5th UNIT

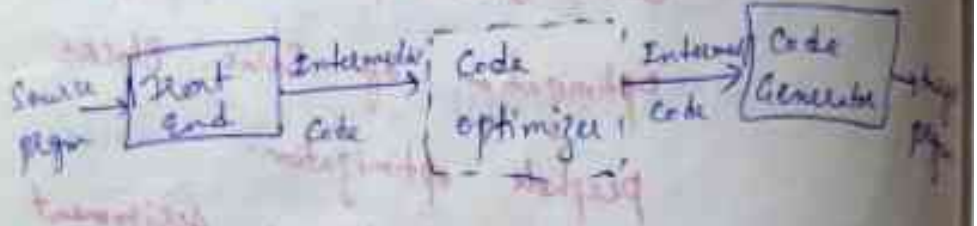
CODE GENERATION

→ A code generator has three primary tasks

Instruction Selection

Register Allocation & Assignment

Instruction Ordering



Issues In The Design of a Code Generator

→ The most important criterion for a code generator is that it produces correct code.

→ Issues are

I/p to the code generator

The target pgm

Instruction Selection

Register Allocation

Evaluation Order

I/p to the Code Generator :-

→ It is the intermediate repr of the source prgm produced by the front end, along with inf from the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

→ The many choices for the IR include

3-addr repr such as Quadruples,
triples,
Indirect triples

→ Virtual machine repr such as bytecodes & stack-machine code.

→ linear repr such as postfix notation & graphical repr such as Syntax trees & DAG's.

Target program :-

→ The instr-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.

→ the most common target-machine architectures are RISC (Reduced Inst set Computer)
CISC (Complex Inst set Computer)
Stack Based.

RISC	CISC
→ has many registers	→ few registers
→ 3-addr Inst	→ 2-addr Inst
→ Simple Inst-set architecture	→ variety of addressing modes

→ Stack based machines almost disappeared because it was felt that the stack organization was too limiting & required too many swap & copy operations.

Stack-based architectures were revived with the intro of JVM.

Instruction Selection :-

→ The code generator must map the IR prgm into a code sequence that can be executed by the target machine.

→ the Complexity of performing this mapping is determined by factors such as:

- The level of the IR
- The nature of the Instruction-Set architecture
- The desired quality of the generated code.

Ex:- $x = y + z$

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

Register Allocation :-

→ A key problem in code generation is deciding what values to hold in what registers.

→ The use of registers is often subdivided into two subproblems:

- Register Allocation, during which we select the set of variables that will reside in registers at each point in the program.
- Register Assignment, during which we pick the specific register that a variable will reside in.

Translate code for Assignment & Allocation

```
t = a + b
t = t * c
t = t / d
```

```
t = a + b
t = t + c
t = t / d
```

```
L   R1, a
A   R1, b
M   R1, c
D   R1, d
ST  R1, t
```

Register Allocation

```
L   R0, a
A   R0, b
A   R0, c
SRDA R0, 32
D   R0, d
ST  R1, t
```

Register Assignment

SRDA - Shift - Right - Double - Arithmetic

Evaluation Order

The order in which computations are performed can affect the efficiency of the target code.

Some computations orders require fewer registers to hold intermediate results than others.

TARGET LANGUAGE

target lang Model
prgm & inst costs

Target language Model

→ our target computer models a 3-addr machine with load & store operations, computation operations, jump operations, & conditional jump

→ The available inst are:

Load operations

LD r1, r2

Store operations

ST x, r

Computation operations

OP dst, src1, src2

Unconditional jumps

BR L

Conditional jumps

BR cond r, L

r - register
L - label

Eg:- $x = *p$

```
LD   R1, P
LD   R2, 0(R1)
ST   x, R2
```

$*p = y$

```
LD   R1, P
LD   R2, y
ST   0(R1), R2
```

⇒ Program & Instruction Costs

- The Inst LD R0, R1 copies the contents of register R1 into register R0.

  Cost is 1 coz no additional memory words are required.

- The Inst LD R0, M loads the contents of memory location M into register R0.

  Cost is 2 coz addr of memory location M is in the word.

- The Inst LD R1, *100(R2) loads into register R1.

