CODE OPTIMISATION

MOTIVATION :

To produce target programs with high execution efficiency.

Constraints :

(a) **Maintain semantic equivalence with source program.**

(b) **Improve a program without changing the algorithm.**

Need :

(a) **'Permissive' programming languages provide many flexibilities, often leading to inefficient coding. For example, $a := a + 1$; where a is 'real' requires a type conversion of '1' to '1.0'. An optimising compiler can avoid type conversion during the execution of the program by using the constant '1.0' instead of '1'.**

(b) **Due to the increasing cost of programmer time, programmers do not pay sufficient attention to execution efficiency of programs. Hence the need for optimisation during compilation.**

CODE OPTIMISATION

EFFECTIVENESS :

A very old result

When optimised by the IBM/360 Fortran H compiler (of mid-sixties vintage),

- **a program executes 3 times faster**
- **a program occupies 25% less storage**

Contemporary scenario

Improvements due to optimistion may be more dramatic in contemporary compilers because

- **More effective optimisation techniques are available.**
- **Today, less emphasis is put on execution efficiency than on other attributes of a program like structure, maintainability, reusability of a program, etc. Code sharing and re-use leads to a 'black box' view of programs, which further deemphasises execution efficiency.**
- **Exploitation of advanced architectural features like instruction pipelining requires 'smart' code generation, which is only possible in an optimising compiler.**

CODE OPTIMISATION

Q : **Can a programmer out-perform an optimiser ?**

YES ! This is possible by

(i) **choosing a better algorithm.**

(ii) **knowing more about the definitions and uses of data items in the program e.g. a programmer may know relative probabilities of branches being taken. Also,**

(a) **An optimiser has to ensure correctness of the optimised program under all conditions, hence it has to be** *conservative*

(b) **An optimiser may miss optimisation opportunities because of this.**

Also, NO ! Because

(i) **It takes too much time to perform some optimisations by hand, viz. strength reduction, copy propagation, dead code elimination.**

(ii) **Certain machine level details are beyond the control of a programmer, viz. instructions and addressing modes supported by the target machine.**

CODE OPTIMISATION

LEVELS OF OPTIMISATION

(a) Machine dependent optimisations, e.g.

1. better choice of instructions,
   e.g. INC instead of a load-add-store sequence
2. better use of addressing modes,
   e.g. base-displacement-offset addressing, etc.
3. better use of machine registers.

(b) Machine independent optimisations :

Based on semantics preserving transformations applied independent of the target machine, e.g. common sub-expression elimination, loop optimisation, etc.

CODE OPTIMISATION

MACHINE INDEPENDENT OPTIMISATION

Cost-effectiveness of machine independent transformations depends on their *scope*.

(a) Local Optimisation

Scope is restricted to *essentially sequential* sections of program code (called *basic blocks* – defined later). This restricts the amount of analysis necessary.

It also restricts

- **the kinds of optimisation feasible, and**
- **the gains of optimisation.**

e.g. loop optimisation can not be performed locally.

(b) Global Optimisation

Global optimisation is applied to a larger section of a program than a basic block, typically a loop or a procedure/function.

Knuth (1971) reports speed-up factors of

- $\geq$ **1.4 due to local optimisation**
- $\geq$ **2.7 due to global optimisation**

CODE OPTIMISATION

Why separate Local and Global optimisation ?

(a) **Local optimisation simplifies global optimisation :**

Consider elimination of redundant computations within a *basic block* **of code**

$$
\begin{array}{lcl}
a * b & & \\
\ \cdots & & \\
a * b & \Leftarrow & \text{assumed eliminated} \\
 & & \text{by local optimisation}
\end{array}
$$

Thus, global optimisation only needs to consider the first occurrence of $a * b$ **within a basic block.**

(b) **Local optimisation can be merged with the preparatory phase of global optimisation**

Common sub-expressions, constant propagation, etc. can be performed while converting a program to triples or quadruples.

OPTIMISING TRANSFORMATIONS

1. COMPILE-TIME EVALUATION

Shifting execution time actions to compilation time, such that they are not performed (repeatedly) during the execution of the program.

(a) Folding

Evaluation of an expression with constant operands at compilation time. In effect, an expression is replaced by a single value (hence the term 'folding').

Example :

$$area = (22.0/7.0)*r**2$$

22.0/7.0 can be performed during compilation itself.

Note **:**

Typical applications of folding are in address calculation for array references, where products of many constants are 'folded' into single constant values.

OPTIMISING TRANSFORMATIONS

1. COMPILE-TIME EVALUATION (Contd.)

(b) Constant Propagation

Propagation **implies replacement of a variable v by an entity appearing on the *rhs* of an assignment to v. Constant propagation is applied when v is assigned the value of a constant. This enhances the scope of optimisation by folding.**

Example :

$a := 3.1;$

$\dots$

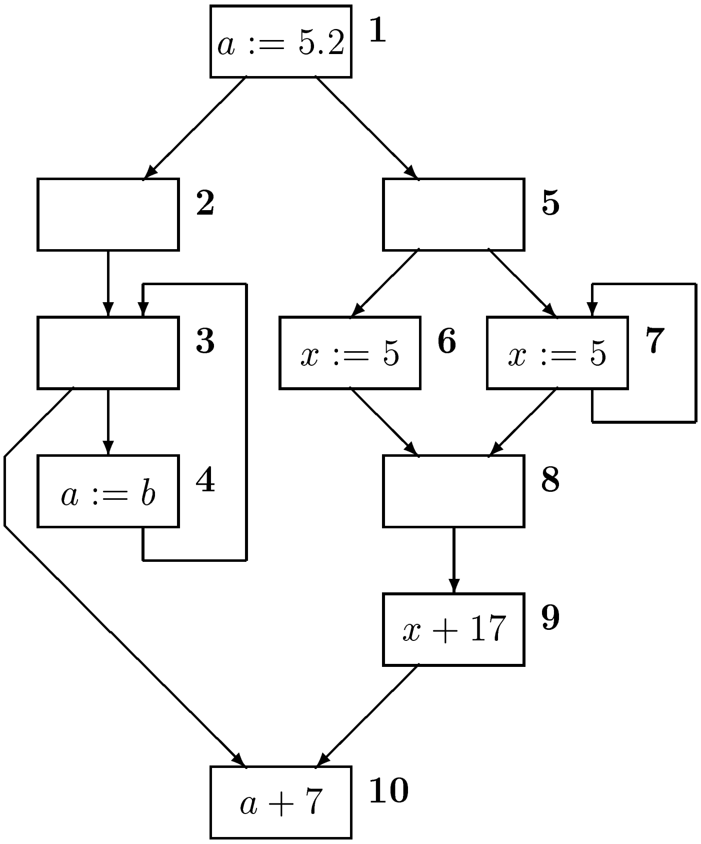
$x := a * 2.5;$

$a * 2.5$ **can be evaluated as** $3.1 * 2.5$ **during compilation.**

Q : **Under what conditions is it correct to perform constant propagation ?**

OPTIMISING TRANSFORMATIONS

1(b). CONSTANT PROPAGATION (Contd.)



Conditions for constant propagation :

A variable should be assigned the same constant value along all paths reaching its use.

In the above *program flow graph* **(formal definition later) constant propagation and folding is possible for** $x + 17$ **of block 9, however it is not possible for** $a + 7$ **of block 10. (***Q:* **Why ?)**

OPTIMISING TRANSFORMATIONS

2. COMMON SUB-EXPRESSION ELIMINATION (CSE)

An expression need not be evaluated if an equivalent value is available and can be used.

