CS 335: Code Generation

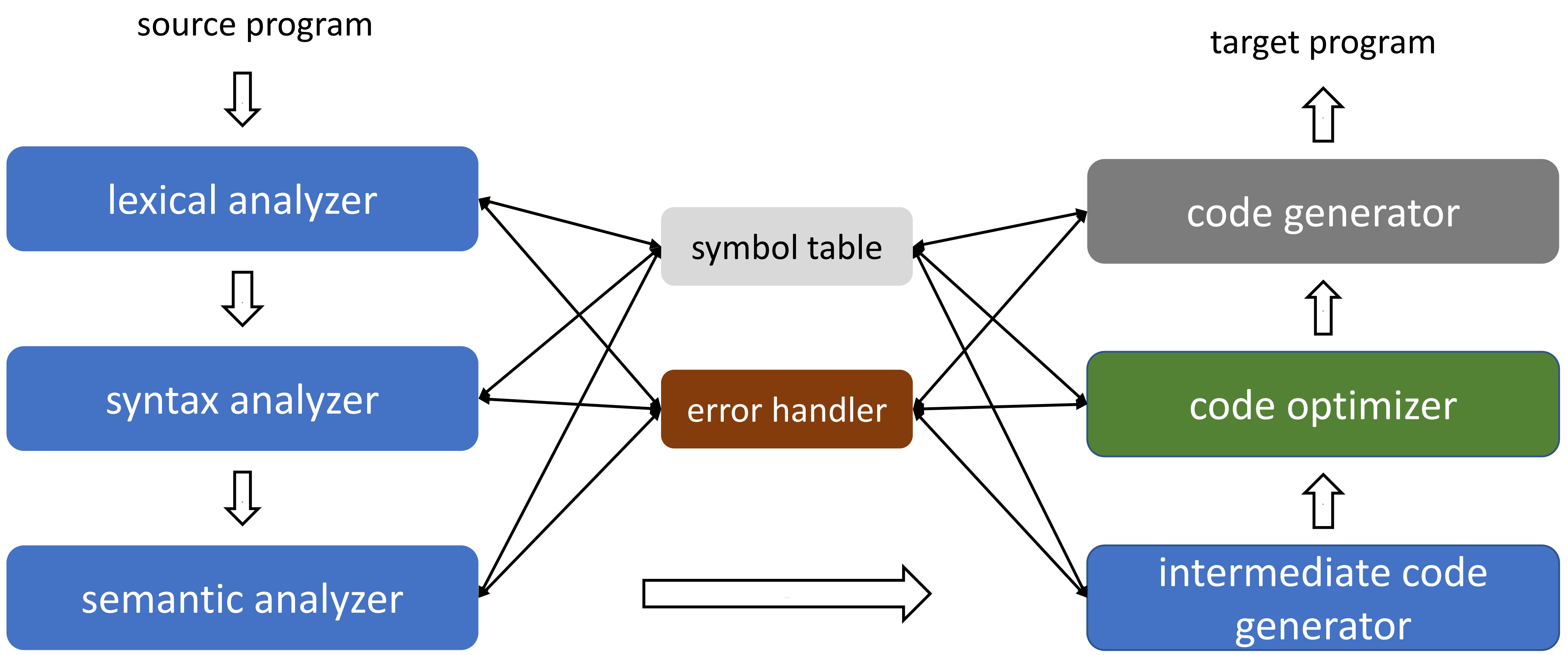
Swarnendu Biswas

Semester 2019-2020-II

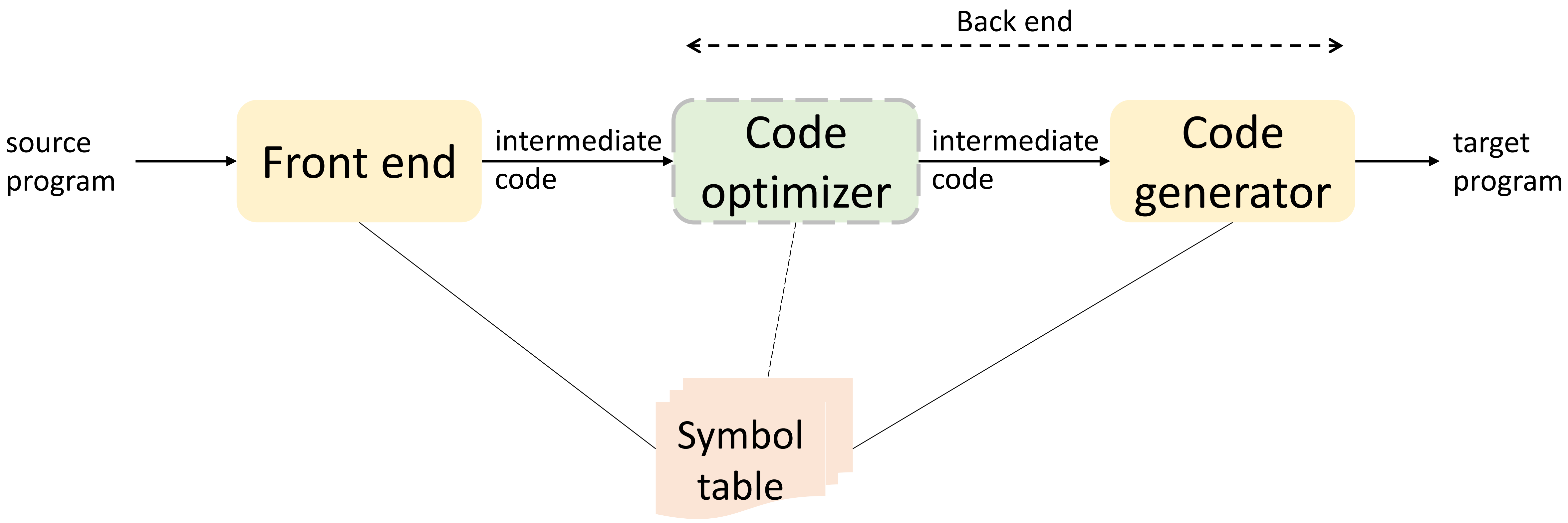
CSE, IIT Kanpur

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

An Overview of Compilation



Code Generation



Code Generation

- Generated output code must be correct
- Generated code must be of “good” quality
    - Notion of good can be vary
    - Should make efficient use of resources on the target machine
- 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 names in the IR can be operated 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
  - Register allocation
    - Decide what values to keep in which registers
  - Instruction scheduling
    - Decide in what order to schedule the execution of instructions
- Memory management

Instruction Selection

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

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

	`INC a`

- Target ISA influences instruction selection

Instruction Selection

- Target features
    - 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

- Other factors are IR level, speed of instructions, energy consumption and space overhead

Possible Idea for Instruction Selection

- Devise a target code skeleton for every 3AC IR instruction
- Replace every 3AC instruction with the skeleton

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

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

Instruction Selection

- Possible idea
  - Devise a target code skeleton for every 3AC IR instruction
  - Replace every 3AC instruction with the skeleton

`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`

Instruction Selection

- Possible idea
  - Devise a target code skeleton for every 3AC IR instruction
  - Replace every 3AC instruction with the skeleton

`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`
	$\boxed{\texttt{ST a, R0} \quad \texttt{LD R0, a}}$
	`ADD R0, R0, e`
	`ST d, R0`

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

- Finding an optimal assignment of registers to variables is NP-complete

Register Allocation

- Architectures may impose restrictions on usage of registers
- For example, 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 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

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

$x =* p$	LD $R1, p$
	$LD\ R2, 0(R1)$
	ST $x, R2$
		// 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 of the callee stores the return address of the caller

Static Allocation	
call callee	ST $callee.staticArea, \#here + 20$ BR $callee.codeArea$
return callee	BR $*callee.staticArea$

Determine Addresses in Target Code

- Need to generate code to manage activation records at runtime

3AC

```
// code for c
action_1
call p
action_2
halt
```

```
// code for p
action_3
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

Static Allocation

		// code for c
100:	ACTION_1	
120:	ST 364, #140	// save return address 140
132:	BR 200	// call p
140:	ACTION_2	
160:	HALT	
		// code for p
200:	ACTION_3	
220:	BR *364	// return to address saved in location 364

		// 300-363 hold activation record for c
300:		// return address
304:		// local data for c
		// 364-451 hold activation record for p
364:		// 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 pointer
BR $callee.codeArea$	// return 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 $\texttt{action}_1$ call q $\texttt{action}_2$ halt
// code for p $\texttt{action}_3$ return
// code for q $\texttt{action}_4$ call p $\texttt{action}_5$ call q $\texttt{action}_6$ call q return

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 *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

		// code for q
300:	ACTION$_4$	// conditional jump to 456
320:	ADD SP, SP, #qsize	
328:	ST *SP, #344	// push return address
336:	BR 200	// call p
344:	SUB SP, SP, #qsize	
352:	ACTION$_5$	
372:	ADD SP, SP, #qsize	
380:	BR *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 *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
  - No jumps into the middle of the block
  - No branch instructions other than the end

```
t1 = a * a
t2 = a * b
t3 = 2 * t2
t4 = t1 + t3
t5 = b * b
t6 = t4 + t5
```

- Consider the 3AC instruction I: $x = y + z$
  - I defines x and uses y and z

- A name in a basic block (BB) is live at a given point if its value is used after that point

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

```
begin
  prod = 0
  i = 1
  do begin
    prod = prod + a[i] * b[i]
  end
  while i <= 20
end
```

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)

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
```

```
1) i = 1
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)
```

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
```

```
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)
```

Leader lines (bold on the slide): **1)**, **2)**, **3)**, **10)**, **12)**, **13)**

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
    - Structure-preserving and algebraic transformations
    - Should not change the set of expressions computed by a block

Structure-Preserving Transformations

