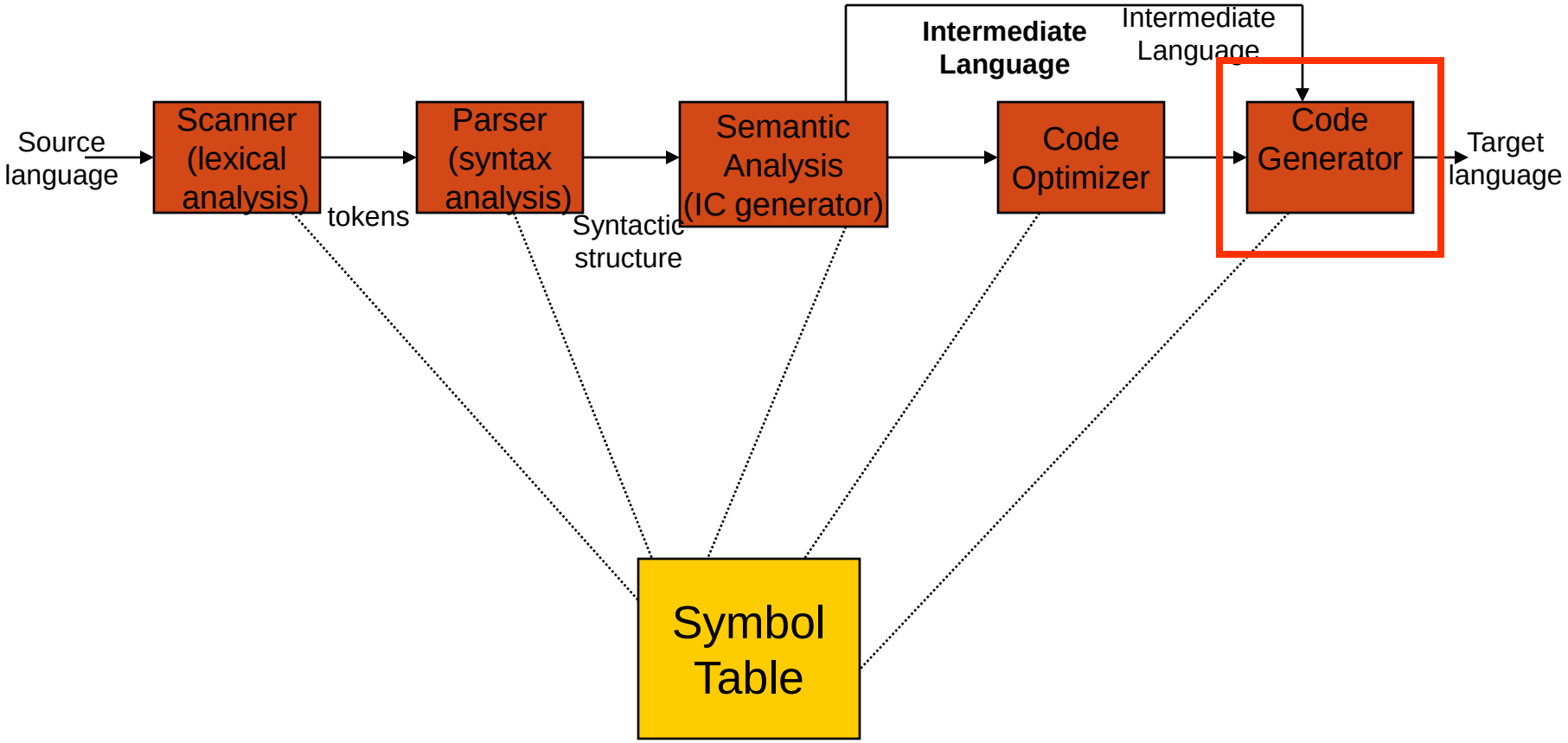
CS 346: Code Generation

Resource: Textbook

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*", Addison-Wesley, 1986.

Compiler Architecture



Code Generator

- Severe requirements imposed
  - Output must be correct and of high quality
  - Generator should run efficiently
- Generating optimal code is undecidable
  - Must rely on heuristic
  - Choice of heuristic-*very important*
- Details depends on
  - target language
  - operating system
- Certain generic issues are inherent in the design of basically all code generators

Input to Code Generator

- Input to the code generator consists of:
  - Intermediate code produced by the front end (*and perhaps optimizer*)
    - Remember that intermediate code can come in many forms
    - Three-address codes are more popular but several techniques apply to other possibilities as well
  - Information in the symbol table
    - used to determine run-time addresses of data objects
- Code generator typically assumes that:
  - Input is free of errors
  - Type checking has taken place and necessary type-conversion operators have already been inserted