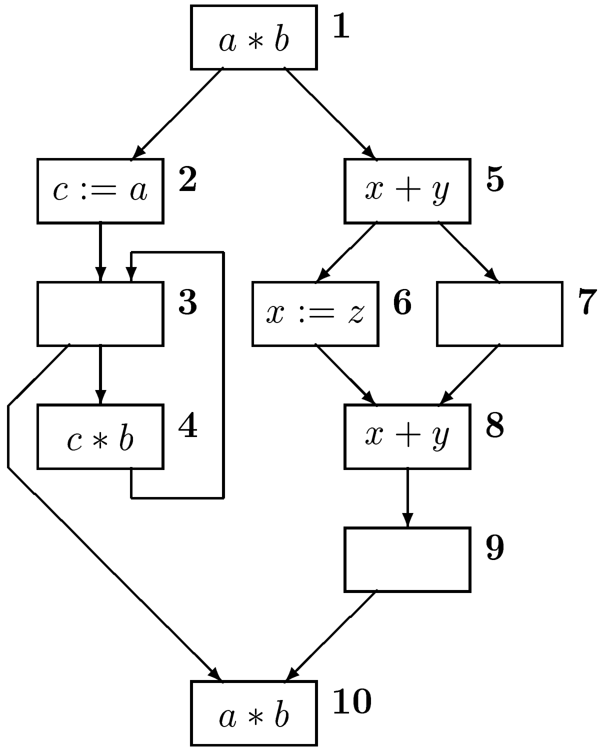
$$
\begin{array}{lcl}
 & & temp := b * c; \\
a := b * c; & & a := temp; \\
\quad \cdots & \Rightarrow & \quad \cdots \\
x := b * c + 5; & & x := temp + 5;
\end{array}
$$

Typically, the scope is restricted to *lexically equivalent* **expressions which evaluate to** *identical* **values during the execution of the program.**

Q : **Under what conditions is CSE optimisation feasible ?**
(*Hint :* **We must preserve** *semantic equivalence* **!**)

A : **Values of the operands must not change along any path between the two occurrences (i.e. the expression must be** *available*).

2. COMMON SUB-EXPRESSION ELIMINATION (Contd.)



1. $a * b$ of block 10 is a common subexpression. (Note that many occurrences of $a * b$ may exist in block 10 of the program, some of which may be eliminated by local optimisation. Global optimisation is only concerned with elimination of the *first* occurrence in the block.)
2. $x + y$ of block 8 is *not* a common subexpression,
3. $c * b$ of block is 4 is also a common subexpression, however it is harder to detect (*non-lexical equivalence !*).

OPTIMISING TRANSFORMATIONS

3. VARIABLE PROPAGATION

Use of a variable v_1 in place of variable v_2.

Example :

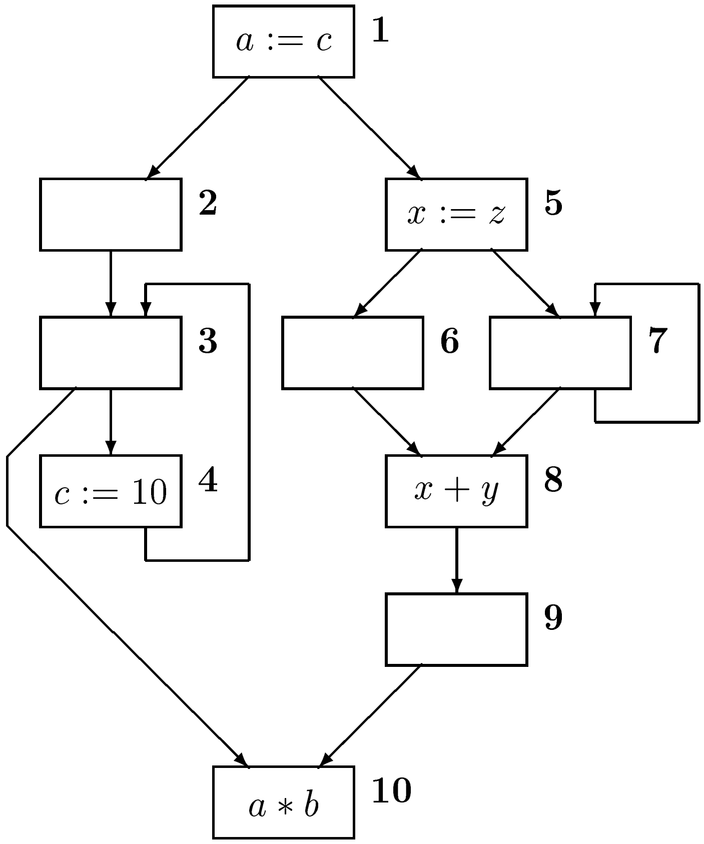
stmt no.	statement
1.	$c := d;$
2.	$\cdots$
3.	$\cdots$
10.	$z := c + e;$
11.	$x := d + e - 79.8;$

Use of d in place of c in statement no. 10 opens up the possibility of identifying $d + e$ of statement no. 11 as a common sub-expression.

Q : **Under what conditions can variable propagation be performed ?**

OPTIMISING TRANSFORMATIONS

3. VARIABLE PROPAGATION (Contd.)



Conditions for variable propagation :

Along all paths reaching its use, a variable should be assigned the value of the same *rhs* variable, and neither variable should be modified following such assignment.

a **can *not* be replaced by** c **in block 10 due to** $c := 10$ **of block 4. However,** x **can be replaced by** z **in block 8.**

OPTIMISING TRANSFORMATIONS

4. CODE MOVEMENT OPTIMISATION

Move the code in a program so as to

- **Reduce the size of the program,**
  — *Code space reduction*
- **Reduce the execution frequency of the code subjected to movement.**
  — *Execution frequency reduction*

Example : **Code space reduction by hoisting**

$$
\begin{array}{lcl}
 & & temp := x \uparrow 2; \\
if\ \ a < b\ then & & if\ \ a < b\ then \\
\qquad z := x \uparrow 2; & & \qquad z := temp; \\
\qquad \ldots & \Rightarrow & \qquad \ldots \\
else & & else \\
\qquad y := x \uparrow 2 + 19; & & \qquad y := temp + 19;
\end{array}
$$

Code for $x \uparrow 2$ is generated only once in the optimised program, as against twice in the original program.

OPTIMISING TRANSFORMATIONS

4. CODE MOVEMENT OPTIMISATION (Contd.)

Examples of Execution Frequency Reduction

(a) Hoisting of code

$$
\begin{array}{lll}
\begin{array}{l}
if\ \ a < b\ then \\
\qquad z := x \uparrow 2; \\
\qquad \cdots \\
else \\
\qquad y := 19; \\
\\
g := x \uparrow 2;
\end{array}
& \Rightarrow &
\begin{array}{l}
if\ \ a < b\ then \\
\qquad temp := x \uparrow 2; \\
\qquad z := temp; \\
else \\
\qquad y := 19; \\
\qquad temp := x \uparrow 2; \\
g := temp;
\end{array}
\end{array}
$$

During execution, $x \uparrow 2$ was evaluated twice in the original program under the condition $a < b$. In the optimised program, it will be evaluated only once.

Q : **Under what conditions can code movement result in frequency reduction ?**

A : **The expression must be** *partially available*, **i.e. available along at least one path reaching the evaluation of the expression.**

OPTIMISING TRANSFORMATIONS

4. CODE MOVEMENT OPTIMISATION (Contd.)

Safety of code movement

Movement of an expression e from some block b_i to block b_j is *safe* only if it does not introduce a new occurrence of e along any path in the program.

Note that unsafe code placement may lead to surprising exception conditions, e.g. overflow, during the execution of the program. This is different from incorrect results when only expressions (rather than assignments) are being moved !