  The Cost is 2 because the constant 100 is stored in the word.

Addresses in the Target Code

→ Each Executing pgm runs in its own logical address space that was partitioned into four code & data areas:

Code Area
Static Area
Heap Area
Stack Area

1. A statically determined area Code that holds the executable target code. The size of the target code can be determined at compile time.
2. A statically determined data area Static for holding Global Constant and other data generated by the compiler. The size of the global constant can be determined at compile time.
3. A dynamically managed area Heap for holding data objects that are allocated & freed during pgm execution. The size of the heap determined at Run time.
4. Area Stack for holding Activation Records as they are created & destroyed during procedure calls & returns. determined at Run time.

⇒ Basic Blocks & Flow Graphs

Basic Blocks

→ our first job is to partition a sequence of 3-addr inst into basic blocks.

→ we begin a new basic blocks with the first inst & keep adding inst until we meet either a jump, a conditional jump, or a label on the following instruction.

→ In the absence of jumps & labels, control proceeds sequentially from one instruction to the next.

Method

→ First, we determine those inst in the intermediate code that are LEADERS, that is, the first inst in some basic blocks.

Rules for finding leaders are:

1. The first 3-addr inst in the I.C is a leader.
2. Any inst that is the target of a conditional or unconditional jump is a leader.
3. Any inst that immediately follows a conditional or unconditional jump is a leader.

Rem: for each leader, its basic block consists of itself and all inst upto but not including the next leader or the end of the intermediate program.

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

→ The leaders are inst 1, 2, 3, 10, 12, & 13.
→ the basic block of each leader contains all the inst from itself until just before the next leader.

Flow Graphs

→ once an I.c prgm is partitioned into basic blocks, we represent the flow of ctl b/w them by a flow graph.

→ The nodes of the flow graph are the basic blocks.

→ There is an edge from block B to block C if & only if it is possible for the first inst in block C to immediately follow the last inst in block B.