Output of Code Generator

- *Target program*: output of the code generator

- Various representations of target forms
  - absolute machine language
  - relocatable machine language
    - Can compile subprograms separately
    - Added expense of linking and loading
  - assembly language
    - Makes task of code generation simpler
    - Added cost of assembly phase

Memory Management

- Compiler must map names in source code to addresses of data objects at run-time
  - Done cooperatively by front-end and code generator
  - Generator uses information in symbol table

- For the generated machine code:
  - Labels in three-address statements need to be converted to addresses of instructions
  - Process is analogous to backpatching

Instruction Selection

- Complexities
  - The level of intermediate representation
  - Nature of the instruction-set architecture
  - Desired quality of the generated code
- Intermediate representation (high level)
  - Translate each intermediate statement into a sequence of machine instructions
  - Statement-statement code generation is not efficient
  - Needs further optimization
- Intermediate representation (low level)
  - Low-level details can be useful for generating efficient codes

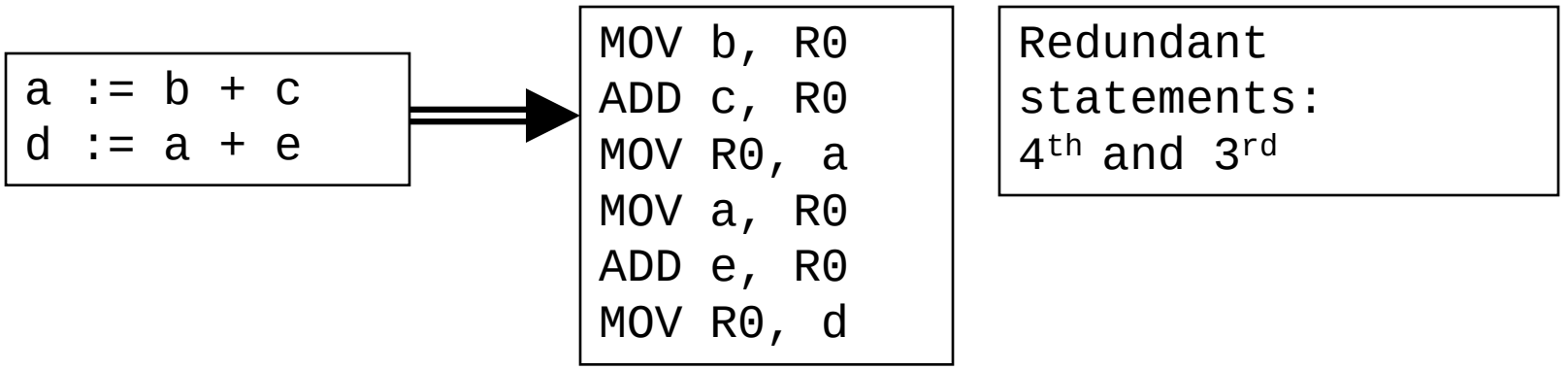
Instruction Selection

- If efficiency is not a concern, instruction selection is straight-forward
  - For each type of three-address statement, there is a code skeleton that outlines target code
  - Example, `x := y + z`, where `x`, `y`, and `z` are statically located, can be translated as:

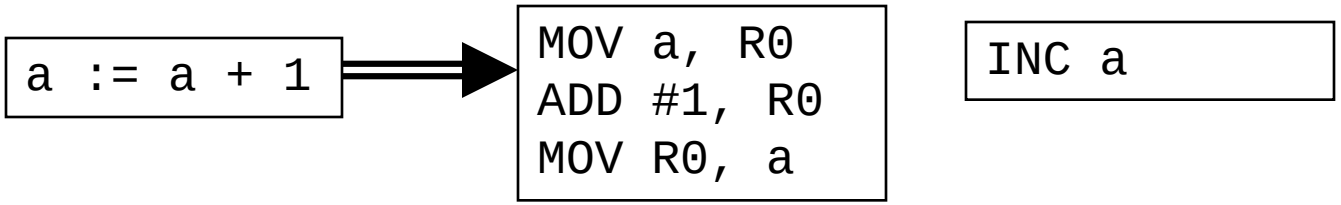
```
MOV y, R0 /* load y into register r0 */
ADD z, R0 /* add z to r0 */
MOV R0, x /* store R0 into x */
```

Instruction Selection

- Often, the straight-forward technique produces poor code:



- A naïve translation may lead to correct but unacceptably inefficient target code:



Register Allocation

- Instructions are *usually faster* if operands are in registers instead of memory
- Efficient utilization of registers is important in generating good code
- *Register allocation* selects the set of variables that will reside in registers
- A *register assignment* phase picks the specific register in which a variable resides
- Finding an optimal assignment of registers to variables is difficult
  - Problem is NP-complete
  - Certain machines require register-pairs for certain operators and results