Example of unsafe hoisting

$$
\begin{array}{lll}
\quad \cdots & & \quad temp := x \uparrow 2; \\
if\ a < b\ then & & if\ a < b\ then \\
\quad z := x \uparrow 2; & \Rightarrow & \quad z := temp; \\
else & & else \\
\quad y := 19; & & \quad y := 19;
\end{array}
$$

Here, $x \uparrow 2$ is newly inserted in the *else* branch of the if statement. This is unsafe.

Unsafe movements of code should be avoided.

OPTIMISING TRANSFORMATIONS

4. CODE MOVEMENT OPTIMISATION (Contd.)

Examples of Execution Frequency Reduction

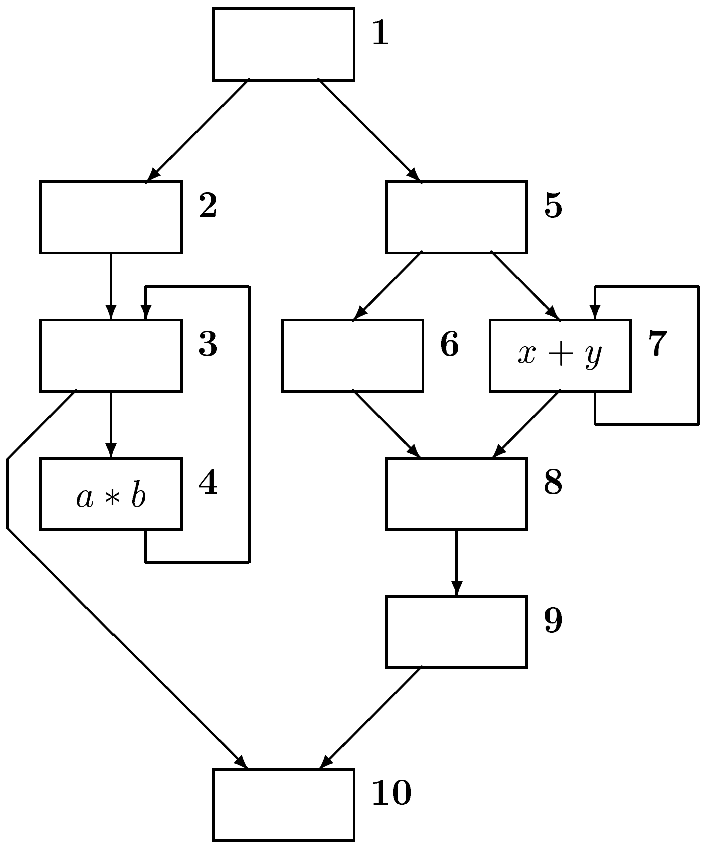
(b) Loop Optimisation :

$$
\begin{array}{lll}
 & & temp := x * y; \\
for\ i := 1\ to\ 10; & & for\ i := 1\ to\ 10; \\
\qquad \ldots & & \qquad \ldots \\
\qquad z := x * y; & \Rightarrow & \qquad z := temp; \\
end; & & end;
\end{array}
$$

Conditions for loop optimisation :

- **The** *rhs* **expression must be loop invariant.**
- **The** *rhs* **expression must** *dominate* **all loop exits, i.e. the node containing the expression must lie along all paths reaching a loop exit.**
  Q : **Why ? (** *Hint :* **See the previous transparency.)**

4(b). Loop optimisation (Contd.)



1. $a * b$ **of block 4 does not dominate all exits of the loop {3,4}, hence its movement out of the loop is unsafe!** (*Note :* **loop optimisation of while loops is unsafe unless some special techniques are used !**)

2. $x + y$ **of block 7 can be safely moved out of loop {7}. However, it is unsafe to insert it into block 5 !**

OPTIMISING TRANSFORMATIONS

5. STRENGTH REDUCTION OPTIMISATION

Replacement of a *high strength* operator by a (possibly repeated) application of a *low strength* operator.

Example : **Replacement of '*' by repeated '+'.**

$$
\begin{array}{lll}
 & & temp := 5; \\
for\ i := 1\ to\ 10; & & for\ i := 1\ to\ 10; \\
\quad \ldots & & \quad \ldots \\
\quad x := i * 5; & \Rightarrow & \quad x := temp; \\
\quad \ldots & & \quad \ldots \\
end; & & temp := temp + 5; \\
 & & end;
\end{array}
$$

Practical scope of strength reduction :

Address calculation in array references typically involves '*', which can be reduced to '+'. Consider

$$
\begin{array}{ll}
for\ i := 1\ to\ 50; & \\
\quad a[i] := ...; & \{\ \textbf{Each element} = \textbf{4 bytes}\ \} \\
end; &
\end{array}
$$

Address of $a[i]$ = address of $a[0] + i * 4$, assuming each element of array a to be 4 bytes in length.

5. STRENGTH REDUCTION OPTIMISATION (Contd.)

Strength reduction is typically applied to *integer* expressions involving an *induction variable* and a high strength operator.

Induction variables

An induction variable v is an integer scalar variable which is only subjected to the following kinds of assignments in a loop :

$$v := v \pm constant;$$

Controlled variable of a *for* loop is an induction variable.

Note :

Strength reduction is *not* performed for floating point expressions because a strength reduced program may produce different results than the original program.

Q : **Why ?**

A : **Consider finite precision of computer arithmetic.**

6. LOOP TEST REPLACEMENT

Replace a loop termination test phrased in terms of one variable, by a test phrased in terms of another variable. This may open up the possibilities of dead code elimination. Typically useful following strength reduction.

Example : **A strength reduced program -**

$$
\begin{array}{rl}
 & temp := 5; \\
 & i := 1; \\
loop: & x := temp; \\
 & i := i + 1; \\
 & temp := temp + 5; \\
 & if\ \ i \leq 10\ then \\
 & \qquad goto\ loop;
\end{array}
\quad \Rightarrow \quad
\begin{array}{rl}
 & temp := 5; \\
 & i := 1; \\
loop: & x := temp; \\
 & i := i + 1; \\
 & temp := temp + 5; \\
 & if\ \ temp \leq 50\ then \\
 & \qquad goto\ loop;
\end{array}
$$

The loop induction variable i is no longer meaningfully used in the program. Hence it can be eliminated.

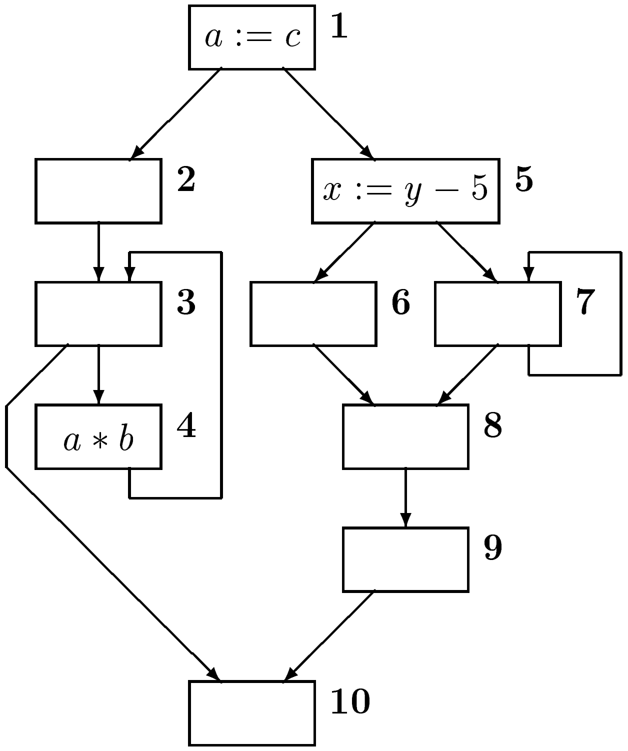
OPTIMISING TRANSFORMATIONS

7. DEAD CODE ELIMINATION

Preliminaries

1. **A variable is said to be *dead* at a place in a program if the value contained in the variable at that place is not used anywhere in the program.**
2. **If an assignment is made to a variable v at a place where v is dead, then the assignment is a *dead assignment*.**
3. **Removing a dead assignment makes no difference to the meaning/results of the program.**

7. DEAD CODE ELIMINATION (Contd.)



1. The assignment $a := c$ of block 1 is not dead (a is used in block 4).

2. The assignment $x := y - 5$ of block 5 is dead. The expression $y-5$ can also be eliminated, since it is no longer meaningful. However an expression capable of producing *side effects* can not be so eliminated.
   Q : Why ? (*Hint :* Think of function calls.)

