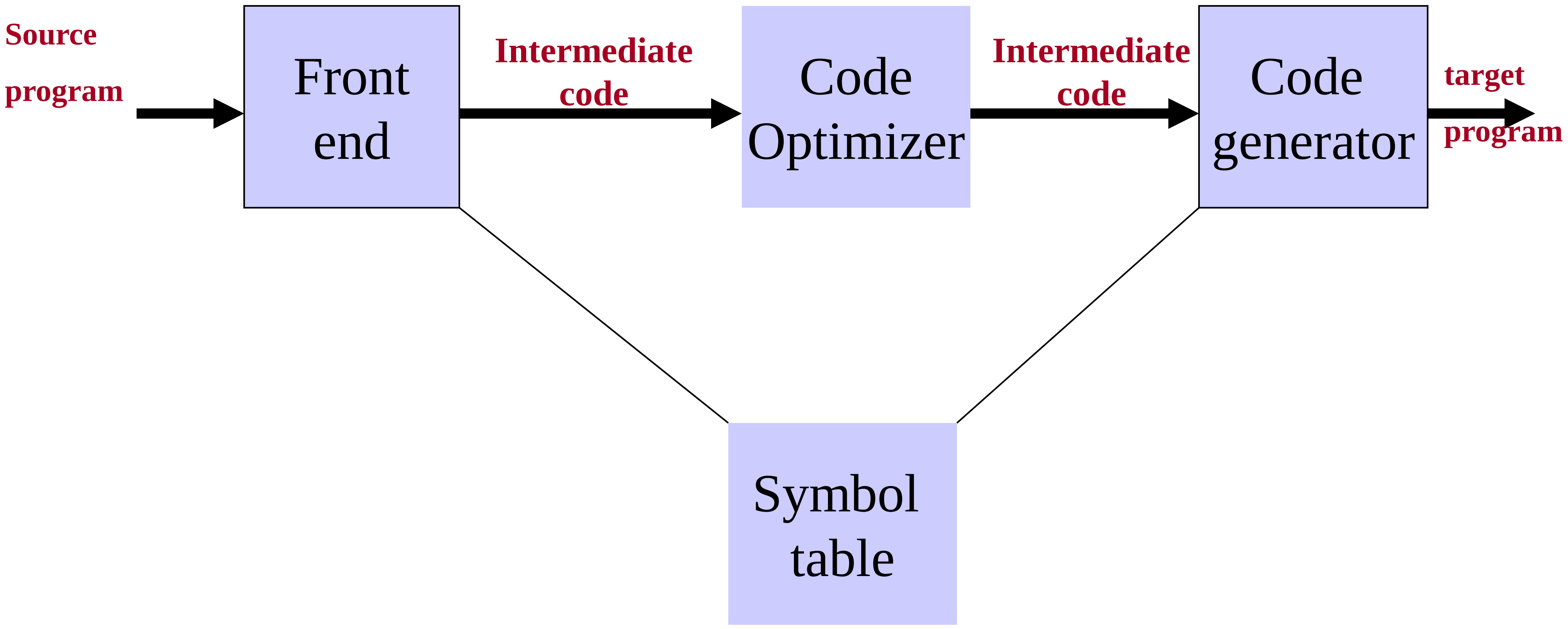
Code Generation

Introduction



Position of code generator

Issues in the Design of a Code Generator

- Input to the code generator
- Target language
- Memory management
- Instruction Selection
- Register allocation
- Evaluation order
- Approaches to code generation

Input to the Code Generator

- We assume, front end has
  - Scanned, parsed and translate the source program into a reasonably detailed intermediate representations
  - Type checking, type conversion and obvious semantic errors have already been detected
  - Symbol table is able to provide run-time address of the data objects
  - Intermediate representations may be
    - Postfix notations
    - Three address representations
    - Stack machine code
    - Syntax tree
    - DAG

Target Programs

- The output of the code generator is the target program.
- Target program may be
  - Absolute machine language
    - It can be placed in a fixed location of memory and immediately executed
  - Re-locatable machine language
    - Subprograms to be compiled separately
    - A set of re-locatable object modules can be linked together and loaded for execution by a linker
  - Assembly language
    - Easier

Memory Management

- Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.
- Symbol table entries were created as declaration in in a procedure were examined.
- From the symbol table information, a relative address can be determined for the name in a data area for the procedure.
- If machine code is being generated, labels in three address statement have to be converted to addresses of instructions.