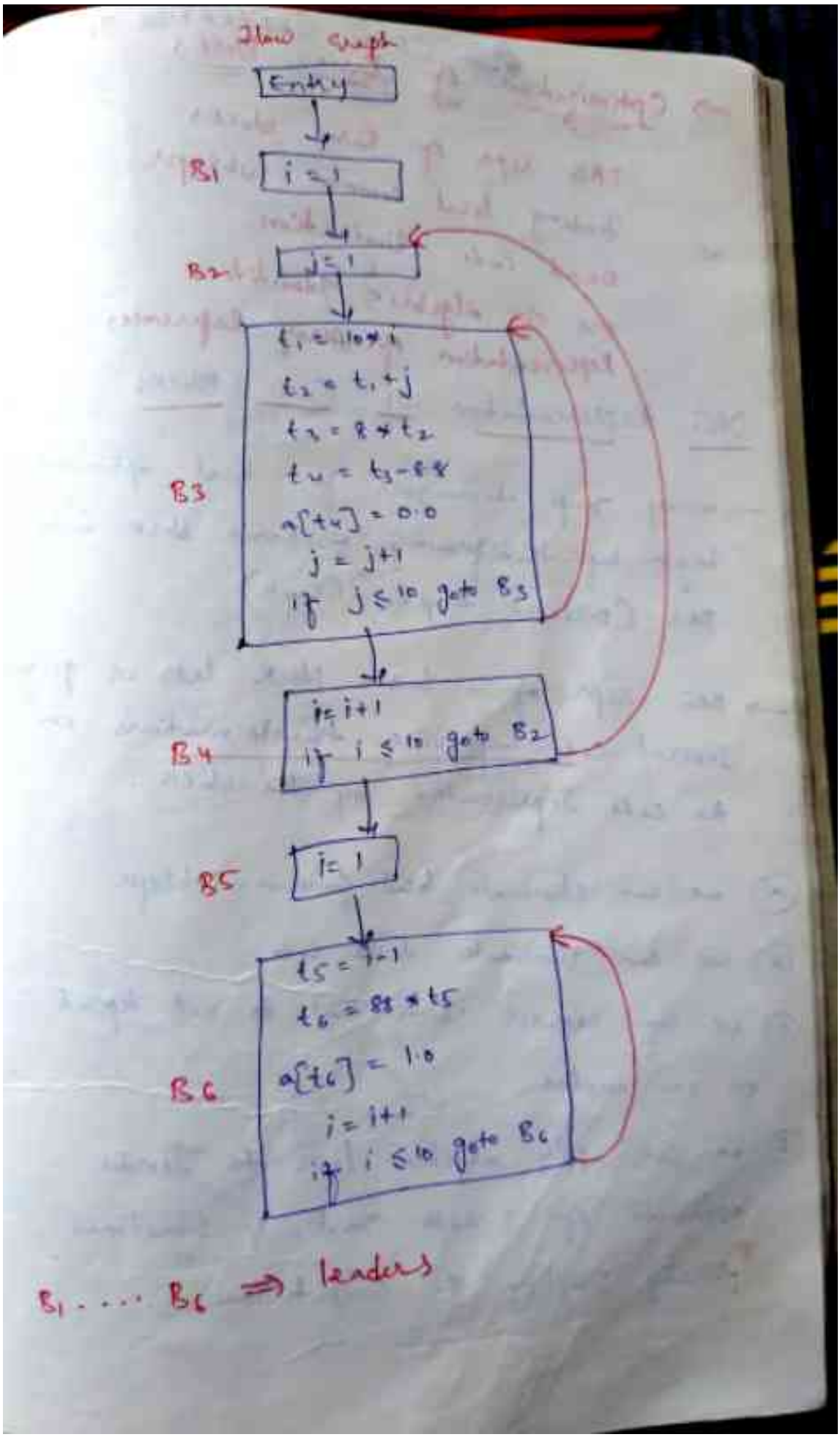
→ Also we added two nodes, ENTRY & EXIT

Loops

→ Prgming language constructs like while-stmt, do-while stmt & for-stmt naturally give rise to loops in prgms.

→ The flow graph has three loops.

1. B_3 by itself
2. B_6 by itself
3. {B_2, B_3, B_4}


Flow Graph
Entry
B1
i = 1
B2
j = 1
B3
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j ≤ 10 goto B3
B4
i = i + 1
if i ≤ 10 goto B2
B5
i = 1
B6
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i ≤ 10 goto B6
B1 . . . B6 ⇒ leaders

CODE GENERATION TECHNIQUES

⇒ Optimization of Basic Blocks

DAG repr of Basic blocks
finding local common subexpr
dead code elimination
use of algebraic identities
Representation of Array References

DAG Representation of Basic Blocks

→ many imp techniques for local optimization begin by transforming a basic block into DAG. (Directed Acyclic Graph)

→ DAG repr of a basic block lets us perform several code improving transformations on the code represented by the block.

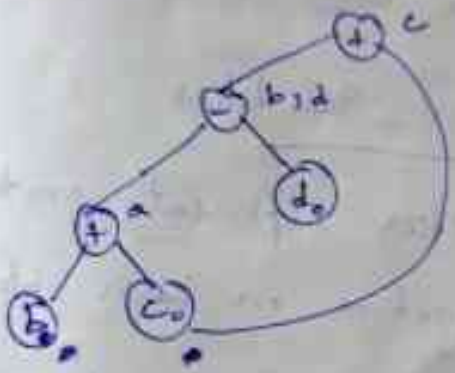
(a) we can eliminate local common subexpr
(b) we can eliminate dead code
(c) we can reorder stmts that do not depend on one another
(d) we can apply algebraic laws to reorder operands of 3-addr inst., & sometimes thereby simplify the computation.

Finding Local Common Subexpr

DAG for block

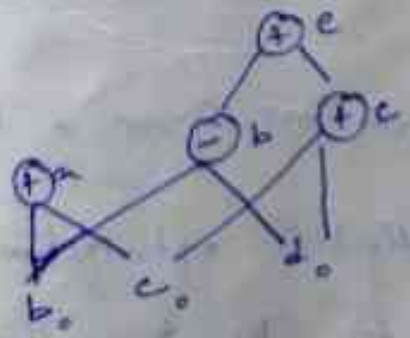
a = b+c
b = a-d
c = b+c
d = a-d

b+c is repeating



Finding Local Common Subexpr

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



Dead Code Elimination

→ We delete from a DAG any root (node with no ancestors) that has no live variables attached.