LOCAL OPTIMISATION

- **Restricted to essentially sequential code**
- **Limited scope for optimisation, viz. loop optimisation, strength reduction, etc. not possible**
- **Low cost of optimisation : can be performed while converting a program to triples/quadruples**

Basic Block

A basic block b of a program P is a sequence of program statements (instructions) $\phi = (i_1, i_2, \dots i_m)$ such that

(i) only i_1 in b can be the destination of a transfer of control instruction,

(ii) only i_m in b can be a transfer instruction itself.

Example :

$$\boxed{\begin{array}{c} a := x * y; \\ \dots \\ z := x; \\ \dots \\ b := z * y; \end{array}}$$

Note the ease of variable propagation and CSE in the above example.

LOCAL OPTIMISATION

1. DAG BASED OPTIMISATION

Build a DAG for the basic block under consideration

- **Initially each variable is represented by a node. The Symbol Table entry identifies the node.**
- **Each node is labelled with the name(s) of the variable(s) whose value it represents.**
- **For each operation in an expression, a new node is created with pointers to the nodes of its operands. A new node is not created if a matching node exists.**
- **At an assignment, the root of the** *rhs* **expression is labelled with the name of the** *lhs* **variable. (SYMTAB entry of the lhs variable is changed appropriately.)**
- **Copy propagation and CSE are now easy**

Example :

stmt no.	statement
1.	$a := x * y;$
2.	$z := x;$
3.	$b := z * y;$
4.	$x := b;$
5.	$g := x * y;$

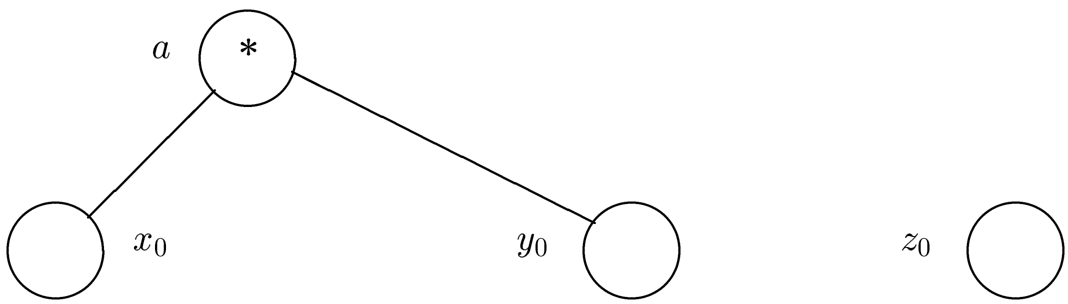
LOCAL OPTIMISATION

1. DAG BASED OPTIMISATION (Contd.)

Example :

stmt no.	statement
1.	$a := x * y;$
2.	$z := x;$
3.	$b := z * y;$
4.	$x := b;$
5.	$g := x * y;$

After processing statement 1 :



Note : x_0, y_0 **and** z_0 **represent the initial values of variables** x, y **and** z **respectively.**

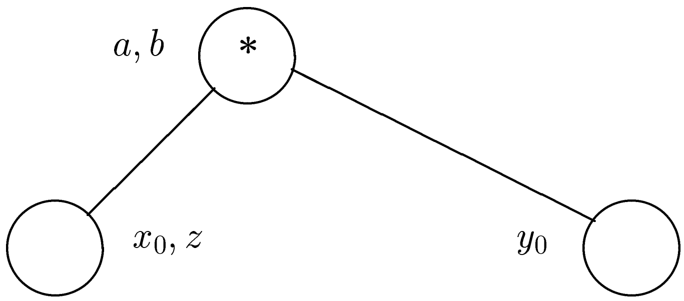
LOCAL OPTIMISATION

1. DAG BASED OPTIMISATION (Contd.)

Example :

stmt no.	**statement**
1.	$a := x * y;$
2.	$z := x;$
3.	$b := z * y;$
4.	$x := b;$
5.	$g := x * y;$

After processing statements 1-3 :



Note : **Variable propagation has occurred for variable z, while common sub-expression elimination has been effected for $a * b$.**

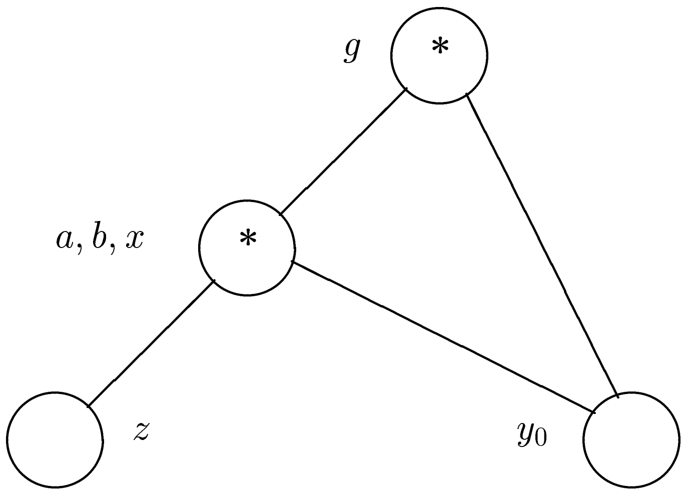
LOCAL OPTIMISATION

1. DAG BASED OPTIMISATION (Contd.)

Example :

stmt no.	statement
1.	$a := x * y;$
2.	$z := x;$
3.	$b := z * y;$
4.	$x := b;$
5.	$g := x * y;$

After processing the entire basic block :



LOCAL OPTIMISATION

2. QUADRUPLE BASED OPTIMISATION USING VALUE NUMBERS

A note on unique result names for quadruples :

To simplify the elimination of common sub-expressions, it is necessary that two quadruples having the same operator and identical operands must have unique result names.

Example

$$a * b$$
$$\ldots$$
$$a * b + c$$

If the quadruple generated for the first occurrence of $a * b$ uses the result name temp$_1$, then the result name for the second occurrence of $a * b$ should be identical.

LOCAL OPTIMISATION

2. QUADRUPLE BASED OPTIMISATION USING VALUE NUMBERS

Notes :

1. **The result name in a quadruple is not necessarily a 'temporary location'. It simply provides a convenient means to refer to the partial result represented by the quadruple.**

2. **The compiler can maintain a hash table of the first three fields of quadruples to ensure uniqueness of the result name.**

3. **The simplification resulting from unique result names is as follows : In the above example, if the second occurrence of $a * b$ is redundant, it can simply be replaced by the result name in the quadruple, viz. temp_1. This can be done by simply examining the result name of the quadruple, and without having to know the other occurrences of $a * b$.**

4. **The result name becomes a temporary location only when some occurrences of the expression can be eliminated.**

LOCAL OPTIMISATION

2. QUADRUPLE BASED OPTIMISATION USING VALUE NUMBERS (Contd.)

The value number associated with a variable uniquely identifies the place in the basic block where the variable was last assigned a value.

