
  - Does not work for modern complex ISAs
- Workflow in a modern peephole optimizer



- In an optimizer, the input and output languages are the same
- With a different output language (e.g., ASM), the optimizer can be used for code generation

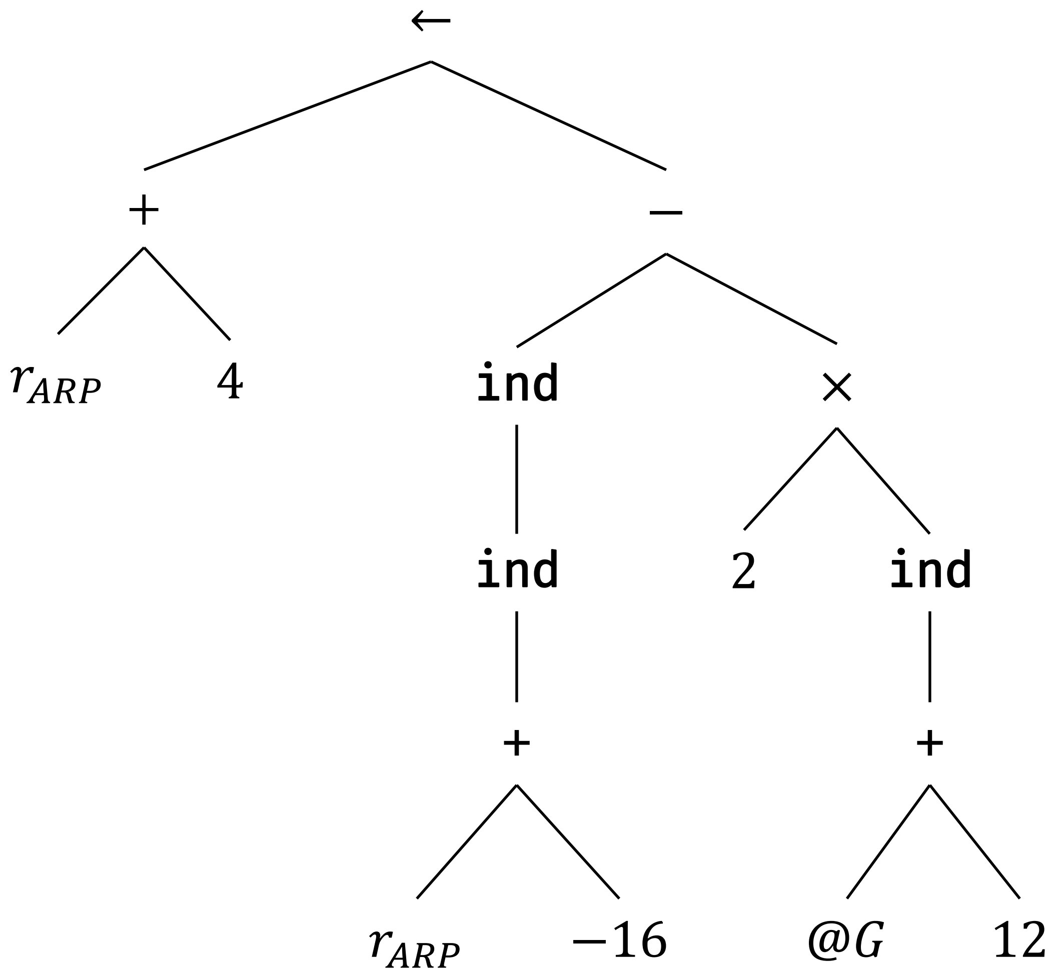
Peephole Optimization based Code Generation

- Expander rewrites the IR to represent all the direct effects of an operation
  - If $\mathrm{OP}\ R_0, R_1, R_2$ sets a condition code, then the LLIR should include an explicit operation to set the condition code
- Simplifier performs limited local optimization on the LLIR in the window
- Matcher compares the simplified LLIR against the pattern library