Address Modes of Sample Machine

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
Register	R	R	0
Indexed	c(R)	c + contents(R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c + contents(R))	1
MODE	**FORM**	**CONSTANT**	**ADDED COST**
literal	#c	c	1

Instruction Costs

- Cost of instruction for sample computer:
  - 1+ costs associated with the source and destination address modes
  - Corresponds to the length (in words) of the instruction
  - Address modes involving registers have cost zero
  - With memory location or literal have cost one since operands have to be stored with instruction

- *Time taken to fetch instruction often exceeds time spent executing instruction*

Example: a := b + c

- The following solution has cost 6:

```
MOV b, R0
ADD c, RO
MOV R0, a
```

- The following solution also has cost 6:

```
MOV b, a
ADD c, a
```

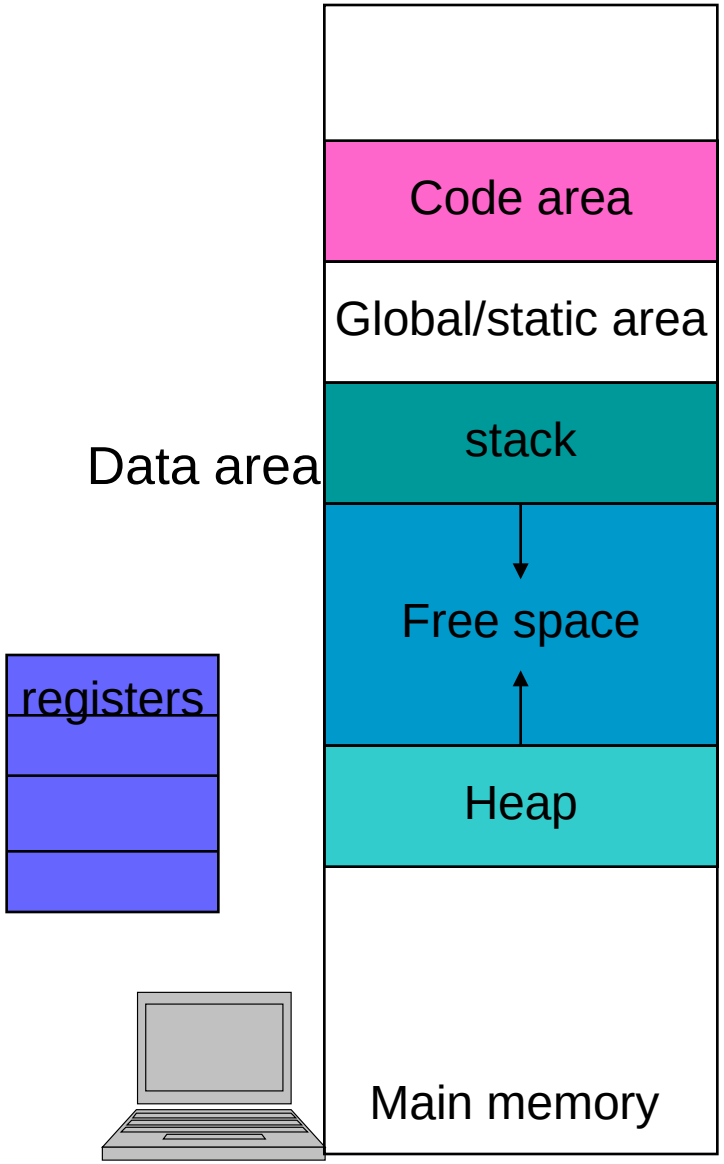
- If `R0`, `R1`, and `R2` contain the addresses of `a`, `b`, and `c`, respectively, the cost can be reduced to 2:

```
MOV *R1, *R0
ADD *R2, *R0
```

- If `R1` and `R2` contain the values of `b` and `c`, respectively, and `b` is not needed again, the cost can be reduced to 3:

```
ADD R2, R1
MOV R1, a
```

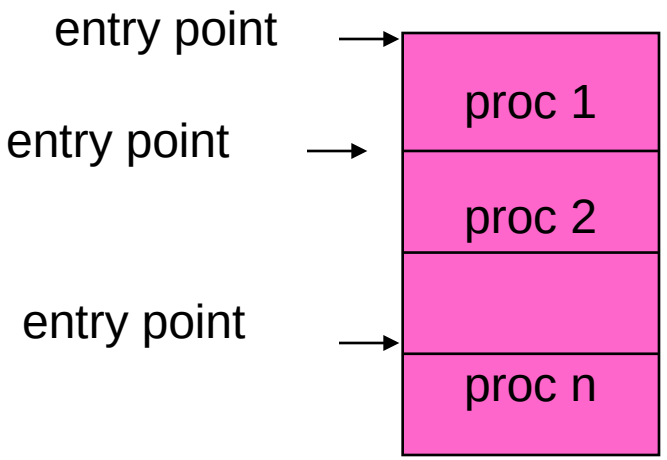
Memory organization during program execution