- **At the start of a basic block, the value numbers of all variables are initialised**
- **At an assignment, the value number of the LHS variable is changed to a new value**
- **In the quadruples table, the pair (symbol, value no.) is stored in the operand field. (Note that the table of quadruples is the intermediate representation of the program. This is distinct from the hash table maintained to ensure uniqueness of the result names.)**
- **For CSE, a new quadruple is compared with all existing quadruples (*note :* the value numbers also participate in a comparison.)**
- **On processing the statement**

  $$v := 25.3;$$

  the value number of v is set to $-m$ where 25.3 occupies the m^{th} entry in CONSTAB. This feature is used for Constant Propagation.

2. QUADRUPLE BASED OPTIMISATION USING VALUE NUMBERS (Contd.)

Procedure for optimization :

1. For every evaluation, a quadruple is generated in a buffer.
2. Current value numbers of the operands are copied from the symbol table.
3. If each operand is either constant or has a negative value number, perform constant folding and skip step 4.
4. The quadruple in the buffer is compared with all existing quadruples in the table.
   (a) Enter the new quadruple in the table (with flag = 0) if no matching quadruple is found.
   (b) If a matching quadruple is found in the table, its flag is changed to 1.
      - The newly generated quadruple is not entered in the table. (This is an instance of common subexpression elimination.)
      - Flag = 1 indicates that the value of the quadruple should be saved for later use.

LOCAL OPTIMISATION

2. QUADRUPLE BASED OPTIMISATION USING VALUE NUMBERS (Contd.)

Example :

stmt no.	statement
5.	$a := 29.3 * d;$
17.	$b := 24.5;$
31.	$c := a * b + w;$
49.	$x := a * b + y;$

After processing statements 1-31 :

Symbol table

symbol	val #
a	5
b	-75
c	31
x	0
w	0

Quadruples table

oper	operand 1		operand 2		result	flag
	sym	val #	sym	val #	name	
...	...	...	...	...	...	...
*	a	5	b	-75	T35	0
+	T35	–	w	0	T56	0

Note : **Value number of '-75' for b implies that b has been assigned the constant occupying 75^{th} entry in the Constants' table (i.e., 24.5).**

LOCAL OPTIMISATION

2. QUADRUPLE BASED OPTIMISATION USING VALUE NUMBERS (Contd.)

Example :

stmt no.	statement
5.	$a := 29.3 * d;$
17.	$b := 24.5;$
31.	$c := a * b + w;$
49.	$x := a * b + y;$

After processing statements 1-49 :

Symbol table

symbol	val #
a	5
b	-75
c	31
x	49
w	0

Quadruples table

oper	operand 1		operand 2		result	flag
	sym	val #	sym	val #	name	
...	...	...	...	...	...	...
*	a	5	b	-75	T35	1
+	T35	–	w	0	T56	0
+	T35	–	y	0	T92	0

GLOBAL OPTIMISATION

Scope

The scope of global optimisation is generally a program unit, viz. a procedure or function body.

Program Representation

The program is represented in the form of a *Program Flow Graph*. **The nodes of the graph represent the basic blocks in the program, and the edges represent the flow of control during the execution of the program.**

Control Flow Analysis

Determines information concerning the arrangement of the graph nodes, i.e. the *structure* **of the program, viz. the presence of loops, nesting of loops, nodes visited before the control of execution reaches a specific node, etc.**

Data Flow Analysis

Determines useful information for the purpose of optimisation, viz. how data items are assigned and referenced in a program, values available when program execution reaches a specific statement of the program, etc.

GLOBAL OPTIMISATION

CONTROL FLOW ANALYSIS

Concepts and Definitions

Program Point

A program point w_j is the instant between the end of execution of instruction i_j, and the beginning of the execution of the instruction i_{j+1}. The effect of execution of instruction i_j is said to be completely realised at program point w_j.

Program Flow Graph

A program flow graph is a directed graph

$$G = (N, E, n_0)$$

where N is the set of nodes (i.e. basic blocks),
E is the set of control flow edges (b_i, b_j),
n_0 is the entry node of the program.

Paths

A sequence of edges (e_1, $e_2, \ldots e_l$), such that the terminal node of e_i is the initial node of e_{i+1}, is known as a *path* **in G.**

Predecessors, Successors, Ancestors and Descendants

b_i is a predecessor (ancestor) of b_j if there exists an edge (a path) from b_i to b_j. A successor / descendant is analogously defined.

GLOBAL OPTIMISATION

CONTROL FLOW ANALYSIS

Concepts and Definitions (Contd.)

Dominators

A block b_i is said to be a *dominator* of block b_j if every path from n_0 to b_j in G passes through b_i.

b_i is a *post-dominator* of b_j if every path starting on b_j passes through b_i before reaching an exit node of G.

Regions

A region $R = (V, E^{'}, V^{'})$ is a connected subgraph of G, where $V \subseteq N$, $E^{'} \subseteq E$ and $V^{'} \subseteq N$ represent the set of nodes, edges and entry nodes respectively. We will have occassion to use many kinds of regions, viz.

- **loops**
- **single entry regions**
- **strongly connected regions**
- **intervals**

Articulation Block

b is an articulation block for R if every path from an entry node to an exit node of R necessarily passes through b.

GLOBAL OPTIMISATION

CONTROL FLOW ANALYSIS

Concepts and Definitions (Contd.)

Q : **Where do we use these concepts ?**

A : **Consider the following situations –**

(a) **It is correct to move some code out of a node i if**

- **it is inserted into a dominator node j of i,**
- **no assignment(s) to any operands of the code occur along any path $j \ldots i$.**

Hint : **Is it always safe to do this ?**

(b) **Meaning of a program may change unless some code c, which is moved out of a loop, occurs in an articulation block of the loop.**

(c) **It is incorrect to move any code out of a loop unless its occurrence(s) dominate all exit nodes of the loop.**

(d) **An expression e can be eliminated from a program point w if and only if,**

- **there exists at least one evaluation of e along every path reaching point w,**
- **no operand of e is assigned a value after last such evaluation.**

DATA FLOW ANALYSIS

Determines useful information for the purpose of optimisation.

Data Flow Property

A data flow property represents an item of data flow information (or a set of items of data flow information), e.g. set of expressions whose values are available.

- **Data flow properties are typically associated with entities in the program flow graph, viz. nodes in the program graph.**
- *Data flow analysis* **is the process of computing the values of data flow properties.**
- **Data flow properties are defined by the compiler writer for use in a specific optimisation.**

A few fundamental data flow properties are defined in the following transparencies.

DATA FLOW ANALYSIS

PRELIMINARIES

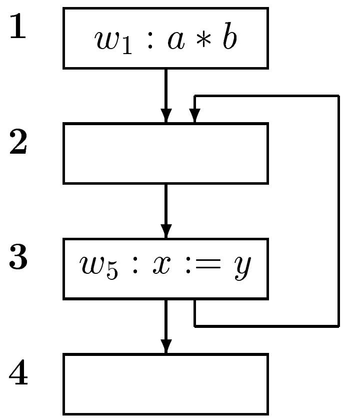
Definition / Reference point

A program point containing a definition (i.e. assignment) / reference of a data item.

Evaluation point for an expression

A program point containing an evaluation of the expression.

Example :



- w_5 **is a reference point for** y. **It is also a definition point for** x. **(Strictly speaking,** $w_5^{'}$ **is the reference point for** y, $w_5^{''}$ **is the definition point for** x, **where** $w_5^{'}$ **precedes** $w_5^{''}$.**)**
- w_1 **is an evaluation point for expression** $a * b$.

DATA FLOW ANALYSIS

FUNDAMENTAL DATA FLOW PROPERTIES

1. Available Expressions

Expression e is *available* at program point w, iff *along all paths* reaching w

(i) there exists an evaluation point for e,

(ii) no definition of any operand of e follows its last evaluation along the path.