Instruction Selection

- The nature of the instruction set of the target machine determines the difficulty of the instruction selection.
- Uniformity and completeness of the instruction set are important
- Instruction speeds is also important
  - Say, x = y + z

```
Mov y, R0
Add z, R0
Mov R0, x
```

Statement by statement code generation often produces poor code

- A target machine with a rich instruction set may provide several ways of implementing a given operation

Instruction Selection (2)

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

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

If a is subsequently used

Instruction Selection (3)

- The quality of the generated code is determined by its speed and size.
- Cost difference between the different implementation may be significant.
  - Say a = a + 1

    ```
    Mov a, R0
    Add #1, R0
    Mov R0, a
    ```

  - If the target machine has increment instruction (INC), we can write

    inc a

Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Efficient utilization of register is particularly important in code generation.
- The use of register is subdivided into two sub problems
  - During register allocation, we select the set of variables that will reside in register at a point in the program.
  - During a subsequent register allocation phase, we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single register value. Mathematically, the problem is NP – complete.

Choice of evaluation order

- The order in which computations are performed can affect the efficiency of the target code.
- Picking up the best order is another difficult, NP-complete problem.

Approaches to code generation

- Correctness – the most important criteria for code generation
- Given the premium on correctness, designing the code generator so it can be easily implemented, tested and maintained is an important design goal.
- Code generation algorithm
- The emphasis is on allocating register for heavily used operands in inner loop.

Run-Time storage management

- Here we discuss what code to generate to manage activation records at run time.
- Static allocation : Position of allocation record is fixed at compile time
- Stack allocation : A new activation record is pushed on to the stack for each execution of procedure. The record is popped when execution ends.
- Since run time allocation and deallocation of activation records occurs as a part of procedure call and return sequence, here we focus on following three-address statements.