Example

AST computes $a = b - 2 \times c$

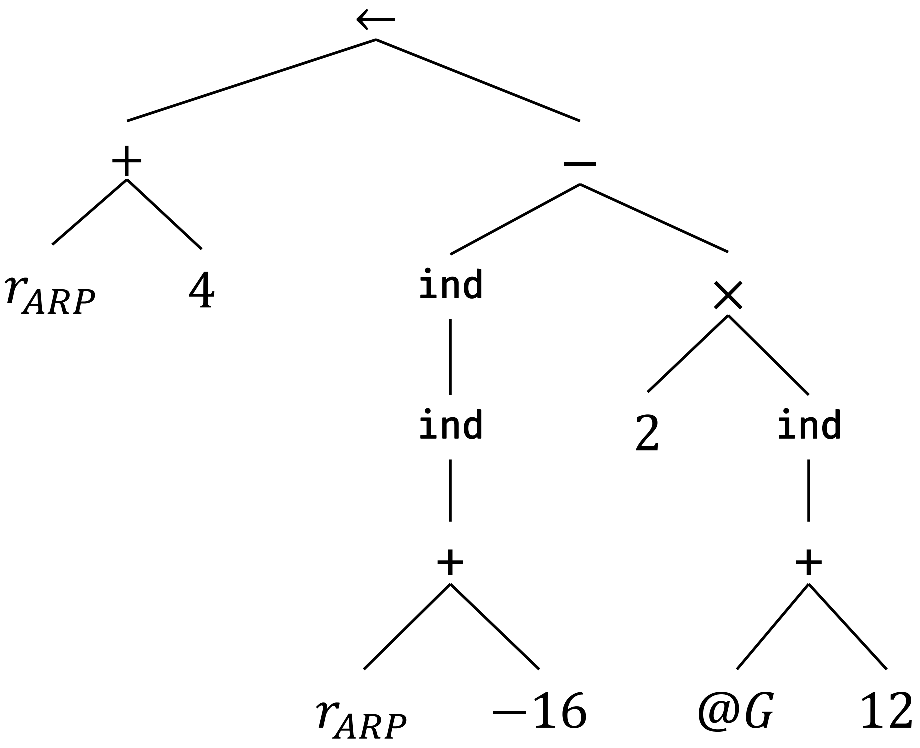
- a is stored at offset 4 in the local AR
- b stored as a call-by-reference parameter whose pointer is stored at offset – 16 from the ARP
- c is at offset 12 from the label $@G$



Op	Arg$_1$	Arg$_2$	Result
×	2	c	t_1
−	b	t_1	a

Example

Op	Arg$_1$	Arg$_2$	Result
$\times$	2	c	t_1
$-$	b	t_1	a

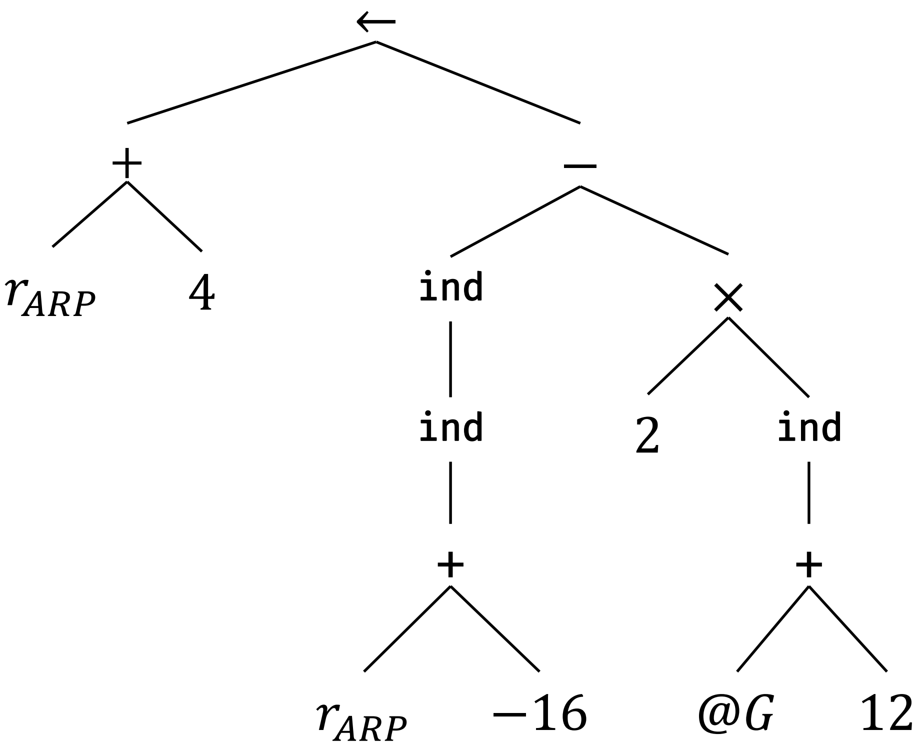


Expand

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$
$r_{22} \leftarrow r_{ARP} + r_{21}$
$M(r_{22}) \leftarrow r_{20}$