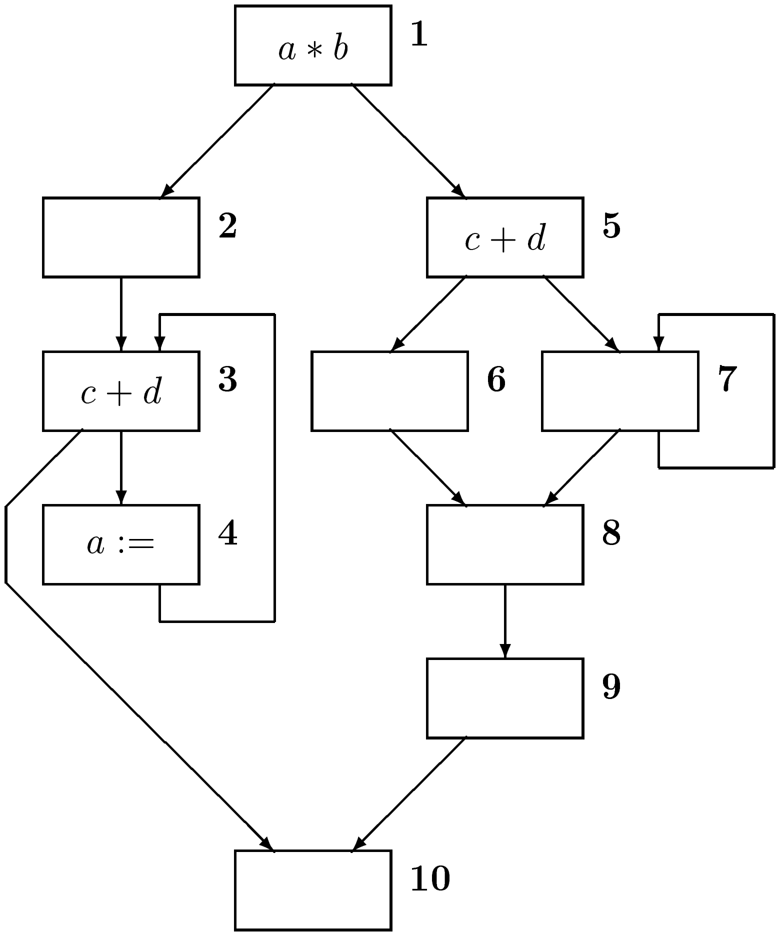
Expression e is said to be *killed* by a definition of any of its operands. Hence it is not available following the definition(s) of any of its operand(s).

Note that an expression is said to be killed, irrespective of whether or not its value is available at the point of the definition. This convention simplifies data flow analysis, as we will see later.

Usage : Common sub-expression elimination

DATA FLOW ANALYSIS

1. Available Expressions (Contd.)



Expressions available at entry of node 10 ?

— $\{c + d\}$

FUNDAMENTAL DATA FLOW PROPERTIES (Contd.)

2. Reaching Definitions

A definition d of a variable v situated at a program point w_i is said to *reach* a program point w_j iff *along some path* $w_i \dots w_j$ variable v is not re-defined.

In other words, definition d of variable v is said to reach w_j only when variable v, if used at w_j, is likely to have the value assigned to it by definition d.

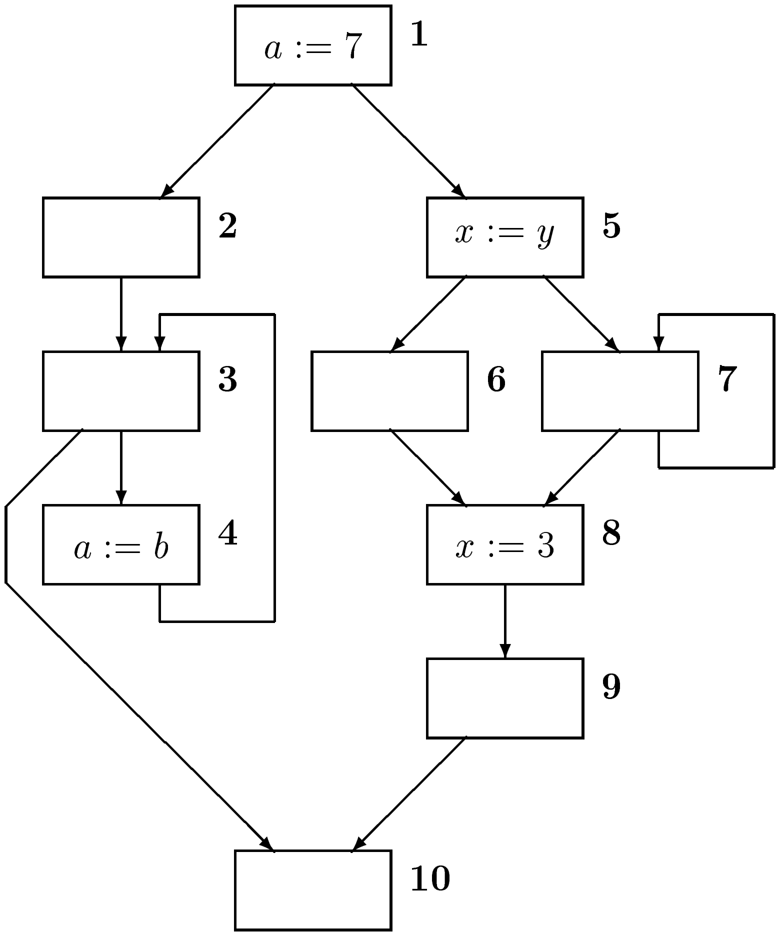
Q : **Where is this data flow concept useful ?**

Let a definition $x := 5$ reach a program point at which the expression $x * 3$ is located. Can constant propagation be performed for the variable x such that $x * 3$ can be replaced by $5 * 3$, and can be folded ?

— Only if $x := 5$ is the only definition of x reaching $x * 3$!

DATA FLOW ANALYSIS

2. Reaching Definitions (Contd.)



Definitions reaching the entry of node 10 ?

— $\{a := b, a := 7, x := 3\}$

FUNDAMENTAL DATA FLOW PROPERTIES (Contd.)

3. Live Variables

A variable v is *live* at a program point w_i iff

(i) **v is referenced** *along some path* $w_i \ldots w_j$ **starting on program point w_i, and**

(ii) **no assignment to v occurs before its reference along the path.**

In other words, variable v is live at w_i only if the value existing in v at point w_i is likely to be used in some computation. If this is not the case, then v is dead.