→ Repeated application of the transformation will remove all the nodes from DAG that correspond to dead code.

Use of Algebric Identities

→ Algebric identities represent another important class of optimizations on basic blocks.

→ For eg, we may apply arithmetic identities, such as

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

to eliminate computations from a basic block

→ Another class of Algebric optimizations includes LOCAL REDUCTION in Strength, that is, replacing a more expensive operator by a cheaper one as in:

Expensive	Cheaper
x^2	$x * x$
$2 * x$	$x + x$
$x/2$	$x * 0.5$

→ A third class of related optimizations is CONSTANT FOLDING

Here, we evaluate constant Expr at compile time and replace the constant Expr by their values.

Ex:- Expr $2 * 3.14$ would be replaced by 6.28

Representation of Array References

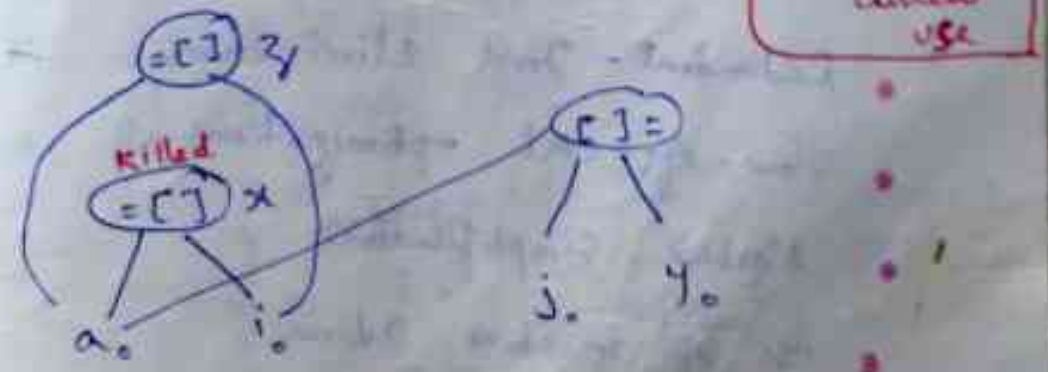
DAG for basic block

$x = a[i]$

$a[j] = y$

$z = a[i]$

Kill = not in current use



⇒ PEEPHOLE OPTIMIZATION

→ Compilers generate naive code & then improve the quality of the target code by applying "optimizing" transformations to the target prgm.

→ A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called peephole) & replacing inst sequences within the peephole by a shorter or faster sequence, whenever possible.

→ peephole optimization can also be applied directly after intermediate code generation to improve the IR (Intermediate Repr).

→ Characteristics of peephole optimizations:

- Redundant - inst elimination
- Flow-of-ctrl optimizations
- Algebric simplifications
- use of machine idioms

Eliminating Redundant loads & stores

```
LD R0, a
ST a, R0
```

→ In the target prgm, we can delete the store Inst because whenever it is executed, the first Inst will ensure that the value of 'a' has already been loaded into Register 'R0'.

→ Note that if the Inst had a Label, we could not be sure that the first Inst is always executed before the 2nd, so we could not remove the store Inst.

Eliminating Unreachable code

→ An Unlabeled Inst immediately following an Unconditional jump may be removed.

→ This operation can be repeated to eliminate a sequence of Inst.

Eg:-
```
    if debug == 1 goto L1
    goto L2          — Unreachable
L1: print debugging inf
L2:
```

Flow-of-control optimizations

→ Simple intermediate code-generation algo frequently produce jumps to jumps, jumps to cond^n jump, cond jumps to jmp

→ These unnecessary jumps can be eliminated in either the I.C or the target code

Eg:- goto L_1 ≡ L_1: goto L_2 } replaced by → goto L_2

Uncond^n jmp

if a<b goto L_1 → if a<b goto L_2

L_1: goto L_2

Cond^n jmp

Algebric Simplification & Reduction in Strength

→ Algebric identities can also be used by a peephole optimizer to eliminate 3-addr stmt such as

$x = x + 0$

or

$x = x * 1$

→ Reduction in Strength

Replacing Expensive Expr by cheaper Expr

x^2 Expensive replaced by $x+x$ cheaper

Use of Machine Idioms

→ the target machine may have H/w inst to implement certain specific operations efficiently.