- Main memory
  - Store large amount of data
  - Slower access
- Registers
  - Very small amount of data
  - Faster access

Code Area

- Addresses in code area are static (i.e. no change during execution) for most programming languages
- Addresses are known at compile time



Data Area

- Addresses in data area are static for some data and dynamic for others
  - Static data are located in static area
  - Dynamic data are located in stack or heap
    - Stack (LIFO allocation) for procedure activation record, etc.
    - Heap for user allocated memory, etc.

Registers

- General-purpose registers
  - Used for calculation
- Special purpose registers
  - Program counter (pc)
  - Stack pointer (sp)
  - Frame pointer (fp)
  - Argument pointer (ap)

Addresses in Target Code (partitions)

- Program runs in own logical address

- Partitions of logical addresses
  - Code
    - Statically determined area that holds executable target code
    - Size of target code determined at compile time

  - Static
    - Holds global constants and other data
    - Size of these entities determined at compile time

Addresses in Target Code

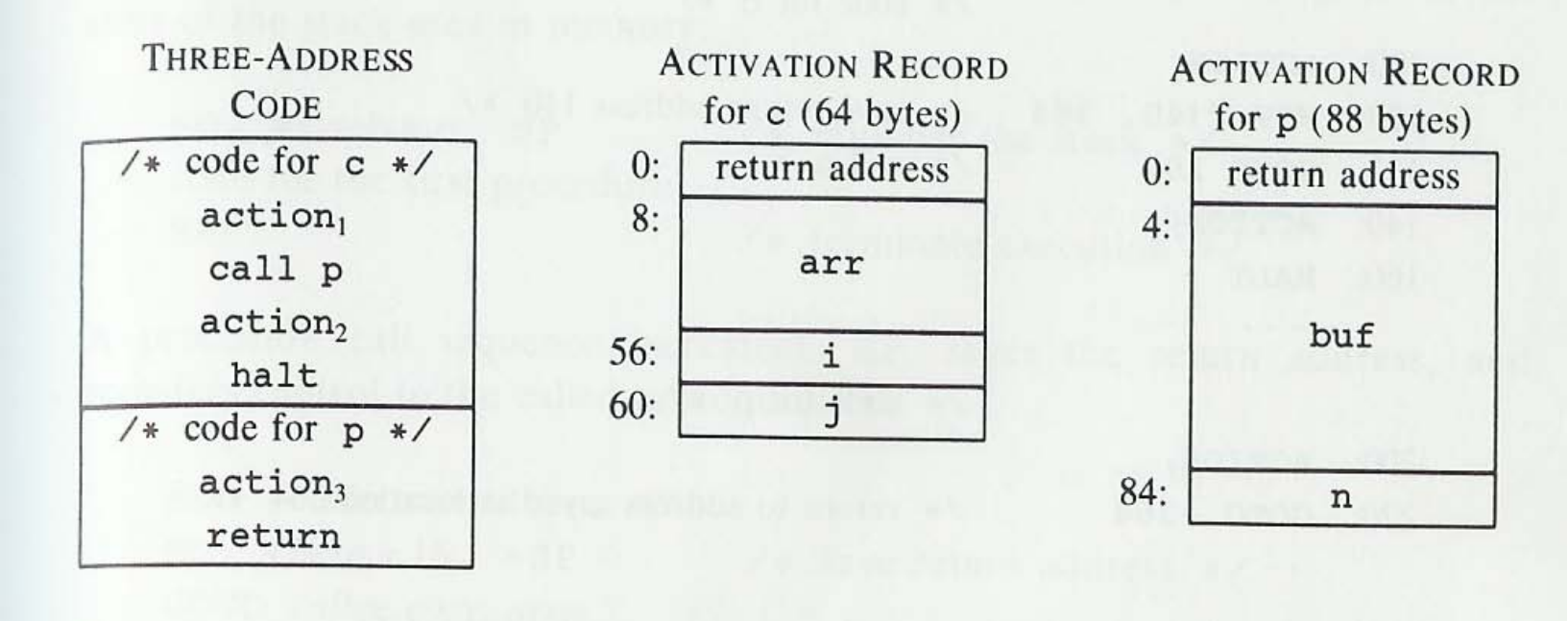
- Heap
  - Holds data objects created and freed during program execution
  - Size of heap can't be determined at compile time
- Stack
  - Dynamically managed area for activation records (created and destroyed during procedure calls and returns)
  - Size cant be determined at compile time