```
a = b + c
b = a − d
c = b + c
d = a − d
```

```
a = b + c
b = a − d
c = b + c
d = b
```

- Common subexpression elimination
- Dead code elimination
  - Remove statements that define variables that are dead
- 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
- Interchange of statements
  - Normal-form blocks permits statement interchanges without affecting the value of the block

$t_1 = b + c$
$t_2 = x + y$

Computing Next-Use Information

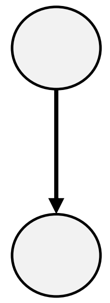
- Knowing when the value of a variable will be used next is import for generating good code
- Suppose a statement I defines x
- If a statement J uses x as an operand, and control can flow from I to J along a path where x is not redefined, then J uses the value of x defined at I
    - x is live at statement I

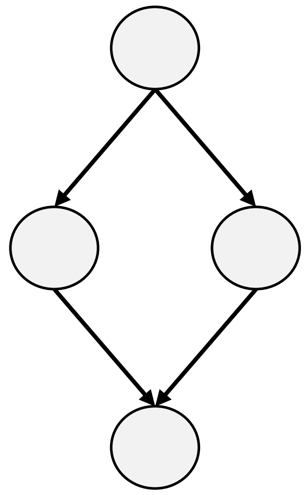
Determining Liveness and Next Use Information

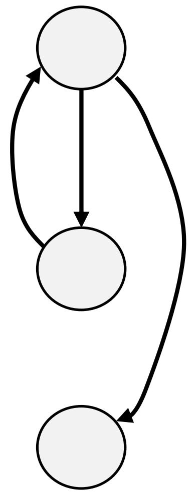
- **Input**
  - A BB (say B) of 3AC
  - Assume symbol table shows all non-temporary variables in B as live on exit
- **Output**
  - Liveness and next use information for each statement $I: x = y + z$ in B
- **Algorithm**
  - Start at the last statement in B. For each statement $I: x = y + z$ in B
    1. Attach to I the next use and liveness information for x, y, and z in the symbol table
    2. Set x to "not live" and "no next use" in the symbol table
    3. Set y and z to live and the next uses of y and z to I

Control Flow Graph (CFG)

- Graphical representation of control flow during execution
  - An edge represents possible transfer of control between BBs
- Each node represents a statement or a BB
- Dummy entry and exit nodes are often added
- Used for compiler optimizations and static analysis
  - Instruction scheduling, global register allocation

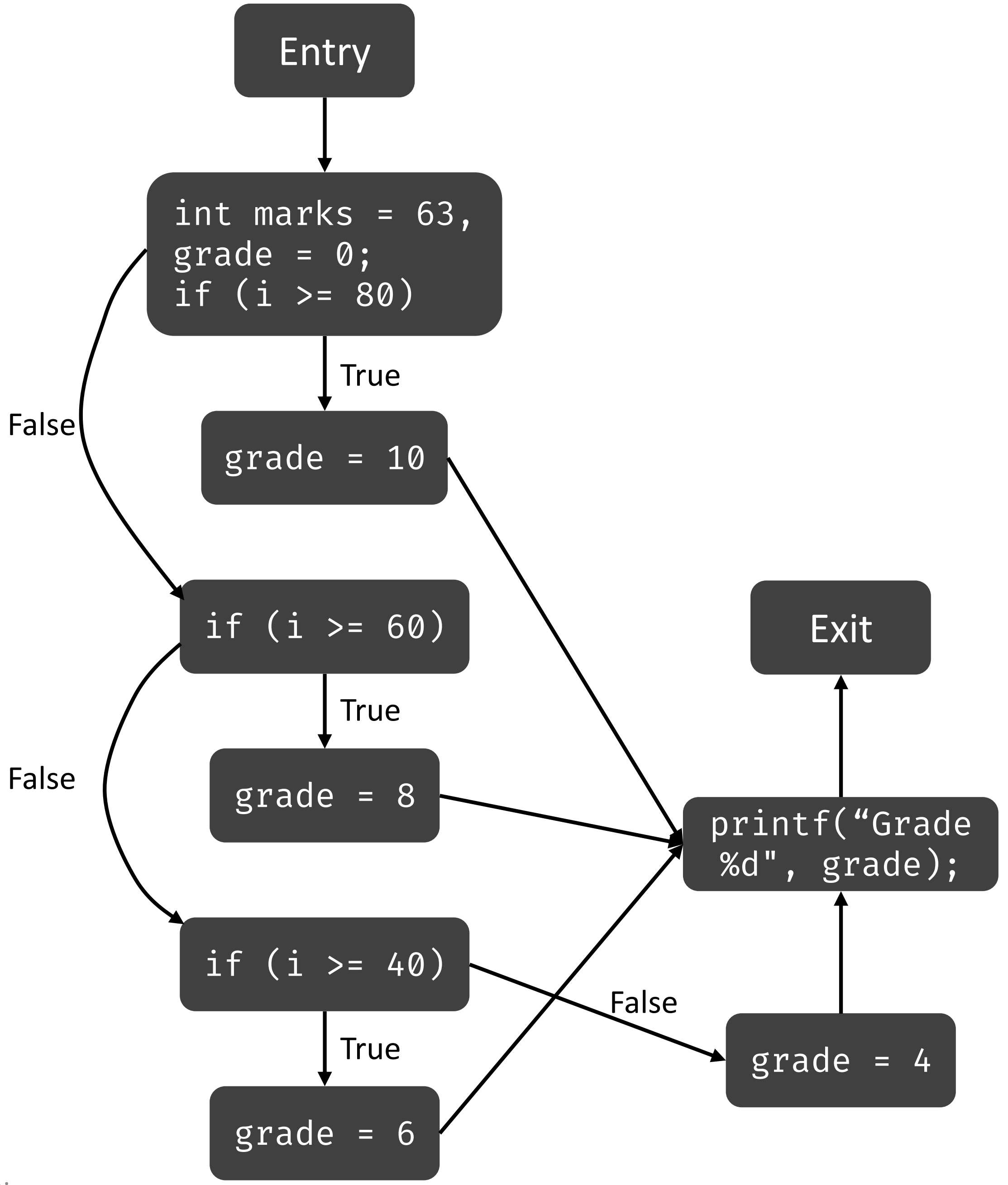

straight-line code


predicate


loop iteration

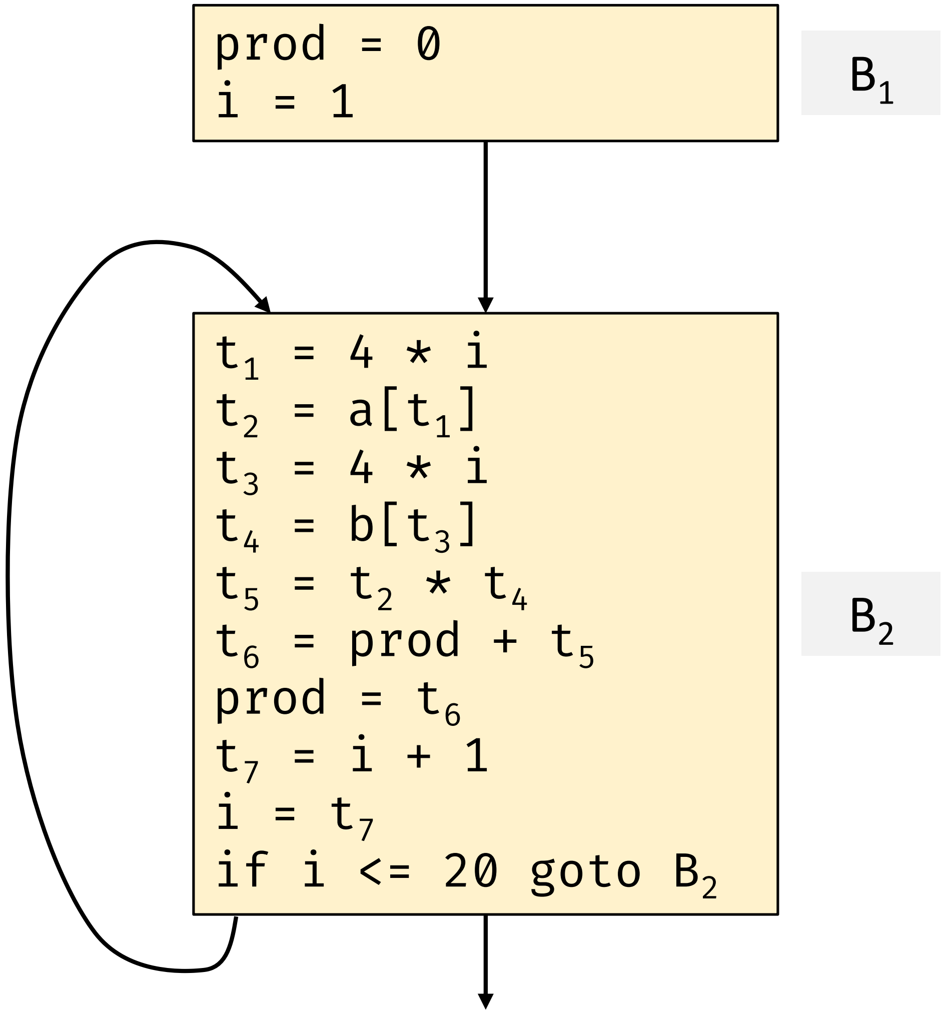
Example of a CFG

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



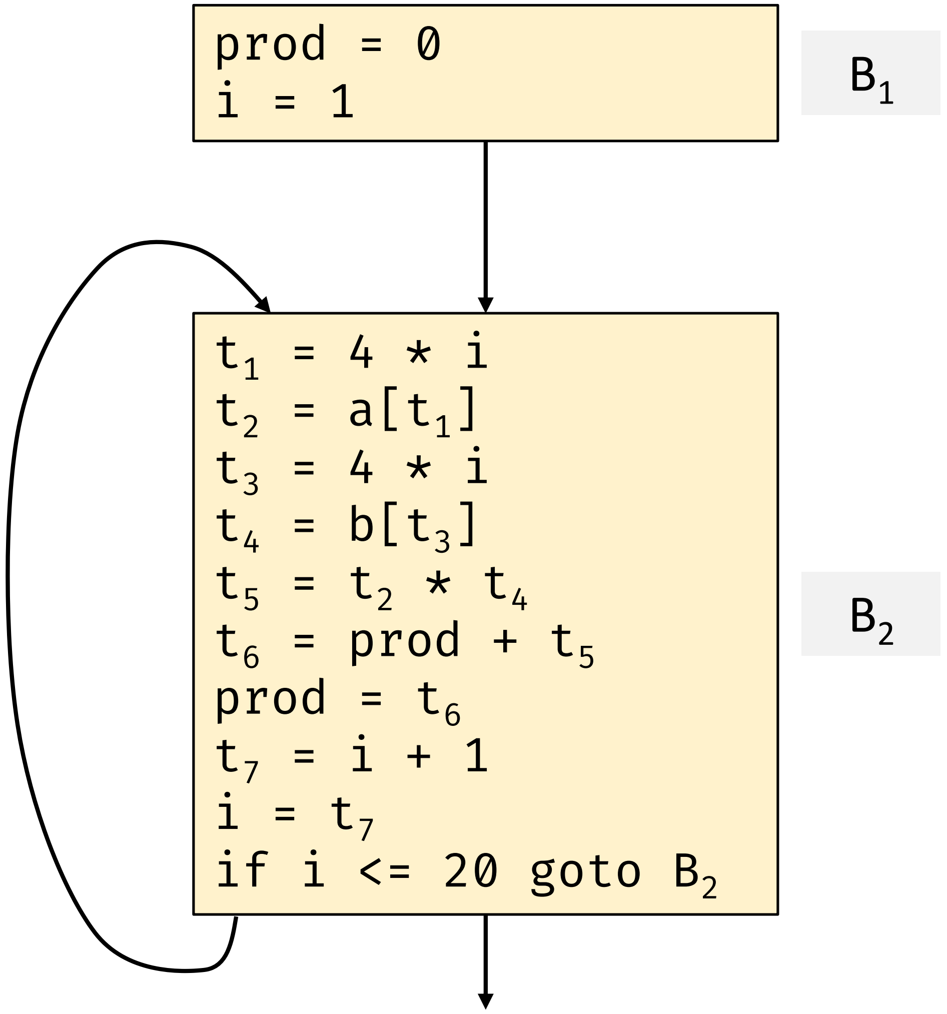
Control Flow Graph (CFG)

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)`



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

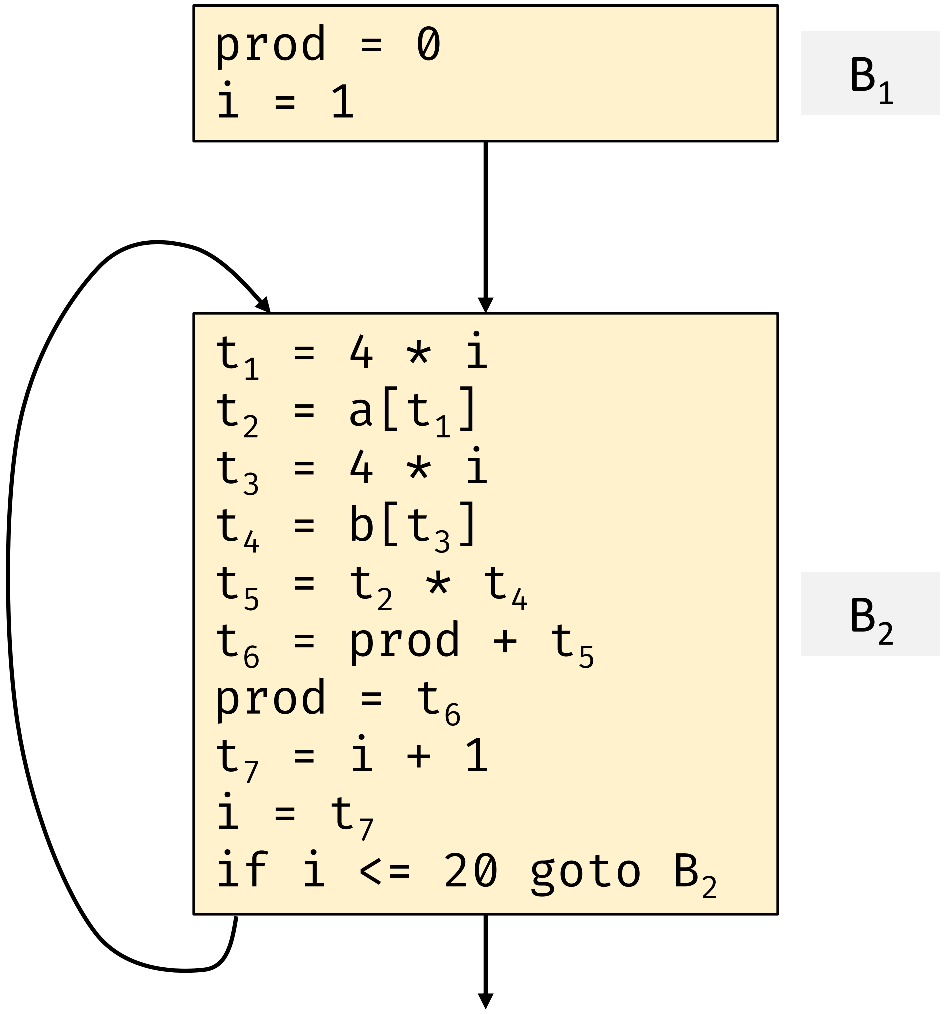