Example

Op	Arg$_1$	Arg$_2$	Result
×	2	c	t_1
−	b	t_1	a



Expand

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$
$r_{22} \leftarrow r_{ARP} + r_{21}$
$M(r_{22}) \leftarrow r_{20}$

Assume a sliding window of size 3

Simplify

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{14} \leftarrow M(r_{11} + 12)$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{18} \leftarrow M(r_{ARP} - 16)$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$M(r_{ARP} + 4) \leftarrow r_{20}$

fewer instructions and registers

Sequences Produced by the Simplifier

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$
$r_{22} \leftarrow r_{ARP} + r_{21}$
$M(r_{22}) \leftarrow r_{20}$

Sequence 1

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$

Sequence 2

$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$

Sequence 3

$r_{11} \leftarrow @G$
$r_{13} \leftarrow r_{11} + 12$
$r_{14} \leftarrow M(r_{13})$

Sequence 4

$r_{11} \leftarrow @G$
$r_{14} \leftarrow M(r_{11} + 12)$
$r_{15} \leftarrow r_{10} \times r_{14}$

Sequence 5

$r_{14} \leftarrow M(r_{11} + 12)$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$

Sequence 6

$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$

Sequence 7

$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{17} \leftarrow r_{ARP} - 16$
$r_{18} \leftarrow M(r_{17})$

Sequence 8

$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{18} \leftarrow M(r_{ARP} - 16)$
$r_{19} \leftarrow M(r_{18})$

Sequence 9

$r_{18} \leftarrow M(r_{ARP} - 16)$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$

Sequences Produced by the Simplifier

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$
$r_{22} \leftarrow r_{ARP} + r_{21}$
$M(r_{22}) \leftarrow r_{20}$

Sequence 10

$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$

Sequence 11

$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$
$r_{22} \leftarrow r_{ARP} + r_{21}$

Sequence 12

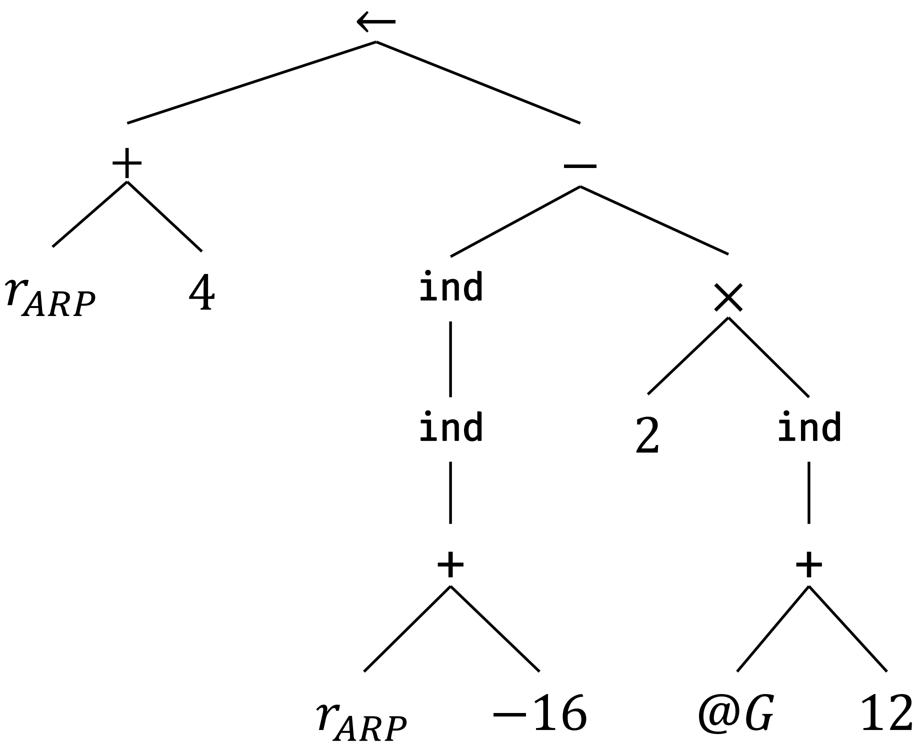
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{22} \leftarrow r_{ARP} + 4$
$M(r_{22}) \leftarrow r_{20}$