Activation Records

- Activation records store information needed during the execution of a procedure
- Two possible storage-allocation strategies for activation records:
  - static allocation (decision can be taken by looking at program)
  - stack allocation (decision taken at run time)
- An activation record has fields which hold:
  - result and parameters
  - machine-status information
  - local data and temporaries
- Size and layout of activation records are communicated to code generator via information in the symbol table

Sample Activation Records

- Assume that run-time memory has areas for *code*, *static data*, and optionally a *stack*
- Heap is not being used in these examples



Static Allocation

- A *call* statement in the intermediate code is implemented by two target-machine instructions:

  MOV #here+20, callee.static_area
  GOTO callee.code_area

  - MOV instruction saves the return address
  - GOTO statement transfers control to the target code of the called procedure

- A *return* statement in the intermediate code is implemented by one target-machine instruction:

  GOTO *callee.static_area

Sample Code

```
                        /* code for c */
100:  action1
120:  MOV #140, 364     /* save return address 140 */
132:  GOTO 200          /* call p */
140:  action2
160:  HALT
      …
                        /* code for p */
200:  action3
220:  GOTO *364
      …
                        /* 300-363: activation record for c */
300:                    /* return address */
304:                    /* local data for c */
      …
                        /* 364–451: activation record for p */
364:                    /* return address */
368:                    /* local data for p */
```

Stack Allocation

- Stack allocation uses relative addresses for storage in activation records
  - address can be offset from any known position in activation record
  - uses positive offset from SP, a pointer to beginning of activation record at top of stack
- Position of an activation record is not known until run-time
  - position usually stored in a register
  - indexed address mode is convenient

Stack Allocation

- The code for the first procedure initializes the stack:

```
LD SP, #stackstart          //intialize the stack
…code for the first procedure…
HALT
```

- A procedure call increments SP, saves the return address, and transfers control:

```
ADD SP, #caller.recordsize //increment SP
ST 0(SP), #here+16          //save return address
BR callee.code_area         //jump to the calle
```

- A return sequence has two parts:
  - First control is transferred to the return address

```
BR *0(SP)            //return to caller
```

  - Next, in the caller, SP is restored to its previous value

```
SUB SP, #caller.recordsize
```

Stack Allocation

```
action1      //code for m
call q
action2
halt
action3      //code for p
return
action4      //code for q
call p
action5
call q
action6
call q
return
```

Stack Allocation

msize: size of activation record for procedure m (20)

psize: size of activation record for procedure p (40)

qsize: size of activation record for procedure q (60)

- First word in each activation record holds return address
- Code for m, p and q start at addresses 100, 200 and 300
- Stack starts at address 600

Stack Allocation

```
                          //Code for m
100: LD SP, #600        //Initializes STACK
108: ACTION1             //code for action 1
128: ADD SP, #msize   //call sequence begins
136: ST O(SP), #152   //push return address
144: BR 300              // call q
152: SUB SP, #msize    //restore SP
160: ACTION2
180: HALT
 …
 ….
```

Stack Allocation

```
                                    //Code for p
200: ACTION3
220: BR *O(SP)      //return
....
...
                            //code for q
300: ACTION4
320: ADD SP, #qsize   //call sequence begins
328: ST O(SP), #344   //push return address
336: BR 200                 // call p
344: SUB SP, #qsize     //restore SP
352: ACTION5
372: ADD SP, #qsize
  ...
  ....
```

Stack Allocation

```
380: ST 0(SP), #396      //push return address
388: BR 300              //call q
396: SUB SP, #qsize      //restore SP
404: ACTION6
424: ADD SP, #qsize
432: ST 0(SP), #448      //push return address
440: BR 300
448: SUB SP, #qsize
456: BR  *0(SP)          //return
.....
....
600:                     //stack starts here
```

Basic Blocks

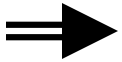
- A basic block is a sequence of statements such that:
  - Flow of control enters at start
  - Flow of control leaves at end
  - No possibility of halting or branching except at end
- A name is "*live*" at a given point if its value will be used again in the program
- Each basic block has a first statement known as the "leader" of the basic block

Partitioning Code into Basic Blocks

- Determination of leaders:
  - The first statement is a leader
  - Any *statement* that is the *target of a conditional or unconditional goto* is a *leader*
  - Any *statement* immediately *following a goto or unconditional goto* is a *leader*
- A basic block:
  - Starts with a leader
  - Includes all statements up to but not including the next leader