```
prod = 0
i = 1
```

```
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
```

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

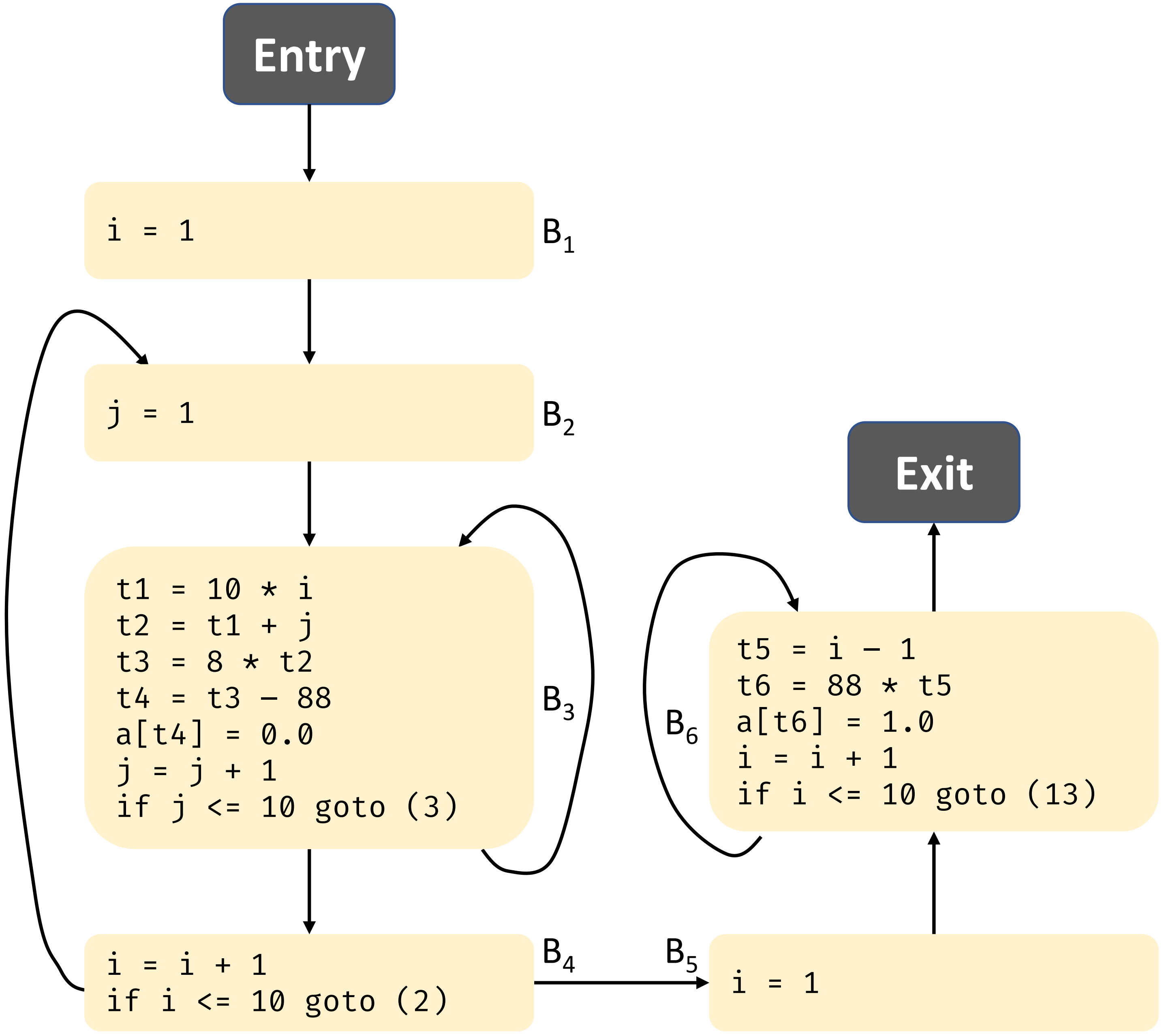