1. call 2. return 3. halt 4. action, a placeholder for other statement

1. Static Allocation

2. Stack Allocation

Assignment: Explain with example Static allocation and Stack allocation

Basic Block

$t_1 = a*a$

$t_2 = a*b$

$t_3 = 2*t_2$

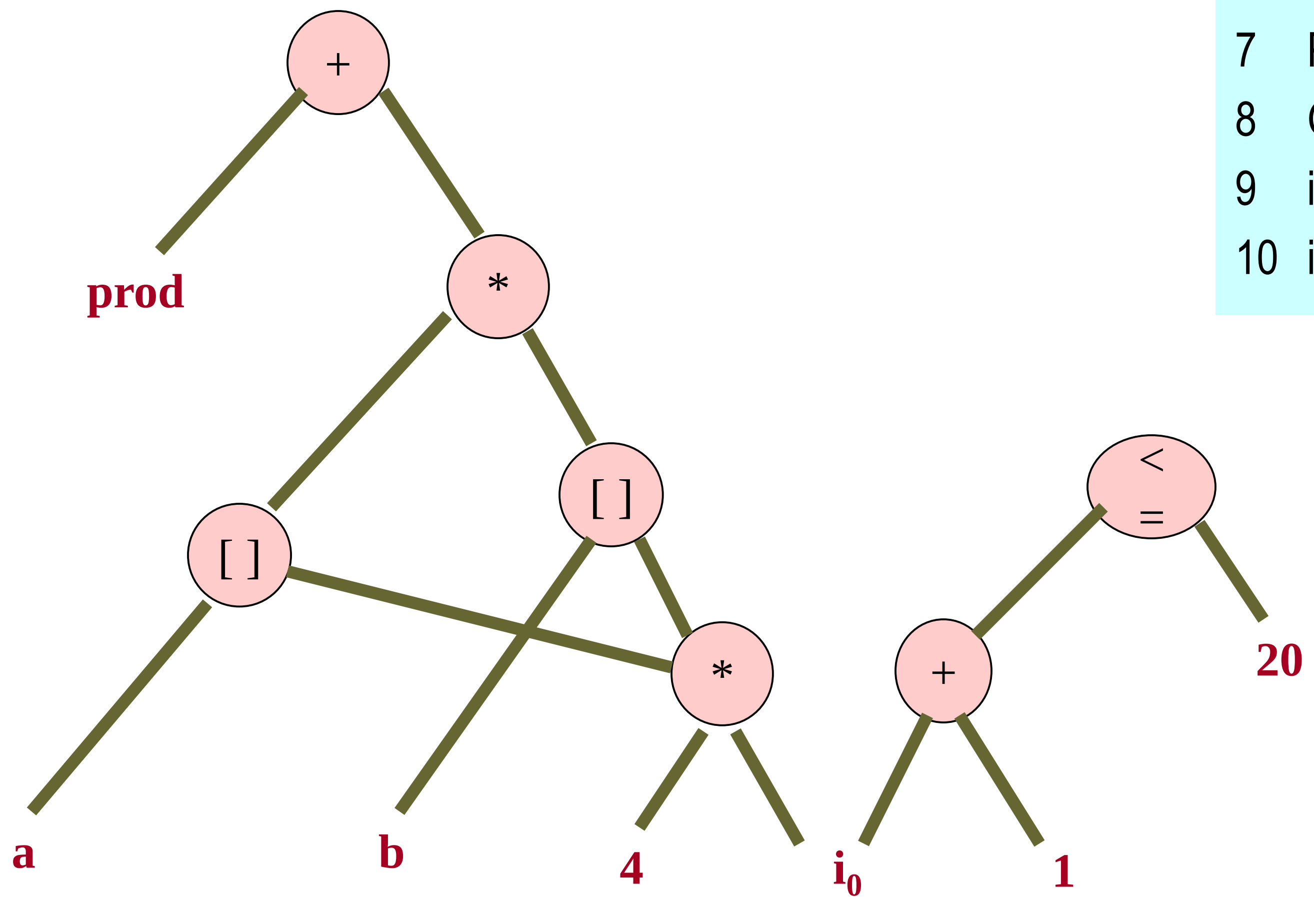
$t_4 = t_1+t_3$

$t_5 = b*b$

$t_6 = t_4+ t_5$

The DAG representation of Basic Block

1 A = 4*i
2 B = a[A]
3 C = 4*i
4 D = b[C]
5 E = B * D
6 F = prod + E
7 Prod = F
8 G = i + 1
9 i = G
10 if I <= 20 goto (1)



Basic Blocks and Flow Graphs

- A graph representation of three address statements, called flow graph.
- Nodes in the flow graph represent computations
- Edges represent the flow of control

Basic Block:

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibly of the branching except at the end.

Basic Blocks and Flow Graphs (2)