Basic Block Example

```
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) t1 := 4 * i
 (4) t2 := a[t1]
 (5) t3 := 4 * i
 (6) t4 := b[t3]
 (7) t5 := t2 * t4
 (8) t6 := prod + t5
 (9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto 3
(13) …
```

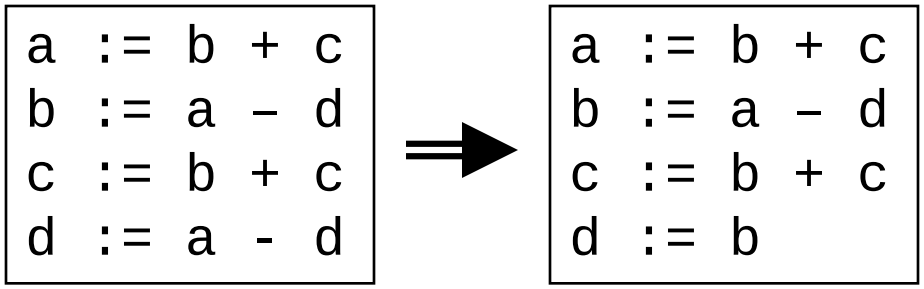
Transformations on Basic Blocks

- Basic block computes a set of expressions
  - *Expressions*: values of names that are live on exit from the block
  - Two basic blocks are equivalent if they compute the same set of expressions

- Certain transformations can be applied without changing the computed expressions of a block
  - Optimizer uses such transformations to improve *running time* or *space requirements* of a program
  - Important classes of local transformations:
    - structure-preserving transformations
    - algebraic transformations

Example Transformations (1)

- Algebraic Transformations:
  - Statements such as `x := x + 0` or `x := x * 1` can be safely removed
  - Statement `x := y ^ 2` can be safely changed to `x := y * y`
- Structure preserving transformations
  - Common sub-expression elimination



-Dead-code elimination

- Suppose the statement `x := y + z` appears in a basic block and `x` is dead (i.e. never used again)
- This statement can be safely removed from the code

Example Transformations (2)

- Renaming of Temporary variables
  - Statement `t := b + c` appears in a basic block and `t` is a temporary
  - We can safely rename all instances of this `t` to `u`, where `u` is a new temporary
  - A normal-form block uses a new temporary for every statement that defines a temporary
- Interchange of two independent adjacent statements
  - Suppose we have a block with two adjacent statements:

```
t1 := b + c
t2 := x + y
```

  - If t1 is distinct from `x` and `y` and `t2` is distinct from `b` and `c`, we can safely change the order of these two statements

Next-Use Information

- Defining use of a name in a three-address statement :
  - Suppose statement i assigns a value to x
  - Suppose statement j has x as an operand
  - We say that j uses the value of x computed at i if:
    - there is a path through which control can flow from statement i to statement j
    - path has no intervening assignments to x

- For each three-address statement that is of the form x := y op z
  - We want to determine the next uses of x, y, and z
  - May be within or outside the current basic block

Why next use information?

- *Knowing when the value of variable will be used next is essential for generating good code*

- *If value of variable stored in a register is never used then register can be freed for use of other's*

Determine Next-Use

- Scan each basic block from end to beginning
  - At start, record, if known, which names are live on exit from that block
  - Otherwise, assume all non-temporaries are live
  - If algorithm allows certain temporaries to be live on exit from block, consider them live as well

- Whenever reaching a three address statement at line `i` with the form `x := y op z`
  - Attach statement `i` to information in symbol table regarding the next use and liveliness of `x`, `y`, and `z`
  - Next, in symbol table, set `x` to "not live" and "no next use"
  - Then set `y` and `z` to "live" and the next uses of `y` and `z` to `I`

A similar approach is taken for unary operators

Flow Graphs

- A graphical representation of three-address statements, useful for optimization
- Nodes represent computations
  - Each node represents a single basic block
  - One node is distinguished as `initial`
- Edges represent flow-of-control
  - An edge exists from `B1` to `B2` if and only if `B2` can immediately follow `B1` in some execution sequence:
    - True if there is an conditional or unconditional jump from the last statement of `B1` to the first statement of `B2`
    - True if `B2` immediately follows `B1` and `B1` does not end with an unconditional jump
  - `B1` is a predecessor of `B2`, `B2` is a successor of `B1`

Flow Graph (Example)