→ Detecting situations that permit the use of these inst can reduce execution time significantly.

→ for eg, Some machines have Auto-increment & Auto-decrement addressing modes.

⇒ Register Allocation & Assignment

→ Inst involving only register operands are faster than those involving memory operands

→ Efficient utilization of registers is vitally imp in generating good code.

- Global Register Allocation
- Usage Counts
- Register Assignment for outer loops
- Register Allocation by Graph Coloring

Global Register Allocation

→ The code generation algo used registers to hold values for the duration of a single basic blocks.

→ However, all live variables were stored at the end of each block.

→ to save some of these stores & corresponding loads, we might arrange to assign registers to frequently used variables & keep these registers consistent across block boundaries (GLOBALLY)

Usage Counts

$$\sum_{\text{block } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

Block B in loop L

use(x, B) :- no of times x is used in Block B prior to any def

live(x, B) :- is 1 if x is live on exit from B & is assigned a value in B.

0 if x is dead on exit

2 :- Constant value.

Register Assignment for outer loops

→ Since programs spend most of their time in inner loops,

→ if an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocated registers $L_1 - L_2$.

↳ However, if we choose to allocate x a register in L_2 but not L_1, we must load x on Entrance to L_2 & store x on exit from

Register Allocation by Graph Coloring

→ when a register is needed for a computation but all Available registers are in use,

then the contents of one of the used registers must be stored (SPILLED) into a memory location in order to free up a register.

→ Graph coloring is a simple, systematic technique for allocating registers & managing register spills.

→ An attempt is made to color the register-interference graph using K colors, where K is the no of Assignable registers

→ A Graph is said to be COLORED if each node has been assigned a color in such a way that no two adjacent nodes have the same color.

→ A color represents a register, and the color makes sure that no two symbolic registers that can interfere with Eachother are assigned the same physical register.

MACHINE - INDEPENDENT OPTIMIZATIONS

Principal Sources of optimization

A Compiler optimization must preserve the semantics of the original program.

Causes of Redundancy

Semantics - preserving transformations

Global Common subexpressions

Copy propagation

Dead-code Elimination

Code Motion

Induction Variables

Reduction in strength

Causes of Redundancy

→ Accesses to the same data structure often share many common low-level operations.

→ programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves.

→ It is, in fact, preferable from a s/w-Engineering perspective that programmers only access data elements by their

high-level names; the prgms are easier to write & more importantly, easier to understand and evolve.

→ By having a compiler eliminate the redundancies, we get the best of both worlds;
the prgm are both efficient & easy to maintain.

⇒ Semantics-Preserving Transformations

→ There are a no. of ways in which a compiler can improve a prgm w/o changing the fn it computes.

→ Common-SubExpression elimination
Copy propagation
dead-code elimination &
Constant folding

Before

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

After

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

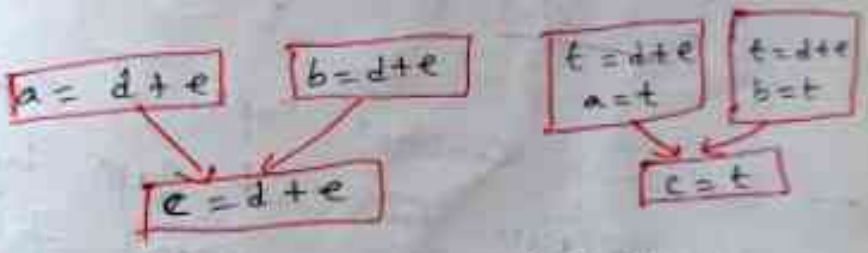
local Common-Subexpr Elimination

Global - Common Subexpressions

the Expr which is a Common in more than one Block has to be Eliminated.

copy propagation

→ one Concern assignments of the form U=V Called COPY STATEMENTS or COPIES for short.



copies introduced during Common subexpr elimination

Dead Code Elimination

→ A Variable is LIVE at a point in a prgm if its value can be used subsequently; otherwise, it is DEAD at that point.

→ A related idea is DEAD (or useless) Code - Stmts that compute values that never get used.