This is a basic block

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

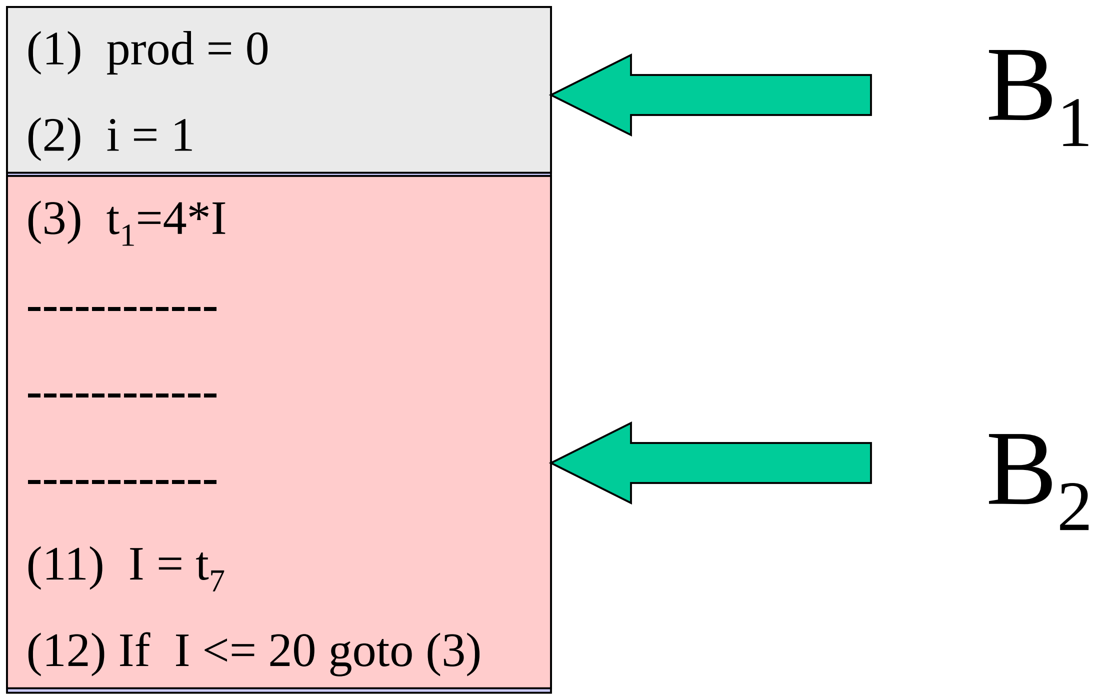
Three address statement $x = y + z$ is said to define x and to use y and z.

A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block

Basic Blocks and Flow Graphs (3)

- **Partition into basic blocks**
  - **Method**
    - **We first determine the leader**
      - **The first statement is a leader**
      - **Any statement that is the target of a conditional or unconditional goto is a leader**
      - **Any statement that immediately follows a goto or unconditional goto statement is a leader**
    - **For each leader, its basic block consists of the leader and all the statements up to but not including the next leader or the end of the program.**

Basic Blocks and Flow Graphs (4)



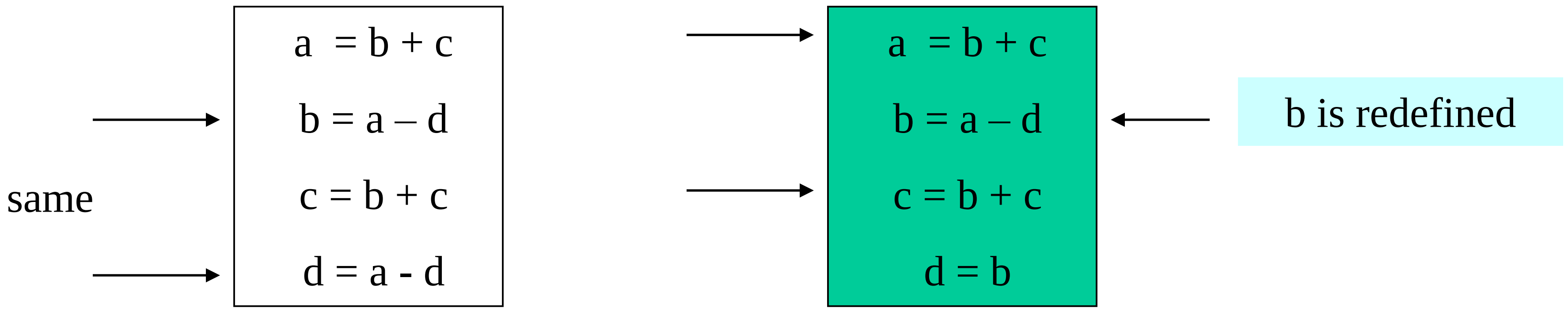
```
(1)  prod = 0
(2)  i = 1
(3)  t1=4*I
------------
------------
------------
(11)  I = t7
(12) If  I <= 20 goto (3)
```

Transformation on Basic Block

- A basic block computes a set of expressions.
- Transformations are useful for improving the quality of code.
- Two important classes of local optimizations that can be applied to a basic blocks
  - Structure Preserving Transformations
  - Algebraic Transformations

Structure Preserving Transformations

➢ <u>**Common sub-expression elimination**</u>



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

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

Structure Preserving Transformations

- **Dead – Code Elemination**

  Say, x is dead, that is never subsequently used, at the point where the statement x = y + z appears in a block.