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

Flow Graph

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

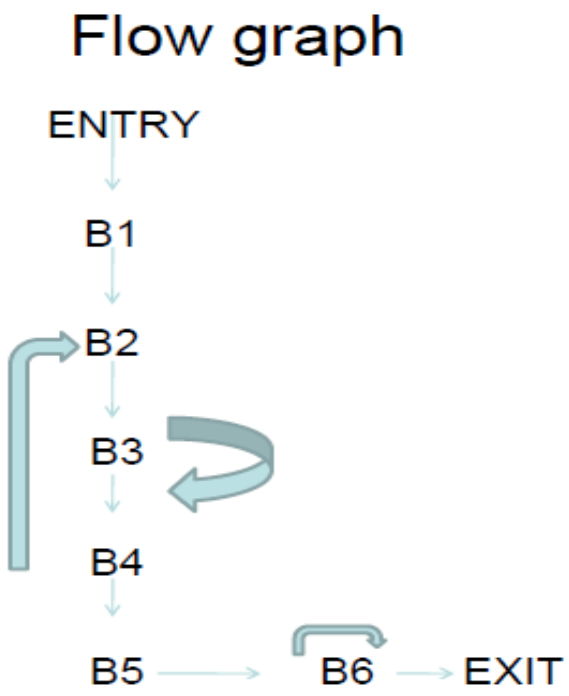
Intermediate code to set a 10 X 10 matrix to an identity matrix

Leaders and Basic Blocks

- Leaders
  - 1 (first instruction)
  - 2,3 and 13 (target of jumps)
  - 10 and 12 (following instructions of jumps)

- Basic blocks
  - B1: 1
  - B2: 2
  - B3: 3-9
  - B4: 10-11
  - B5: 12
  - B6: 13-17

Flow Graph



Representations of Flow Graphs

- A basic block can be represented by a record consisting of:
  - A count of the number of quadruples in the block
  - Or, a pointer to the last instruction
  - A pointer to the leader of the block
  - The list of predecessors and successors

- Alternative is to use linked list of quadruples
  - More efficient as number of instructions in basic block changes

- Explicit references to quadruple numbers in jump statements can cause problems:
  - Quadruples can be moved during optimization
  - Better to have jumps points to blocks

Loops and Flow Graphs

- Every program spends most of its time in executing its loops (*while*, *do-while*, *for* etc.)
  - Important for a compiler to generate good codes for loops

Set of nodes-L

Any node *e* called the loop entry such that

1. *e is not ENTRY, entry of the entire flow graph*
2. *No node in L besides e has a predecessor outside L*

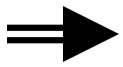
   -i.e. every path from ENTRY to any node in L goes through e

3. *Every node in L has a nonempty path, completely within L, to e.*

Reusing Temporaries

- Convenient during optimization for every temporary to have its own name
- Space can be saved
- Two temporaries can be packed into the same location if they are not live simultaneously

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



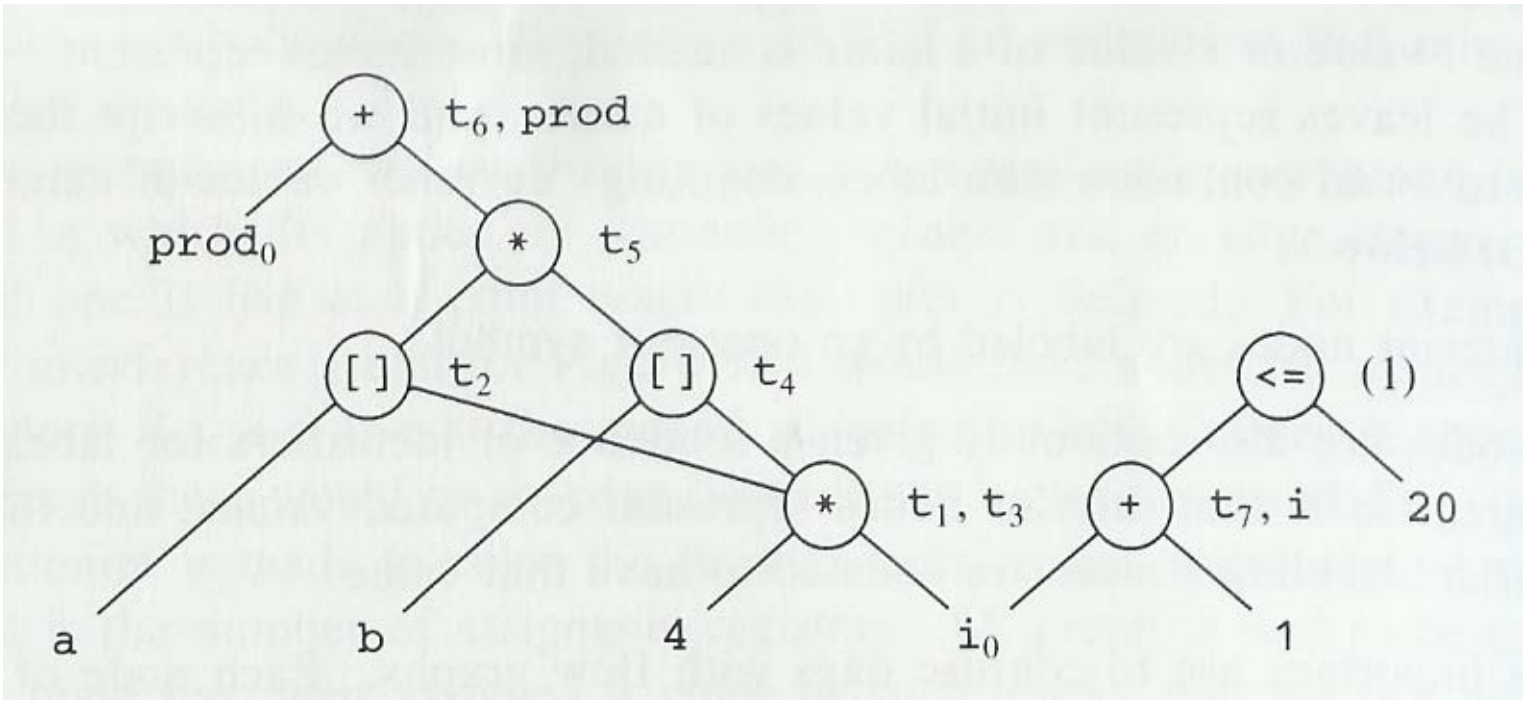
```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t1 := t1 + t3
t2 := b * b
t1 := t1 + t2
```