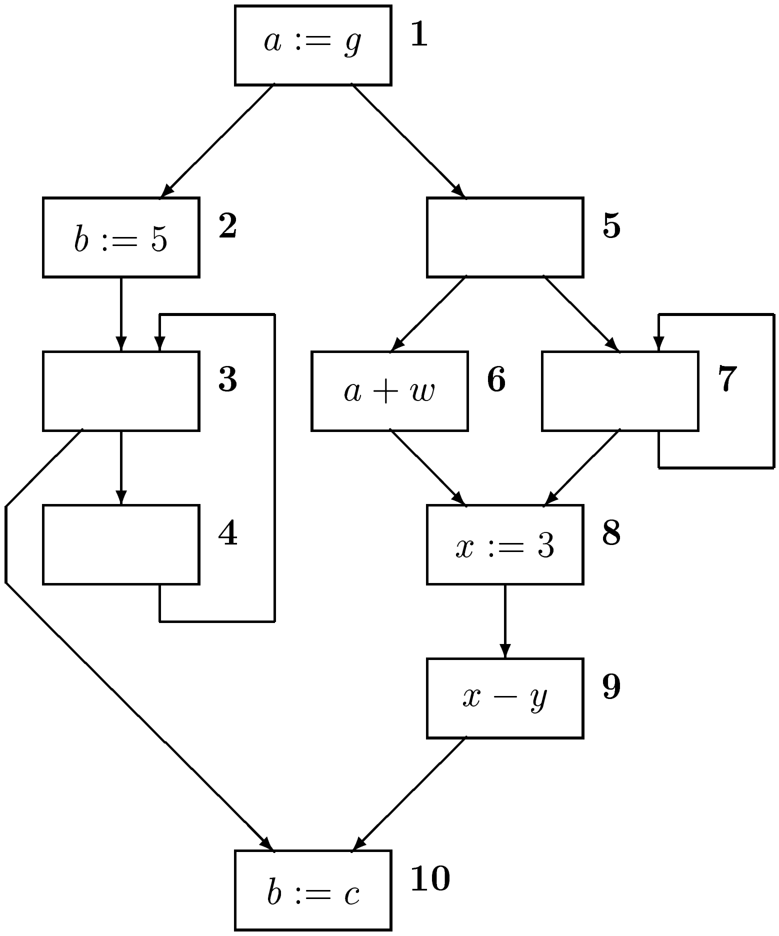
Usage :

1. **We can eliminate an assignment to a dead variable, viz. $x := e$; where x is dead.**
   Hint : **Expression e should have no side effects !**

2. **We can release / reuse storage allocated to a dead variable, since its value need not be maintained any longer.**

DATA FLOW ANALYSIS

3. Live Variables (Contd.)



Variable a is live in nodes : 1, 5, 6.

Variable x is live in nodes : 8, 9.

Variable b is not live in any node.

DATA FLOW ANALYSIS

FUNDAMENTAL DATA FLOW PROPERTIES (Contd.)

4. Busy Expressions

An expression e is *busy* at a program point w_i iff

(i) an evaluation of e exists *along some path* $w_i \ldots w_j$ starting on program point w_i, and

(ii) no definition of any operand of e exists before its evaluation along the path

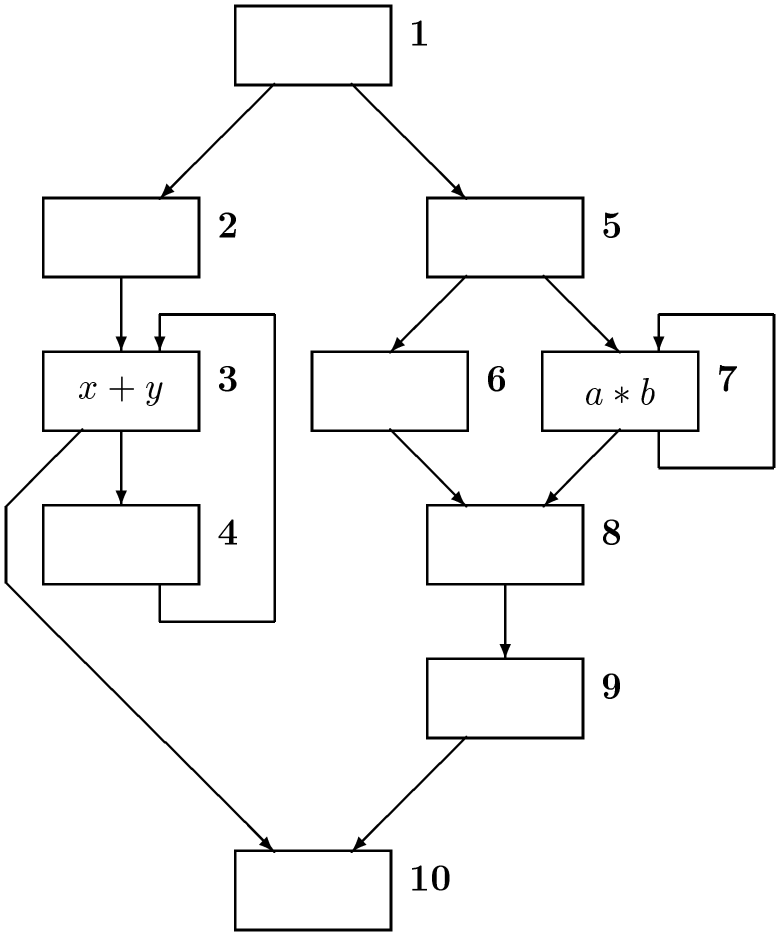
In other words, if the expression were to be evaluated at program point w_i, it would be useful along some path.

Very Busy Expression

If the expression is busy *along all paths* starting at a program point w_i.

Q : **Is it safe to insert an evaluation of expression e at a program point at which it is not very busy ?**

4. Busy Expressions (Contd.)



$a * b$ **is busy in node 5, but not very busy.**

— **Hence its movement from node 7 to 5 is unsafe !**

$x + y$ **is busy and very busy in node 2.**

— **Hence its movement from node 3 to 2 is safe.**

DATA FLOW ANALYSIS

REPRESENTING DATA FLOW INFORMATION

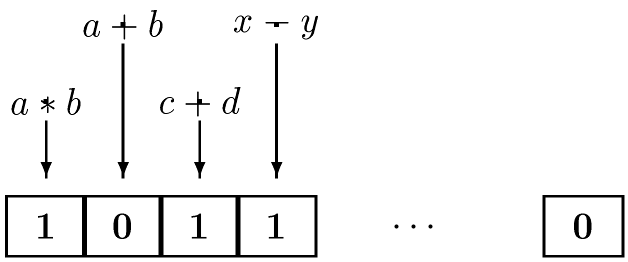
Data flow information can be represented by a *set of properties*, **or by a** *bit vector* **with each bit representing a property. The former is more general, while the latter is more convenient in practice. Unless otherwise stated, these representations can be used interchangeably in our discussions.**

Example : Available expressions

(a) Set Representation

$$\{\ a * b,\ c + d,\ x - y,\ \cdots\ \}$$

(b) Bit Vector Representation



Q : **What is the size of the bit vector ?**

A : **Size equals the number of distinct expressions in a program. The bit number for an expression can be determined the same way as the temporary name for it (i.e. by building a table of unique expressions in the program).**

DATA FLOW ANALYSIS

OBTAINING DATA FLOW INFORMATION

Data flow information of a node has the following components :

(a) Data flow information *generated* in a node, viz. an expression becomes available following its computation in the node.

(b) Data flow information *killed* in a node, viz. a definition of variable v kills all expressions involving v.

(c) Data flow information *obtained* from neighbouring nodes, viz. a definition reaching the exit of a predecessor also reaches the entry of a node.

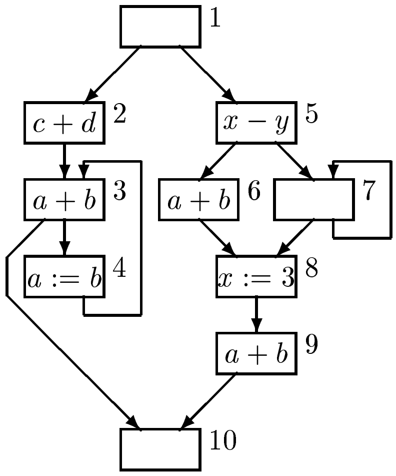
Information in items (a) and (b) above is *local* in nature, i.e. the data flow information generated and killed in a node of the program flow graph depends on the nature of the computations in the corresponding basic block of the program. Information in item (c) is *non-local* in nature.