```
prod = 0
i = 1
```

```
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

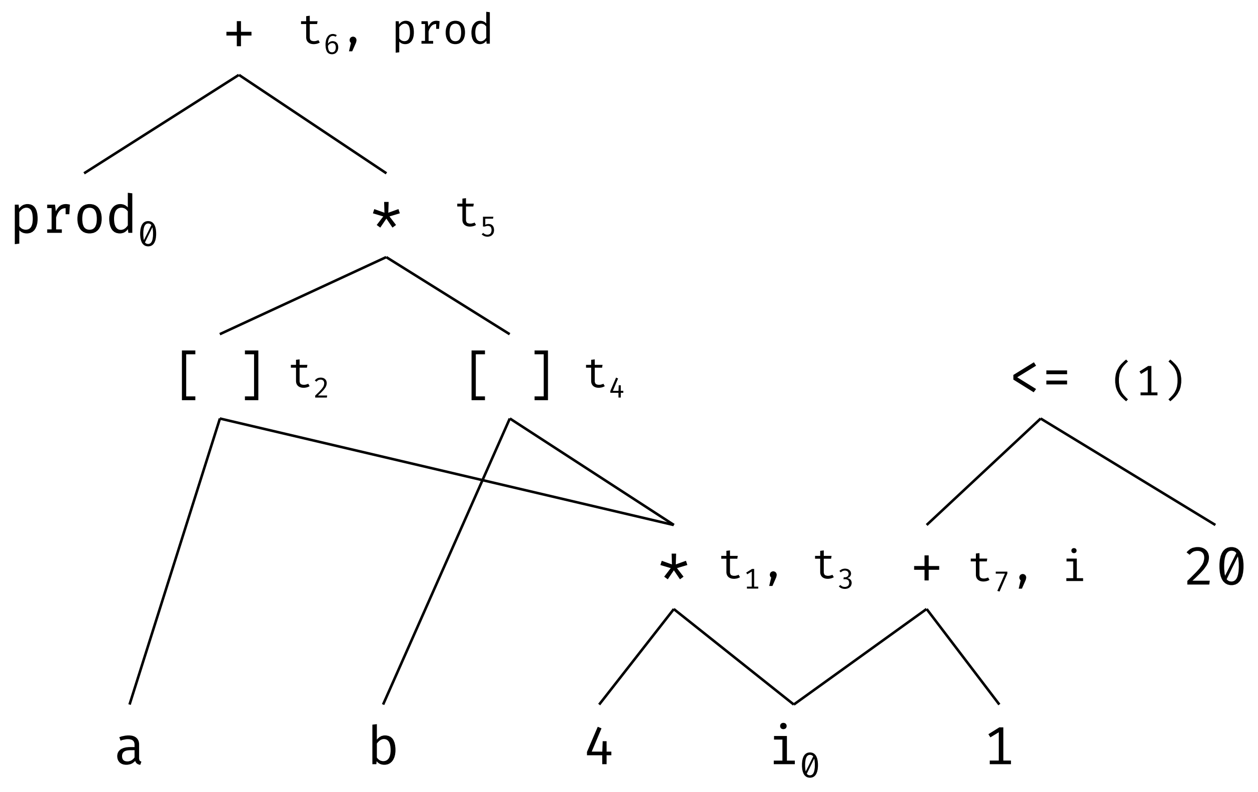
```
1) i = 1  // Leader
2) j= 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Optimizing BBs

Example 3AC