Directed Acyclic Graphs (DAGS)

- DAG: Represents basic block
  - Directed acyclic graph such that:
  - leaves represent the initial values of name
    - Labeled by unique identifiers, either variable names or constants
    - Operator applied to name determines if l-value (address) or r-value (contents) is needed; usually it is r-value
  - Interior nodes are labeled by the operators
  - Nodes are optionally also given a sequence of identifiers (identifiers that are assigned their values)

- Useful for implementing transformations on and determining information about a basic block

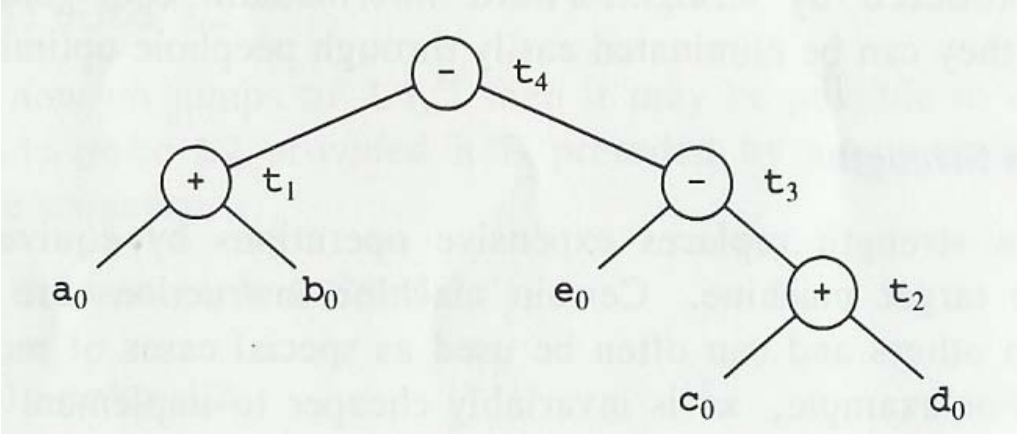
DAG Example



Using DAGS

- DAG can be automatically constructed from code using a simple algorithm
- Several useful pieces of information can be obtained:
  - Common sub-expressions
  - Which identifiers have their values used in the block?
  - Which statements compute values that could be used outside the block?
- Can be used to reconstruct a simplified list of quadruples
  - Can evaluate interior nodes of DAG in any order that is a topological sort (all children before parent)
  - Heuristics exist to find good orders
  - If DAG is a tree, a simple algorithm exists to give optimal order (order leading to shortest instruction sequence)

Generating Code from DAGS (1)




```
t1 := a + b
t2 := c + d
t3 := e – t2
t4 := t1 – t3
```

```
t2 := c + d
t3 := e – t2
t1 := a + b
t4 := t1 – t3
```

Generating Code from DAGS (2)

- Assume only t_4 is live on exit of previous example and two registers (R0 and R1) exist
- Code generation algorithm discussed earlier leads to following solutions (second saves two instructions)

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```