Sequence 13

$r_{20} \leftarrow r_{19} - r_{15}$
$M(r_{ARP} + 4) \leftarrow r_{20}$

Example

Op	Arg$_1$	Arg$_2$	Result
$\times$	2	c	t_1
$-$	b	t_1	a



Expand

$$\begin{aligned}
&r_{10} \leftarrow 2\\
&r_{11} \leftarrow @G\\
&r_{12} \leftarrow 12\\
&r_{13} \leftarrow r_{11} + r_{12}\\
&r_{14} \leftarrow M(r_{13})\\
&r_{15} \leftarrow r_{10} \times r_{14}\\
&r_{16} \leftarrow -16\\
&r_{17} \leftarrow r_{ARP} + r_{16}\\
&r_{18} \leftarrow M(r_{17})\\
&r_{19} \leftarrow M(r_{18})\\
&r_{20} \leftarrow r_{19} - r_{15}\\
&r_{21} \leftarrow 4\\
&r_{22} \leftarrow r_{ARP} + r_{21}\\
&M(r_{22}) \leftarrow r_{20}
\end{aligned}$$

Simplify

$$\begin{aligned}
&r_{10} \leftarrow 2\\
&r_{11} \leftarrow @G\\
&r_{14} \leftarrow M(r_{11} + 12)\\
&r_{15} \leftarrow r_{10} \times r_{14}\\
&r_{18} \leftarrow M(r_{ARP} - 16)\\
&r_{19} \leftarrow M(r_{18})\\
&r_{20} \leftarrow r_{19} - r_{15}\\
&M(r_{ARP} + 4) \leftarrow r_{20}
\end{aligned}$$

Match

$$\begin{aligned}
&\mathrm{LD} && r_{10}, 2\\
&\mathrm{LD} && r_{11}, @G\\
&\mathrm{LD} && r_{14}, 12(r_{11})\\
&\mathrm{MUL} && r_{15}, r_{10}, r_{14}\\
&\mathrm{LD} && r_{18}, -16(r_{ARP})\\
&\mathrm{LD} && r_{19}, r_{18}\\
&\mathrm{SUB} && r_{20}, r_{19}, r_{15}\\
&\mathrm{ST} && 4(r_{ARP}), r_{20}
\end{aligned}$$

Example

Op	Arg$_1$	Arg$_2$	Result
$\times$	2	c	t_1
$-$	b	t_1	a

Expand

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$r_{21} \leftarrow 4$
$r_{22} \leftarrow r_{ARP} + r_{21}$
$M(r_{22}) \leftarrow r_{20}$

Simplify

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{14} \leftarrow M(r_{11} + 12)$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{18} \leftarrow M(r_{ARP} - 16)$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$M(r_{ARP} + 4) \leftarrow r_{20}$

Match

$\text{LD} \quad r_{10}, 2$
$\text{LD} \quad r_{11}, @G$
$\text{LD} \quad r_{14}, 12(r_{11})$
$\boxed{\text{MUL} \quad r_{15}, \boldsymbol{r_{10}}, r_{14}}$
$\text{LD} \quad r_{18}, -16(r_{ARP})$
$\text{LD} \quad r_{19}, r_{18}$
$\text{SUB} \quad r_{20}, r_{19}, r_{15}$
$\text{ST} \quad 4(r_{ARP}), r_{20}$

- Correctly identifying dead values, presence of control flow, and window size limit the effectiveness of peephole optimizations
- Can use logical instead based on data flow instead of physical windows

Current State in Code Generation

- Modern peephole systems automatically generates a matcher from a description of a target machine's instruction set
- Eases the work in retargeting the backend
  i. Provide a new appropriate machine description to the pattern generator to produce a new instruction selector
  ii. Change the LLIR sequences to match the new ISA
  iii. Modify the instruction scheduler and register allocator to reflect the characteristics of the new ISA
- GCC uses a low-level IR Register-Transfer Language (RTL) for optimization and for code generation
  - The backend uses a peephole scheme to convert RTL into assembly code

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 1st edition, Chapter 9.1-9.8, 9.10, 9.11.
- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 8.1-8.6, 8.9, 8.10.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 11.