→ While the pgmer is unlikely to introduce any dead code intentially, it may appear as the result of previous transformations.

Eg:-
```
debug = FALSE
if (debug)
    print: ≡
```

Note:- if debug T then only print stmt will get executed, but debug=false . So print stmt never gets executed.

→ Evaluating Constant Expr at compile time and replace the Const Expr by their values is called CONSTANT FOLDING

⇒ Code Motion

→ loops are a very imp place for optimization. Especially the inner loops where pgmrs tends to spend the bulk of their time.

→ The running time of a pgm may be improved if we decrease the no. of inst in an inner loop, even if we increase the amount of code outside that loop.

→ Imp modification that decreases the amount of code in a loop is CODE MOTION.

→ this transformation takes an expr that yields the same result independent of the no of times a loop is executed (loop-invariant computation) & evaluates the expr before the loop

→ Induction Variables & Reduction in Strength

→ Another imp optimization is to find induction variables in loops and optimize their computation.

→ A variable x is said to be an INDUCTION VARIABLE if there is a positive or negative constant c such that each time x is assigned, its value increases by c.

→ Induction variables can be computed with a single increment (add/sub) per loop iteration.

→ the transformation of replacing an expensive operation, such as mul, by a cheaper one, such as add, is STRENGTH REDUCTION.

⇒ INTRODUCTION TO DATA-FLOW ANALYSIS

→ Data-flow analysis refers to a body of techniques that derive inf about the flow of data along prgm execution paths.

DATA-Flow Abstraction
Data-flow Analysis schema
Data-flow schemas on Basic blocks
Reaching Definations
live-Variable Analysis
Available Expressions

⇒ Data-Flow Abstraction

→ Each execution of an intermediate-code stmt transforms an i/p state to a new o/p state.

→ the i/p is associated with the prgm point before the stmt and the o/p state is associated with the prgm point after the stmt.

→ When we analyze the behavior of a prgm, we must consider all the possible sequence of prgm points ("paths") through

flow graph that the program execution can take.

→ we often Extract, from the possible program states at each point, the info we need for the particular data-flow analysis prob we want to solve.

→ In more Complex analyses, we must consider paths that jump among the flow graphs for various procedures, as calls & returns are Executed.

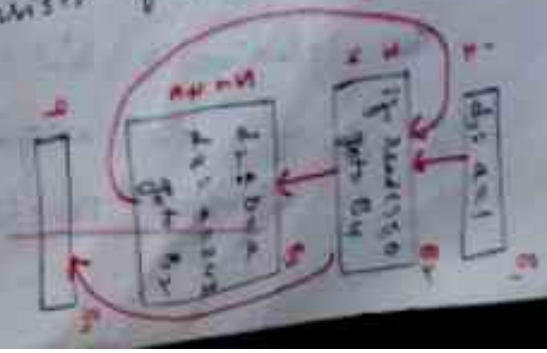
What the flow Graph tells about the possible Execution paths.

→ Within one basic block, the prgm point after a stmt is the same as the prgm point before the next stmt.

→ If there is an edge from block B_1 to block B_2, then the prgm point after the last stmt of B_1 may be followed immediately by the prgm point before the first stmt of B_2.

→ Not Entering the loop at all, the shortest complete Execution path consists of the prgm points (1, 2, 3, 4, 9)

the longest path is (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)



→ Data-flow Analysis Schema

→ In Each appli" of data-flow Analysis, we associate with every pgm point a data-flow value that represents an abstraction of the set of all possible pgm states that can be observed for that point.

→ The set of possible data-flow values is the domain for this appli".

→ the data-flow values before & after each Stmt S by IN[S] & OUT[S] respectively.

→ The data-flow problem is to find a Sol" to a set of constraints on the IN[S] & OUT[S] for all stmts S.

→ There are two sets of constraints:
those based on the semantics of the Stmt TRASFER FUNCTIONS &
flow of ctrl

→ Transfer Functions

The Relationship blw the data-flow values before & after the assignment Stmt is known as a Transfer Function.

Transfer fns come in two flavors:
info may propagate FORWARD along execution paths, or it may flow BACKWARDS up the execution paths.