```
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 (1)
```



Representing BBs with DAGs

- Rules
  - Initial values for each variable is represented by a node
    - Leave nodes are labeled with variable names or constants
  - 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
- **Assumption**
  - Three kinds of 3AC: (i) $x = y \text{ op } z$, (ii) $x = \text{op } y$, and (iii) $x = y$

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
       - For case (i), create a leaf labeled z and set $node(z)$ 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
    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
    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

Optimization of BBs

- Code optimization can lead to substantial improvement in running time and/or energy consumption
- Global optimization analyzes control and data flow among BBs
    - Performs control flow, data flow, 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 determining common sub-expressions

Local Optimizations in a BB

Local common subexpression elimination

- Instructions compute a value that has already been computed

Dead code elimination

- Instructions compute a value that is never used

Statement reordering

- Reorder statements that with no dependence
- May improve latency of accesses and register usage

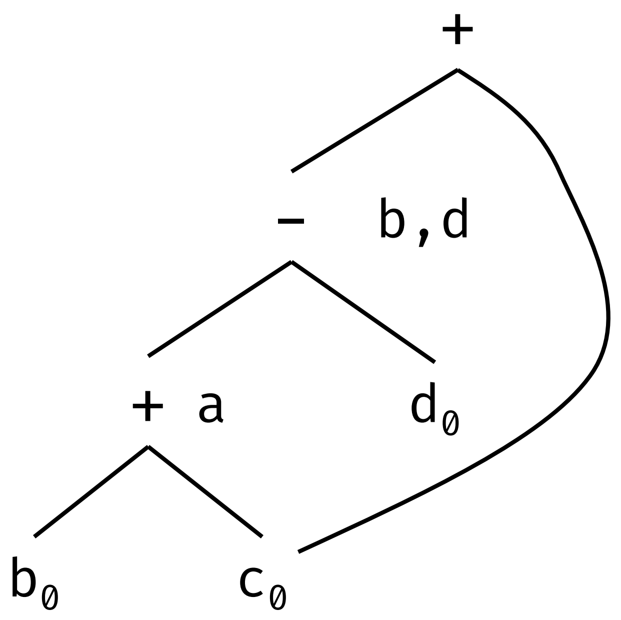
Algebraic simplification

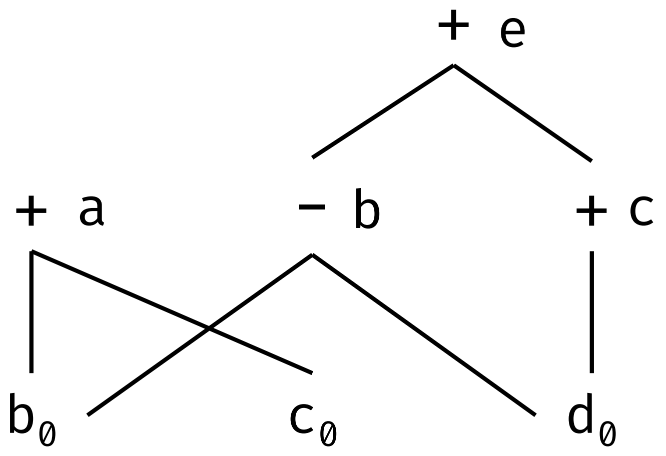
- Apply algebraic laws to simplify computation

Local Common Subexpressions

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

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



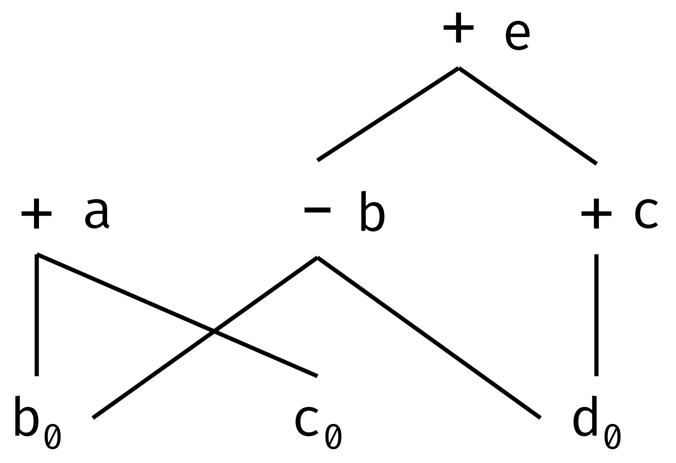


Dead Code Elimination

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

- Assume a and b are live but c and e are not live

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



Code Generation Algorithm

Code Generation Strategy

- 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

`a = b + c`	`ADD Rj, Ri`	b is in Ri, c is in Rj, b is no longer live
	`ADD c, Ri`	b is in Ri, b is no longer live

	`MOV c, Rj`
	`ADD Rj, Ri`
		b is in Ri, b is no longer live

Usually there can be numerous cases to consider

- Properties of the operator (for example, commutativity) can add to the complexity

Code Generation Strategy

- **Goal**: Generate target code for a sequence of 3AC
- **Assumptions**
  - Track whether variables are in registers
  - 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,
    2. Or just before a procedure call, jump, or a labelled statement
       - Implies every register must be saved before the end of a BB

Register and Address Descriptors

Register descriptor

- Keep 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
- 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 from the current locations of y
    - Prefer a register for y' if it exists
  - Perform the same steps for z
  - Generate the instruction $\text{ADD } R_x, R_y, R_z$

Code Generation for Copy Instruction $x = y$

- y is in a register