  We can safely remove x

- **Renaming Temporary Variables**
  - say, t = b+c where t is a temporary var.
  - If we change u = b+c, then change all instances of t to u.
- **Interchange of Statements**
  - $t_1 = b + c$
  - $t_2 = x + y$
  - We can interchange iff neither x nor y is t_1 and neither b nor c is t_2

Algebraic Transformations

- Replace expensive expressions by cheaper one
  - X = X + 0 eliminate
  - X = X * 1 eliminate
  - X = y**2 (why expensive? Answer: Normally implemented by function call)
    - by X = y * y
- **Flow graph:**
  - We can add flow of control information to the set of basic blocks making up a program by constructing directed graph called flow graph.
  - There is a directed edge from block B_1 to block B_2 if
    - **There is conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or**
    - **B_2 is immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.**

Loops

- A loop is a collection of nodes in a flow graph such that
  - All nodes in the collection are strongly connected, that is from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
  - The collection of nodes has a unique entry, that is, a node in the loop such that, the only way to reach a node from a node out side the loop is to first go through the entry.

Exercise

- given the code fragment

```
x := a*a + 2*a*b + b*b;
y := a*a – 2*a*b + b*b;
```

draw the dependency graph before and after common subexpression elimination.

Answers

dependency graph before CSE

$$\begin{tikzcd}
 & & & & & \mathbf{x} \arrow[d] & & & \\
 & & & & & + \arrow[dll] \arrow[drr] & & & \\
 & & & + \arrow[dll] \arrow[dr] & & & & * \arrow[dl] \arrow[dr] & \\
 & * \arrow[dl] \arrow[dr] & & & * \arrow[dl] \arrow[dr] & & b & & b \\
 a & & a & * \arrow[dl] \arrow[dr] & & b & & & \\
 & & 2 & & a & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & & \mathbf{y} \arrow[d] & & & \\
 & & & & & + \arrow[dll] \arrow[drr] & & & \\
 & & & - \arrow[dll] \arrow[dr] & & & & * \arrow[dl] \arrow[dr] & \\
 & * \arrow[dl] \arrow[dr] & & & * \arrow[dl] \arrow[dr] & & b & & b \\
 a & & a & * \arrow[dl] \arrow[dr] & & b & & & \\
 & & 2 & & a & & & &
\end{tikzcd}$$

Answers

dependency graph after CSE

$$\begin{tikzcd}
 & & & \mathbf{x} \arrow[d] & & \\
 & & & + \arrow[dll] \arrow[dr] & & \\
 & + \arrow[dl] \arrow[dr] & & & * \arrow[d, bend right] \arrow[d, bend left] & \\
* \arrow[d, bend right] \arrow[d, bend left] & & * \arrow[d] \arrow[rr, bend right] & & b & \\
a & & * \arrow[dl] \arrow[ll, bend left=60] & & & \\
 & 2 & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & \mathbf{y} \arrow[d] & & & \\
 & & & & + \arrow[dl] \arrow[drr] & & & \\
 & & & - \arrow[dl] \arrow[d] & & & * \arrow[dl] \arrow[dr] & \\
 & * \arrow[dl] \arrow[dr] & & * \arrow[d] \arrow[dr] & & b & & b \\
a & & a & * \arrow[dl] \arrow[dr] & b & & & \\
 & & 2 & & a & & &
\end{tikzcd}$$