→ In a forward-flow prob, the transfer fn of a stmt S, which we usually denote f_s, takes the data-flow value before the stmt & produces a new data-flow value after the stmt.

that is,

$$OUT[S] = f_s(IN[S])$$

→ In a backward-flow prob, the f_s for stmt S converts a data-flow value after the stmt to a new data-flow value before the stmt. that is,

$$IN[S] = f_s(OUT[S])$$

→ Control-flow Constraints

If a block B consists of stmts $S_1, S_2, \dots S_n$ in that order, then the ctrl-flow value out of S_i is the same as the ctrl-flow value into S_{i+1}. That is,

$$IN[S_{i+1}] = OUT[S_i] \text{ for all } i = 1, 2, \dots n-1$$

→ ctrl flow edges b/w basic blocks create more complex constraints b/w the last stmt of one basic block and the first stmt of the following block.

→ Data-flow schemas on Basic Blocks

→ The transfer fn of a basic block B, which we denote f_B

$$OUT[B] = f_B(IN[B])$$

Forward-flow problem in which

$$IN[B] = \bigcup_P \text{ a predecessor of } B \; OUT[P]$$

→ data-flow is backwards

$$IN[B] = f_B(OUT[B])$$

$$OUT[B] = \bigcup_S \text{ a successor of } B \; IN[S]$$

→ Reaching Definitions

→ Reaching definations is one of the most common & useful data-flow schemas

→ By knowing where in a prgm each var x may have been defined when ctrl reaches each point P, we can determine many things about x.

Eg: a Compiler then knows whether x is a constant at point P, and a debugger can tell whether it is possible for x to be an Undefined Var, should x be used at P.

Live - Variable Analysis

→ In Live - Variable Analysis we wish to know for variable x and point P whether the value of x at P could be used along some path in the flow graph starting at P. If so, we say x is live at P; otherwise, x is dead at P.

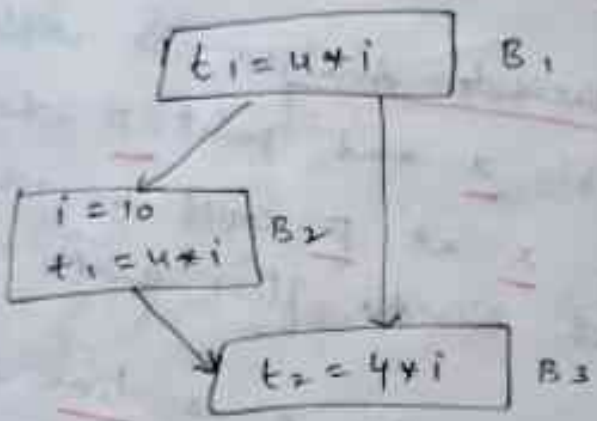
→ Define

1. def B as the set of var defined (definitely assigned values) in B prior to any use of that var in B, and
2. use B as the set of var whose values may be used in B prior to any defination of the var.

Available Expressions

→ An Expr $x+y$ is available at a point P if every path from the entry node to P evaluates $x+y$

→ The primary use of available-expr info is for detecting Global Common Subexpr.



→ 4*i is Available in all Blocks.

FOUNDATION OF DATA-FLOW ANALYSIS

The data-flow analysis framework (D, V, ∧, F) consists of

→ A Direction of the data flow D, which is Either FORWARDS or BACKWARDS

→ A Semilattice, which includes a Domain of Values V and a meet operator ∧.

→ A family F of transfer fns from V to V. this family must include fns suitable for the boundary conditions, which are constant transfer fns for the special nodes ENTRY & EXIT in any flow graph.

Semilattices

A semilattice is a set V and a binary meet operator ∧ such that for all x, y, z in V:

1. $x \wedge x = x$ (meet is idempotent)
2. $x \wedge y = y \wedge x$ (meet is commutative)
3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (meet is associative)

A Semilattice has a top element, denoted $\top$, such that

for all x in V, $\top \wedge x = x$

optionally, a Semilattice may have a bottom element, denoted $\bot$, such that

for all x in V, $\bot \wedge x = \bot$

PDF Created Using

Camera Scanner

Easily Scan documents & Generate PDF

https://play.google.com/store/apps/details?id=photo.pdf.maker