  - Change the address and register descriptors to indicate that x is now in the register originally holding y
  - If y has no next use and is not live on exit from the BB, the register no longer holds the value of y
- y is in memory
  - Invoke $getreg$ to find a register to load y, make that register the location for x
  - Alternatively, generate $\text{MOV } y, x$ instruction if the value of x has no next use in the BB

Updating 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

Updating Descriptors

- For an instruction such as $\text{ADD } R_x, R_y, R_z$, 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
  - Add x to the register descriptor for R_y
  - Change the address descriptor for x so that its only location is R_y

Defining Function $getreg()$

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

Defining Function $getreg()$

- If y is in a register, then return a register containing y as R_y
- If y is not in a register, but there is an empty register available, then pick one such register as R_y
- If y is not in a register and no empty register is available
    - Let R be a candidate register and suppose v is one of the variables stored in R
    - If the address descriptor for v says that v is somewhere else beside R, then choose R
    - Otherwise if v is x, and x is not an operand of I (i.e., $x \neq z$), then choose R
    - Otherwise if v is not used later, then choose R
    - Else, generate $\mathrm{ST}\ v, R$ (called a register spill)

Defining Function $getreg()$

- R may hold several variables, so we need to repeat for each such variable
- Compute a score for R
    - Score is the number of store instructions generated
- Pick a register with the lowest score
- We need to consider similar issues for z and R_z

Defining Function $getreg()$

- Consider selection of a register R_x for x
  - 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	$d, R2$		$R1$	$R3$

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

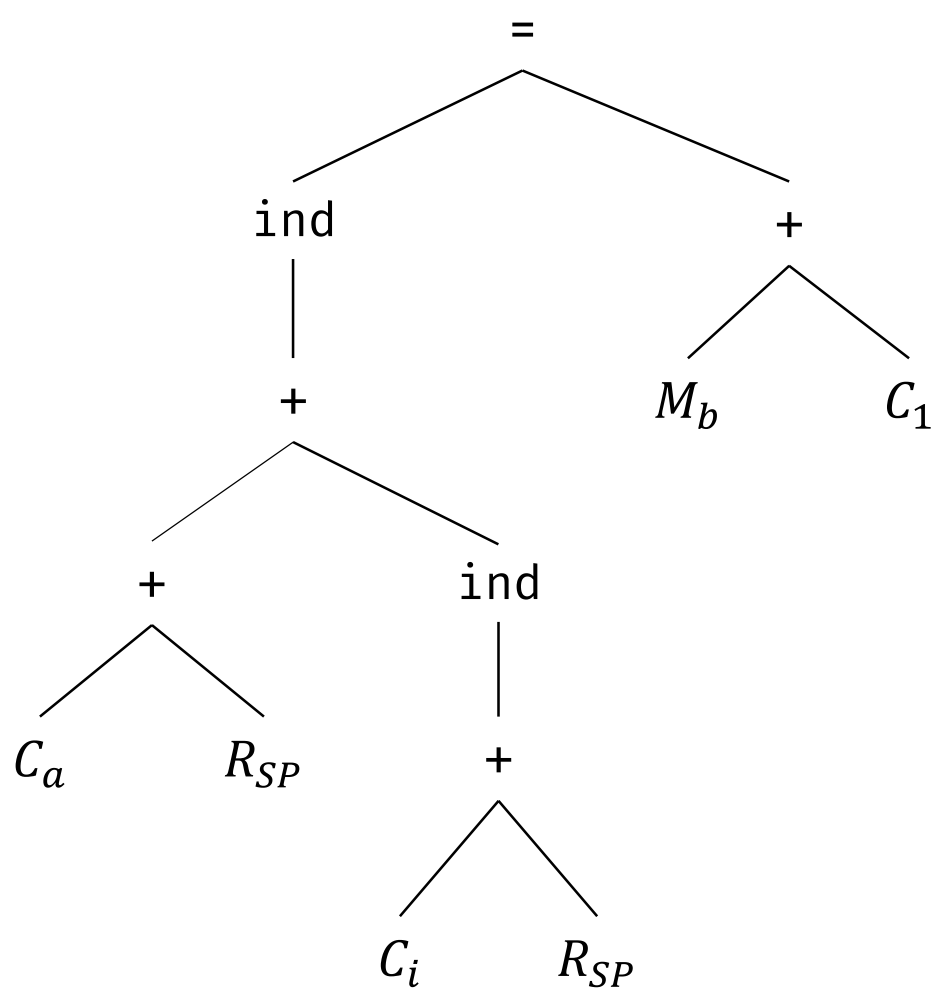
3AC	i in register Ri	i in memory Mi	i in Stack
$a = b[i]$	MOV $b(Ri), R$	MOV Mi, R MOV $b(R), R$	MOV $Si(A), R$ MOV $b(R), R$
$a[i] = b$	MOV $b, a(Ri)$	MOV Mi, R MOV $b, a(R)$	MOV $Si(A), R$ MOV $b, a(R)$

3AC	p in register Rp	p in memory Mp	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

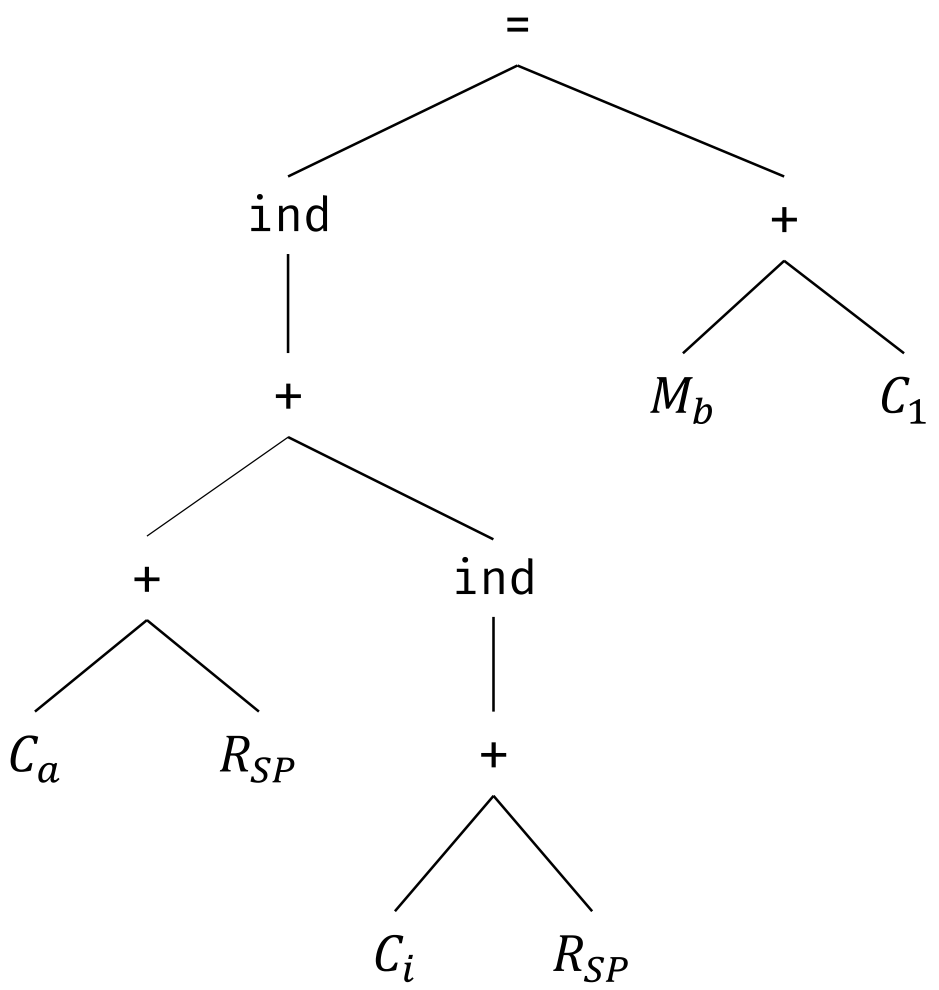
Tree Representation

- Consider the statement
$$a[i] = b + 1$$
  - Assume b is in memory location M_b
  - Array a is a local and is stored on the stack
  - Width of array values are 1B
  - Addresses of locals a and i are given as constant offsets C_a and C_i from the register SP
  - SP points to the beginning of the current activation record



Tree Representation

- `ind` operator (denotes indirection) treats its argument as a memory address



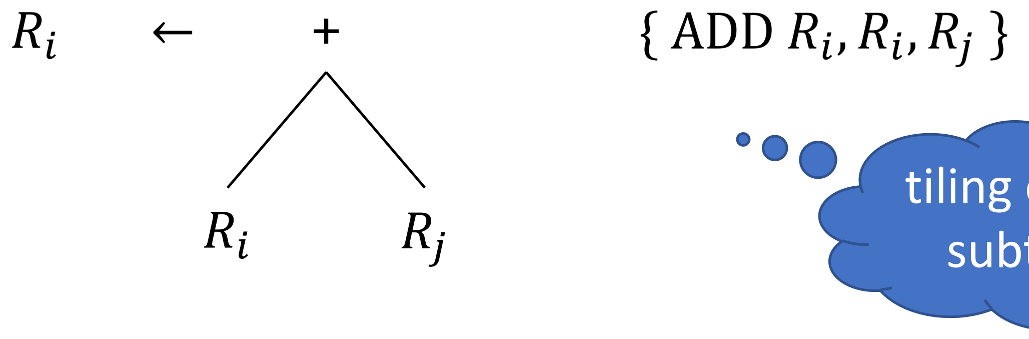
Tree Rewriting

- 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

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

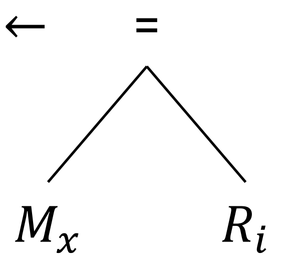


Tree Rewriting Rules

$R_i \quad \leftarrow \quad C_a \qquad \{ \text{ADD } R_i, \#a \}$