Answers

dependency graph after CSE

$$\begin{tikzcd}
 & & \mathbf{x} \arrow[d] & & & \mathbf{y} \arrow[d] \\
 & & + \arrow[dl] \arrow[dr] & & & + \arrow[dl] \arrow[dll, bend left] \\
 & + \arrow[dl] \arrow[dr] & & * \arrow[dr, bend right] \arrow[dr, bend left] & - \arrow[dllll, bend left] \arrow[dll, bend left] & \\
 * \arrow[d, bend right] \arrow[d, bend left] & & * \arrow[rr, bend right] \arrow[dl] & & b & \\
 a & * \arrow[d] \arrow[l, bend left] & & & & \\
 & 2 & & & &
\end{tikzcd}$$

Next-Use information

- We collect next-use information about names in basic blocks. If the name in register is no longer needed, then the register can be assigned to some other names.
- We wish to determine for each three address statement

  x := y op Z

That the next use of x, y and z.

After finding end of basic block we scan backwards to the beginning, recording (in the symbol table) for each name x whether x has next use in the block and if not, whether it is live on exit from the block

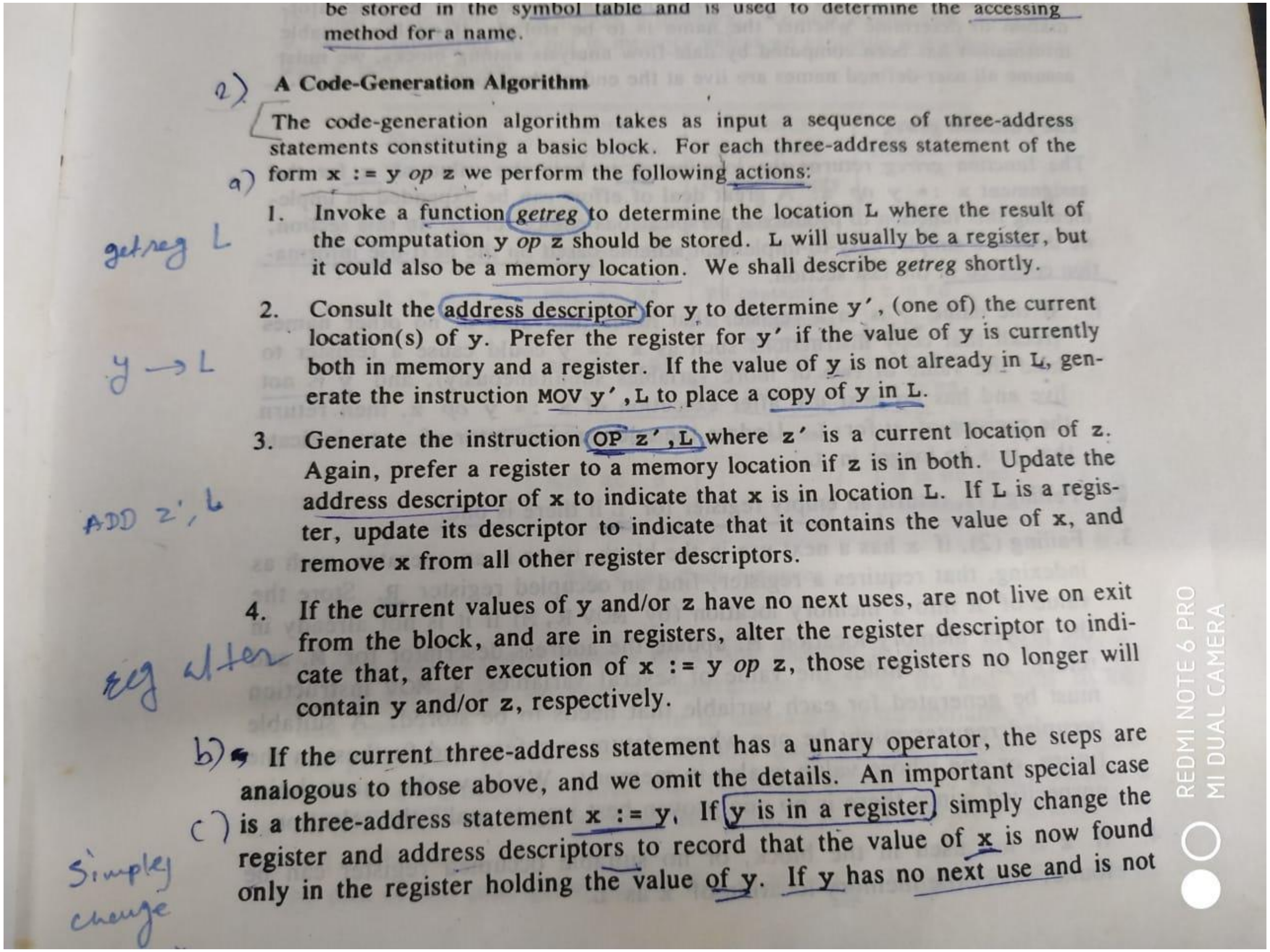
If no live-variable analysis has been done, we can assume all non-temporary variables are live on exit.