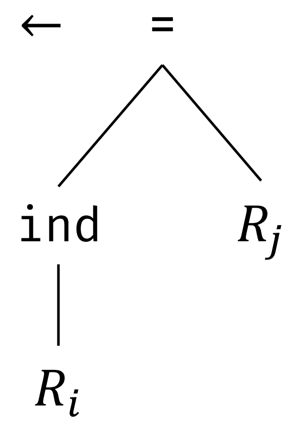
$R_i \quad \leftarrow \quad M_x \qquad \{ \text{ADD } R_i, x \}$

$M \quad \leftarrow$



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

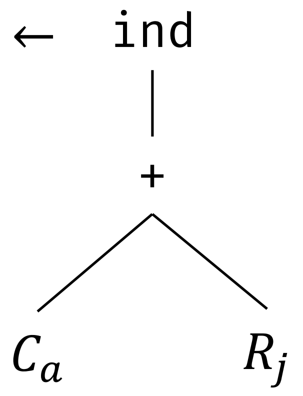
$M \quad \leftarrow$



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

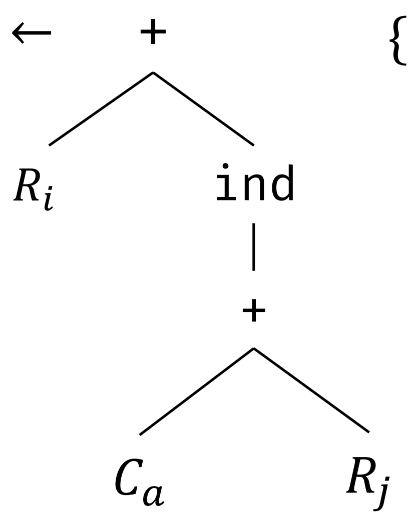
Tree Rewriting Rules

$R_i \leftarrow$



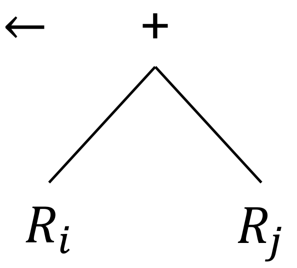
$\{ \text{LD } R_i, a(R_j) \}$

$R_i \leftarrow$



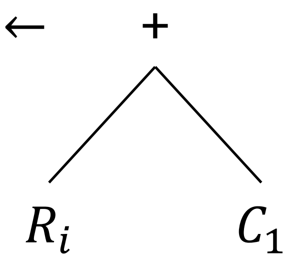
$\{ \text{ADD } R_i, R_i, a(R_j) \}$

$R_i \leftarrow$



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

$R_i \leftarrow$

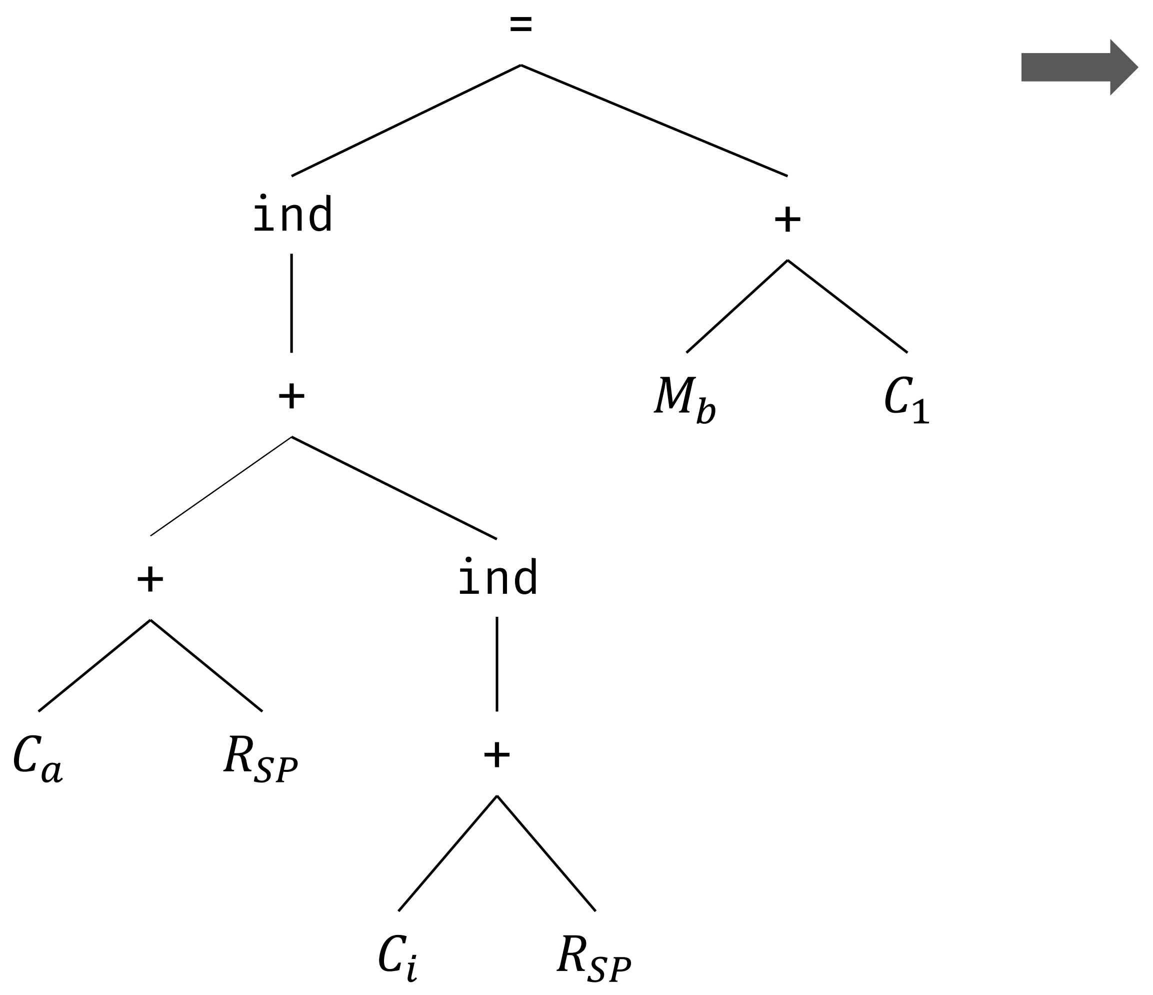


$\{ \text{INC } R_i \}$

Code Generation by Tiling an Input Tree

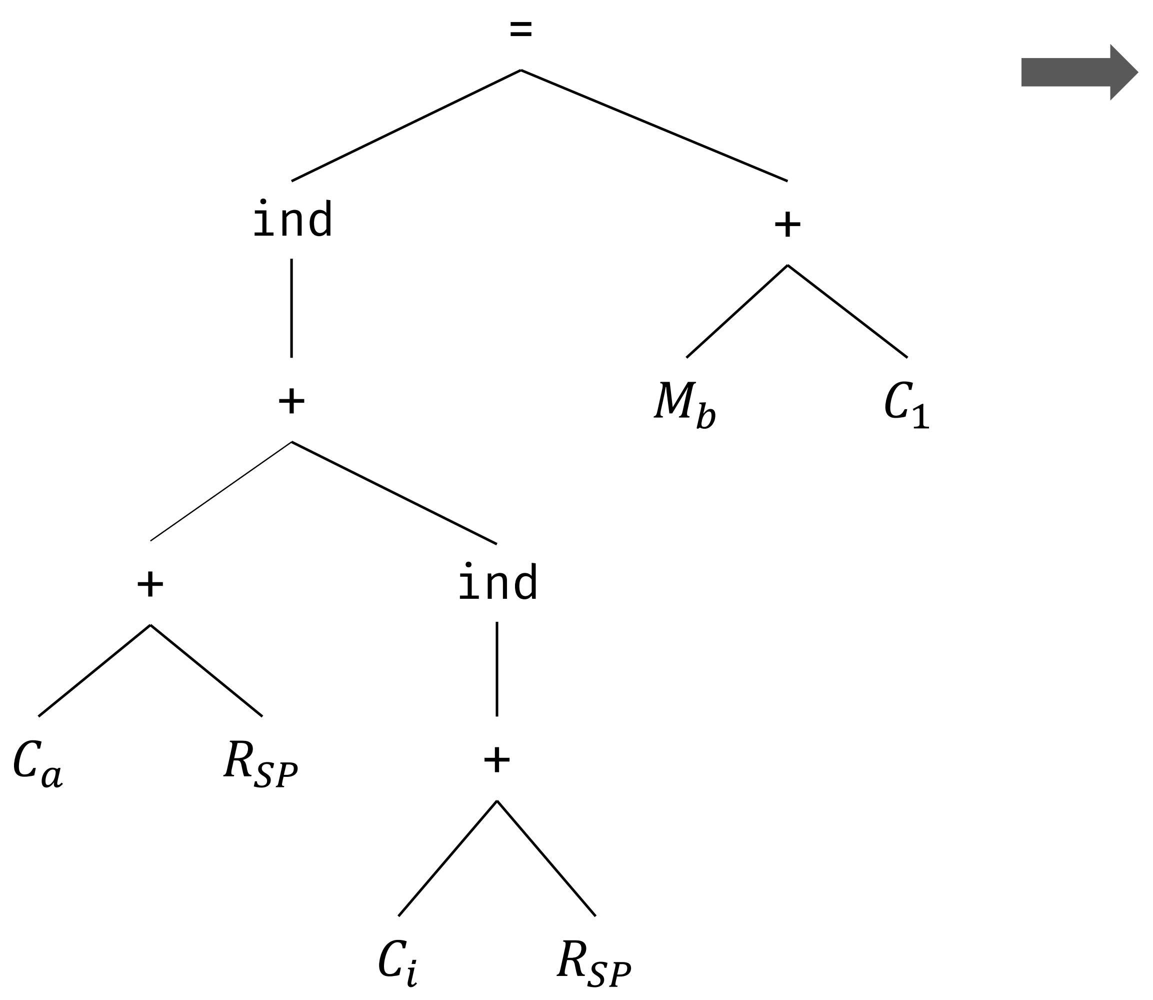
- High-level steps in a tree-translation scheme
  - Given an input tree, the templates in the tree-rewriting rules are applied to tile its subtrees
  - If a template matches, replace the matching subtree with the replacement node of the rule
    - Execute the action associated with the rule
    - If 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



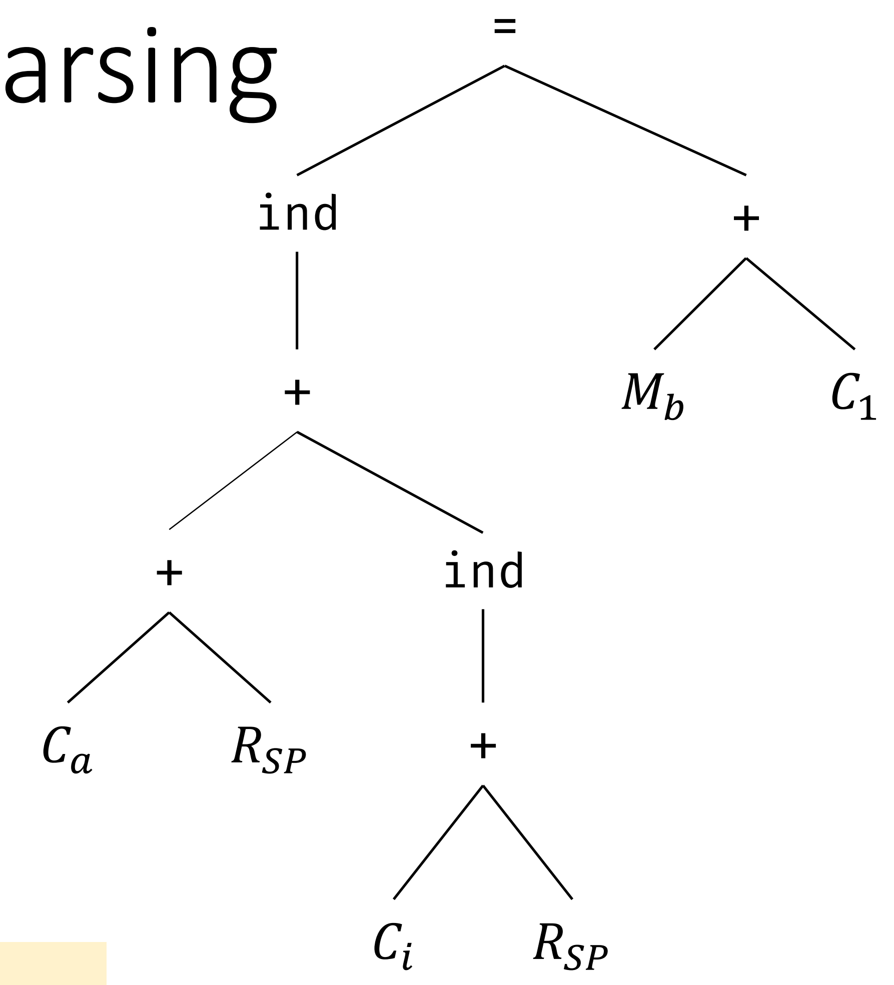
LD $R_0, \#a$
ADD $R_0, R_0,$ SP
ADD R_0, R_0, i(SP)
LD R_1, b
INC R_1
ST $*R_0, R_1$

Considerations during Tree Reduction

i. Performance of the tree matching algorithm impacts the efficiency of the code generation process 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
  - User a parsing mechanism for pattern matching
  - Come up with a syntax-directed translation (SDT) as an alternate for tree rewriting rules

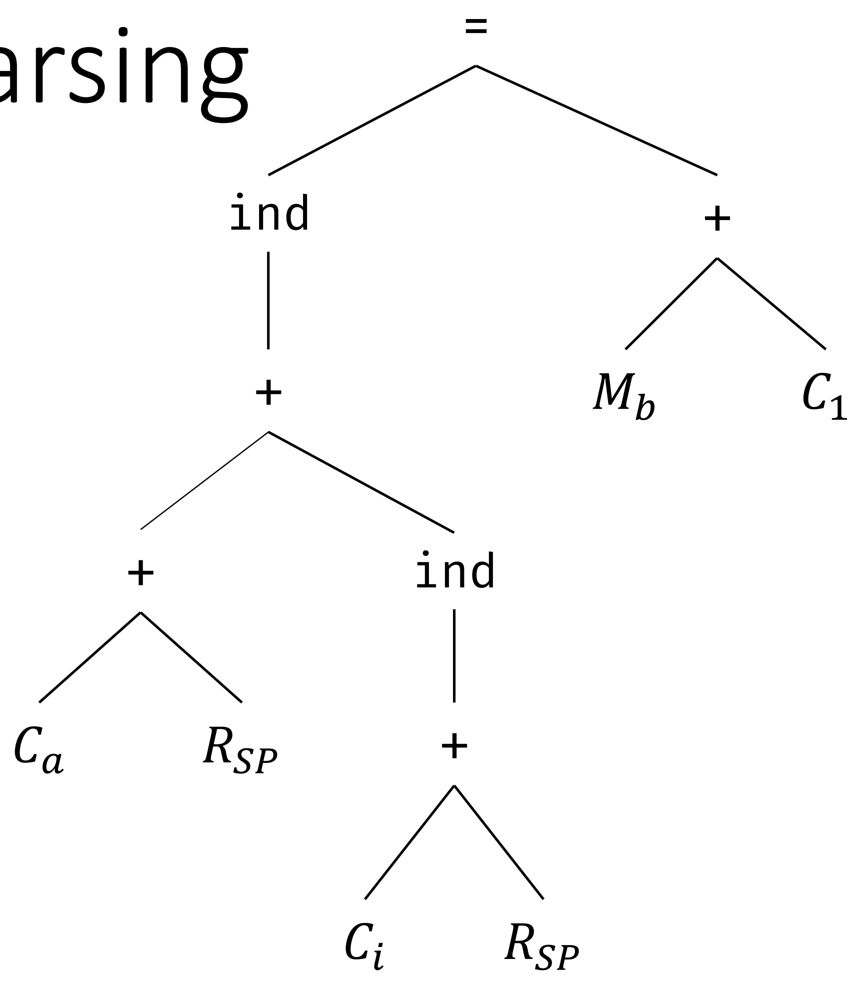


Prefix representation

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

Pattern Matching with LR Parsing

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



SDT for Tree Rewriting

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

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

Instruction Selection via Peephole Optimization

Peephole Optimization

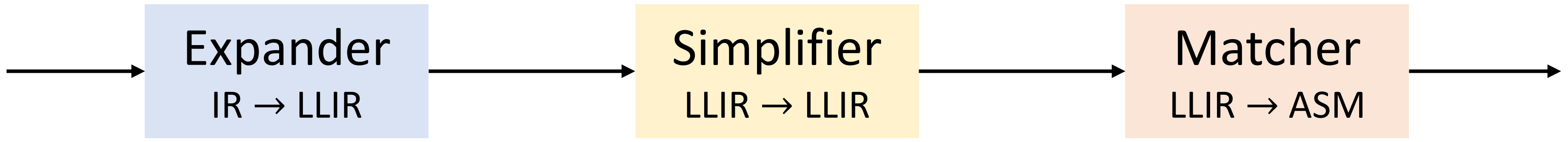
- **Insight**: Find local optimizations by examining short sequences of adjacent operations
  - The sliding window, or the peephole, moves over code
    - Code in a peephole need not be contiguous
  - Goal is to identify code patterns that can be improved
  - Rewrite code patterns with improved sequence

ST a, R_0
LD R_3, a
$\Rightarrow$
ST a, R_0
MOV R_3, R_0

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

Peephole Optimization based Code Generation

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


$$\begin{tikzcd} {} \arrow[r] & \boxed{\text{Expander}\ \text{IR} \rightarrow \text{LLIR}} \arrow[r] & \boxed{\text{Simplifier}\ \text{LLIR} \rightarrow \text{LLIR}} \arrow[r] & \boxed{\text{Matcher}\ \text{LLIR} \rightarrow \text{ASM}} \arrow[r] & {} \end{tikzcd}$$

- In an optimizer, the input and output language are the same
- With a different output language, the optimizer can be used for code generation

Peephole Optimization based Code Generation