Suppose we reach three address statement i: x := y op z in our backward scan we then do the following

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y and z
2. In the symbol table, set x to non-live and ‘no next-use’.
3. In the symbol table, set y and z to “live” and the next-use of y and z to i.

- **Storage for Temporary Names** it may be useful in an optimizing compiler to create distinct name each time a temporary needed. The size of the temporaries in the activation record grow with the number of temporaries.
- In general, pack two temporaries in to a same location if they are not live simultaneously. Since almost all temporaries are defined and used within basic blocks, next use information can be applied to pack temporaries.

Code Generation Algorithm



be stored in the symbol table and is used to determine the accessing method for a name.

a) **A Code-Generation Algorithm**

The code-generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form **x** := **y** *op* **z** we perform the following actions:

a)

1. Invoke a function *getreg* to determine the location L where the result of the computation **y** *op* **z** should be stored. L will usually be a register, but it could also be a memory location. We shall describe *getreg* shortly.

getreg L

2. Consult the address descriptor for **y** to determine **y**′, (one of) the current location(s) of **y**. Prefer the register for **y**′ if the value of **y** is currently both in memory and a register. If the value of **y** is not already in L, generate the instruction MOV **y**′,L to place a copy of **y** in L.

y → L

3. Generate the instruction OP **z**′,L where **z**′ is a current location of **z**. Again, prefer a register to a memory location if **z** is in both. Update the address descriptor of **x** to indicate that **x** is in location L. If L is a register, update its descriptor to indicate that it contains the value of **x**, and remove **x** from all other register descriptors.

ADD z′, L

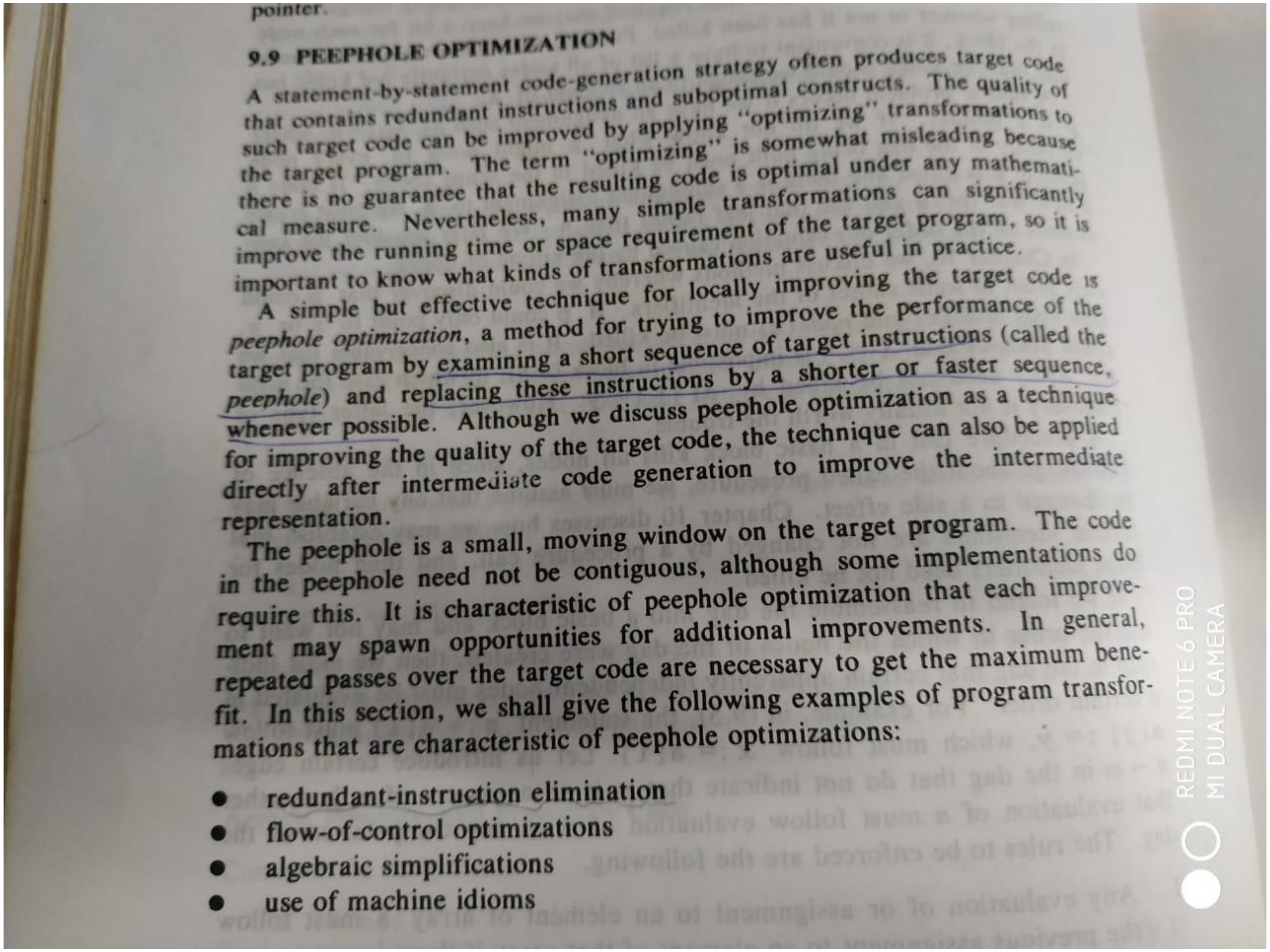
4. If the current values of **y** and/or **z** have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of **x** := **y** *op* **z**, those registers no longer will contain **y** and/or **z**, respectively.

reg alter

b) If the current three-address statement has a unary operator, the steps are analogous to those above, and we omit the details. An important special case c) is a three-address statement **x** := **y**. If **y** is in a register, simply change the register and address descriptors to record that the value of **x** is now found only in the register holding the value of **y**. If **y** has no next use and is not

Simply change

Peephole Optimization



pointer.

9.9 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program, so it is important to know what kinds of transformations are useful in practice.

A simple but effective technique for locally improving the target code is *peephole optimization*, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the *peephole*) and replacing these instructions by a shorter or faster sequence, whenever possible. Although we discuss peephole optimization as a technique for improving the quality of the target code, the technique can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit. In this section, we shall give the following examples of program transformations that are characteristic of peephole optimizations:

- redundant-instruction elimination
- flow-of-control optimizations
- algebraic simplifications
- use of machine idioms

Compiler Construction Tools

- Parser generators : Produces Syntax Analyzers from input that is based on a context free grammar
- Scanner generators : Automatically generates lexical analyzer normally from a specification based on regular expressions
- Syntax-directed translation engine : Produces collection of routines that walk the parse tree
- Automatic code generators : Takes a collection of rules that define the translation of each operation of the intermediate language in to the machine language for target machine.
- Data flow engine : Needed for code optimization, the gathering of information about how values of transmitted from one part of the program to each other

Assignment

1. What is Just In Time Compiler
2. Explain the function *getreg*

The End