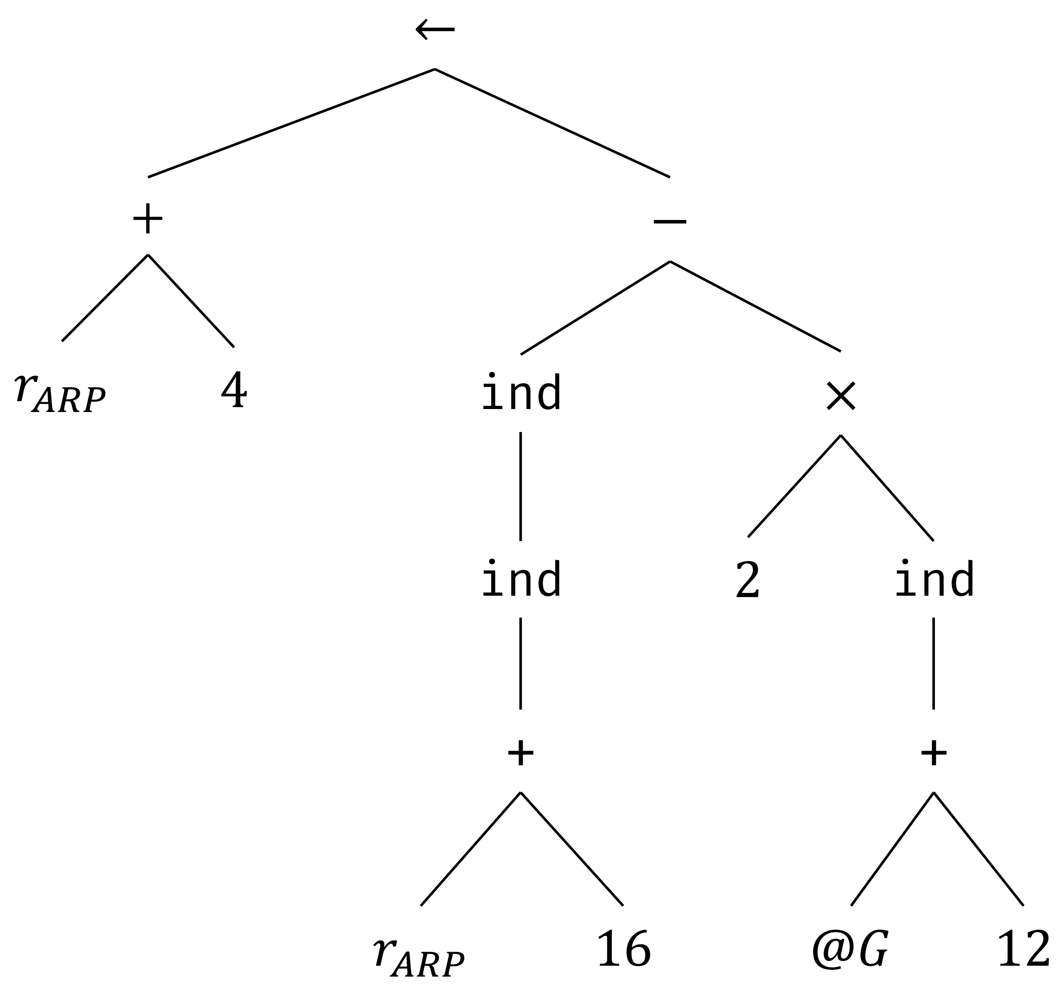
- Expander rewrites the IR to represent all the direct effects of an operation
  - If $\mathrm{ADD}\ R_0, R_1, R_2$ sets a condition code, then the LLIR should include an explicit operation to set the condition code
- Simplifier performs limited local optimization on the LLIR in the window
- Matcher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

Example

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

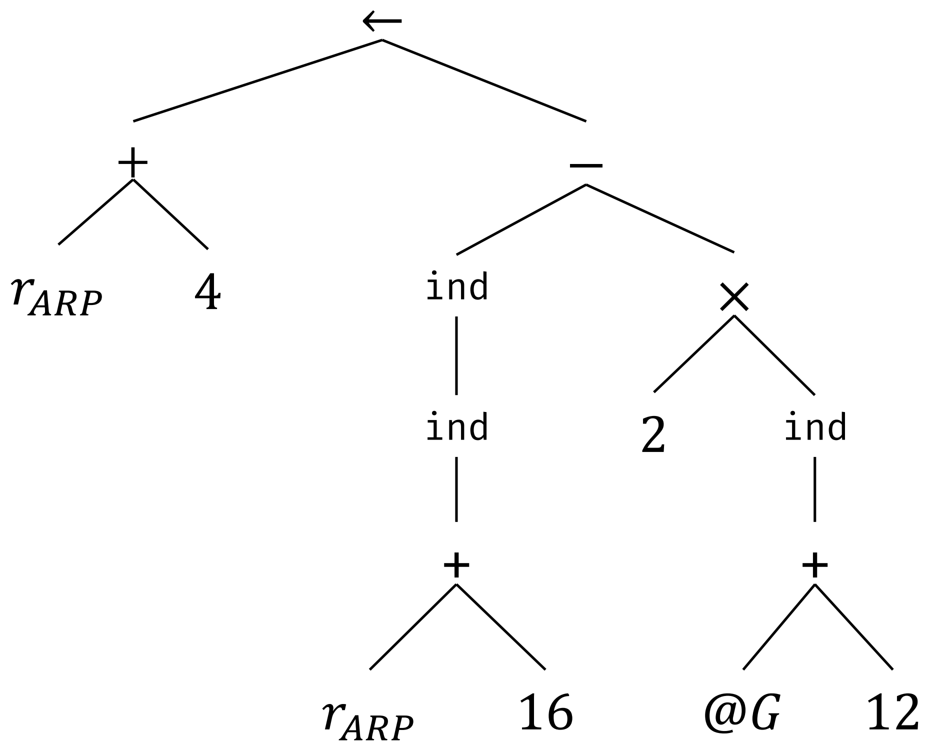
- a is stored at offset 4 in the local AR
- b stored as a call-by-reference parameter whose pointer is stored at offset – 16 from the ARP
- c is at offset 12 from the label $@G$

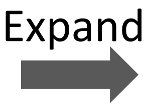
Op	Arg_1	Arg_2	Result
$\times$	2	c	t_1
$-$	b	t_1	a



Example

Op	Arg_1	Arg_2	Result
$\times$	2	c	t_1
$-$	b	t_1	a

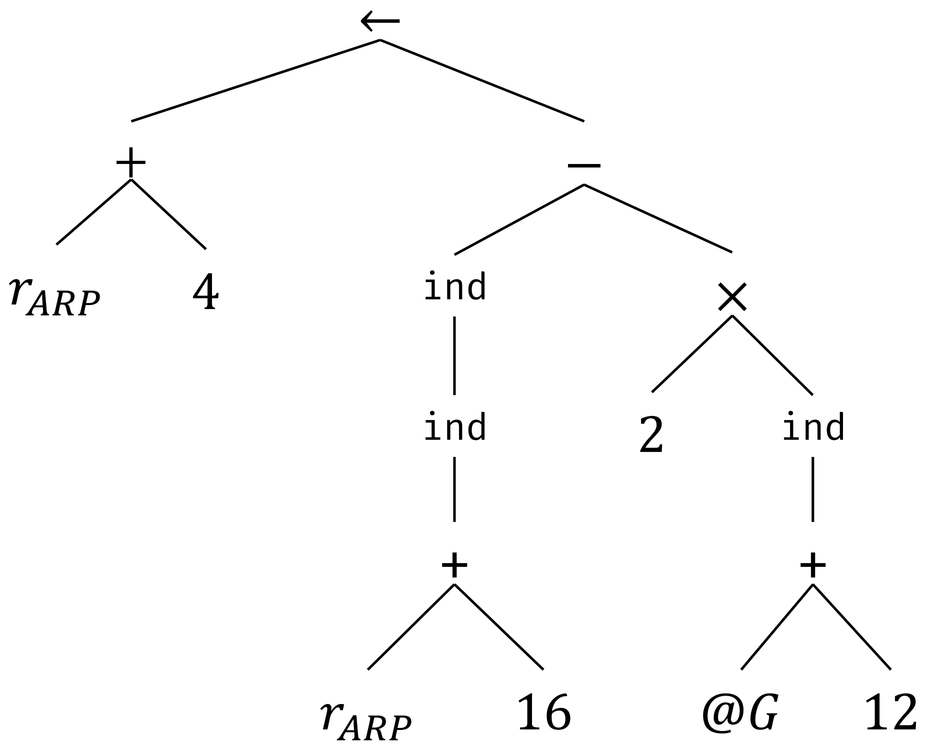




$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20}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

Example

Op	Arg$_1$	Arg$_2$	Result
$\times$	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

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

Assume a sliding window of size 3

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

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

Sequence 4

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

Sequence 5

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

Sequence 6

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

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 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

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

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

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

Sequences Produced by the Simplifier

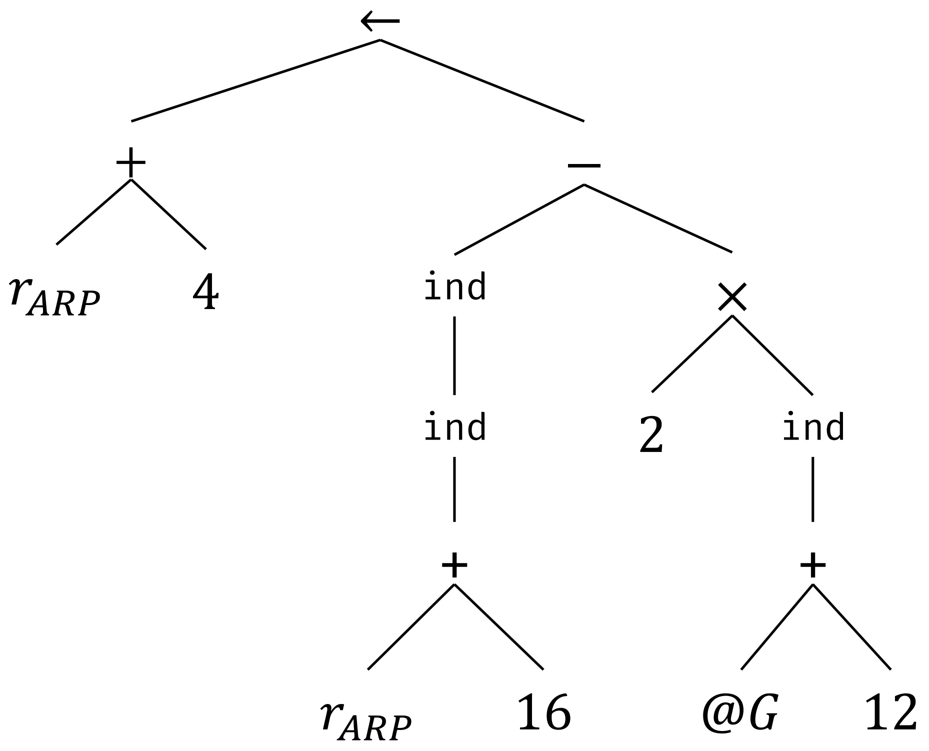
$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{12} \leftarrow 12$
$r_{13} \leftarrow r_{11} + r_{12}$
$r_{14} \leftarrow M(r_{13})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{16} \leftarrow -16$
$r_{17} \leftarrow r_{ARP} + r_{16}$
$r_{18} \leftarrow M(r_{17})$
$r_{19} \leftarrow M(r_{18})$
$r_{20}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

Sequence 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

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

Simplify

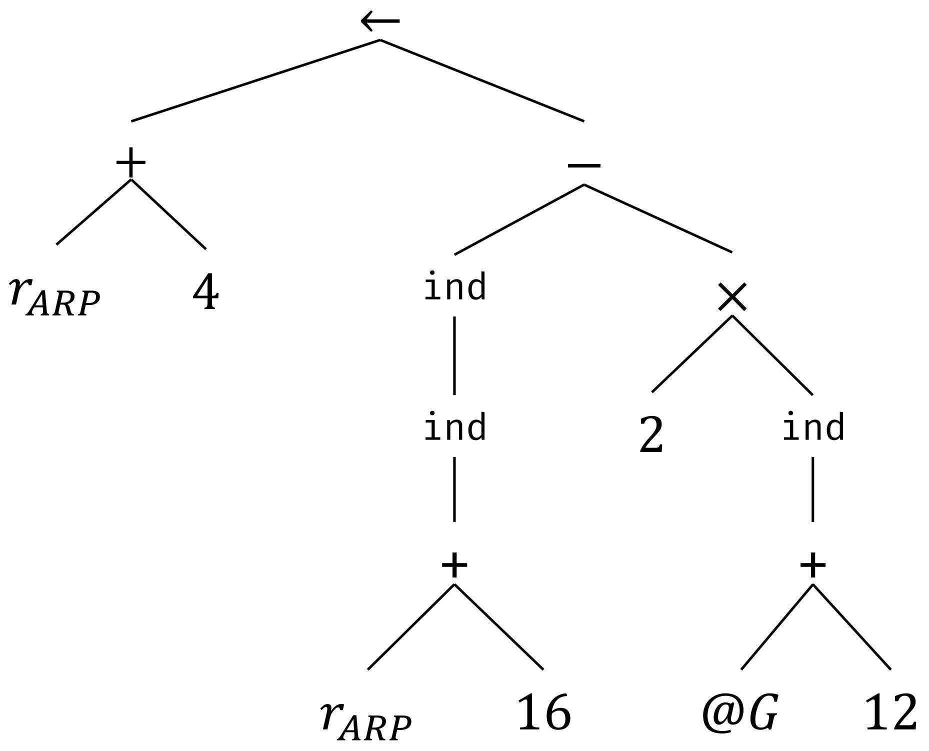
$r_{10} \leftarrow 2$
$r_{11} \leftarrow @G$
$r_{14} \leftarrow M(r_{11} + r_{12})$
$r_{15} \leftarrow r_{10} \times r_{14}$
$r_{18} \leftarrow M(r_{ARP} - 16)$
$r_{19} \leftarrow M(r_{18})$
$r_{20} \leftarrow r_{19} - r_{15}$
$M(r_{ARP} + 4) \leftarrow r_{20}$

Match

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

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

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

Match

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

Current State

- 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 back end uses a peephole scheme to convert RTL into assembly code

References

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