Hence

- **data flow information at a node depends on the data flow information at other nodes in the program flow graph.**
- **data flow information at the entry and exit of a node is likely to be different.**

DATA FLOW ANALYSIS

LOCAL DATA FLOW INFORMATION



node	kill	gen
1	ϕ, **i.e. 0000**	ϕ, **i.e. 0000**
2	ϕ, **i.e. 0000**	$\{c + d\}$, **i.e. 0010**
3	ϕ, **i.e. 0000**	$\{a + b\}$, **i.e. 0100**
4	$\{a * b,\ a + b\}$, **i.e. 1100**	ϕ, **i.e. 0000**
5	ϕ, **i.e. 0000**	$\{x - y\}$, **0001**
6	ϕ, **i.e. 0000**	$\{a + b\}$, **i.e. 0100**
8	$\{x - y\}$, **i.e. 0001**	ϕ, **i.e. 0000**
9	ϕ, **i.e. 0000**	$\{a + b\}$, **i.e. 0100**

DATA FLOW ANALYSIS

COMPUTING GLOBAL DATA FLOW INFORMATION

1. **Meet over paths solution (MOP)**

   (a) **Consider all paths reaching (starting on) a node.**

   (b) **Determine information flow along each path.**

   (c) **Take the meet over all the paths, i.e.** *merge* **the information about the paths in an appropriate manner to determine the data flow information obtained from the neighbours of the node.**

Example : **Available expressions for node** i

1. **Consider all paths reaching node** i **(and starting on the program entry node** n_0**).**

2. **Compute the set of available expressions along each path from** n_0 **to a predecessor of node** i**. (Assume that the set of available expressions at the entry of** n_0 **is** ϕ**.)**

3. **Since available expressions is an** *all paths* **problem, take the** *intersection* **of available expressions along each path. This gives the set of available expressions at the entry of node** i**.**

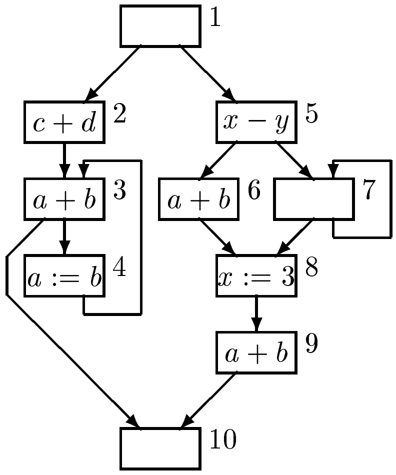
Thus the *meet* **operator here is** *intersection***.**

DATA FLOW ANALYSIS

COMPUTING GLOBAL DATA FLOW INFORMATION

1. Meet over paths solution (MOP) (Contd.)

Example : Available expressions



node	available expressions at entry	available expressions at exit
2	ϕ, **i.e. 0000**	$\{c+d\}$, **i.e. 0010**
3	$\{c+d\}$, **i.e. 0010**	$\{a+b,\ c+d\}$, **i.e. 0110**
4	$\{a+b,\ c+d\}$, **i.e. 0110**	$\{c+d\}$, **i.e. 0010**
8	$\{x-y\}$, **i.e. 0001**	ϕ, **i.e. 0000**
9	ϕ, **i.e. 0000**	$\{a+b\}$, **i.e. 0100**
10	$\{a+b\}$, **i.e. 0100**	$\{a+b\}$, **i.e. 0100**

DATA FLOW ANALYSIS

COMPUTING GLOBAL DATA FLOW INFORMATION

We can make the following observations concerning the problem of available expressions :

(a) *Generated information :* **An expression is generated in a node if the corresponding basic block contains a** *downwards exposed* **occurrence of the expression, i.e. an expression evaluation which is not followed by a definition of any of its operands till the end of the block.**

Example :

$$\boxed{\begin{array}{c} a * b \\ c + d \\ a := \ldots \end{array}} \quad 4$$

Here $\mathbf{Gen}_4 = \{\ c + d\ \}$**. Note that the occurrence of** $a * b$ **is not downwards exposed.**

(b) *Killed information :* **A definition of a variable (i.e. an assignment to it) kills all expressions involving the variable.**

(c) *Merging of information :* **Merging is done using the** *set intersection* **operation** $\cap$ **or the bitwise 'and' operation** $\prod$ **since this is an** *all paths* **problem.**

DATA FLOW ANALYSIS

FORWARD DATA FLOW PROBLEMS

In the available expressions problem

1. **The data flow information generated in a node flows to the exit of the node,**
2. **The data flow information reaches a node i along all paths from n_0 to i.**
3. **Thus, the data flow information always flows in the direction of the flow of control in the program.**

Such a data flow problem is called a *forward data flow problem*.

The problem of reaching definitions is also a forward data flow problem.

DATA FLOW ANALYSIS

RECOGNIZING FORWARD/BACKWARD DATA FLOW PROBLEMS

(a) Forward Data flow problems

We can recognise a data flow problem to be *forward* **when the data flow information for a program point w_i depends on the computations placed along one or more paths** *reaching* w_i**. In this case, the information must flow along the direction of flow of control in the program.**

(b) Backward Data flow problems

A data flow problem is *backward* **if the data flow information for a program point w_i depends on the computations placed along one or more paths** *starting on* w_i**. In this case, the information flows opposite to the direction of flow of control in the program since computations occurring along the path affect the data flow property of w_i.**

Live variables and Busy expressions are backward data flow problems.

Live Variables : A backward data flow problem

A variable v is *live* at a program point w_i iff v is referenced *along some path* $w_i \ldots w_j$ starting on w_i, and $\ldots$

(a) *Generated information :* **A variable v becomes live when it is referenced (i.e. used) in an expression. Since the data flow is backwards, the liveness due to a use in an expression extends *backwards* within the basic block, till a definition of v. Hence a live variable is generated in a basic block due to an *upwards exposed* reference of a variable v, i.e. a use of v not preceded by a definition within the basic block.**

Example :

$$\boxed{\begin{array}{r} a := \ldots \\ \ldots := a * b \\ \ldots := v * 3 \end{array}} \quad \mathbf{5}$$

Here Gen$_5$ = $\{ v, b \}$. Note that the reference of a is not upwards exposed.

(b) *Killed information :* **A definition of a variable (i.e. an assignment to it) kills its liveness.**

(c) *Merging of information :* **Merging is done using the *set union* operation $\cup$ or the bitwise 'or' operation $\sum$ since this is an *any path* problem.**

The problem of busy expressions is also a backward data flow problem.

DATA FLOW ANALYSIS

SUMMARY OF DATA FLOW PROBLEMS

Data Flow Problem	Generated Information	Killed Information	*Confluence* i.e., Merge
Available Expressions	*Downwards exposed* occ. of an exp.	Occ. $a := \ldots$ kills exps using a	$\cap$ or Π
Reaching Definitions	*Downwards exposed* occ. of $d_i : v := \ldots$	Occ. $d_k : v := \ldots$ kills $d_i^\prime s : v := \ldots$	$\cup$ or Σ
Live Variables	*Upwards exposed* reference of x	Occ. $x := \ldots$ kills variable x	$\cup$ or Σ
Very Busy Expressions	*Upwards exposed* occ. of an exp.	Occ. $a := \ldots$ kills exps using a	$\cap$ or Π

Notes :

(a) A *upwards / downwards exposed* occurrence of an expression implies an expression evaluation not preceded / followed by any operand definition in the basic block.

(b) Upwards exposed reference of a variable is analogously defined.

DATA FLOW ANALYSIS

COMPUTING GLOBAL DATA FLOW INFORMATION

2. Data Flow Equations :

In this approach, we use the following procedure to obtain global data flow information :

(a) Take the meet of the information available at the entry (exit) of each node.

(b) Consider the information being generated or killed within the node.

(c) Obtain the information available at the exit (entry) of the node.

To realise this, we set up a *data flow equation* **for each node of the graph.**

Example : available expressions

$$\begin{aligned} \mathbf{AVIN}_i &= \cap_{\forall p}\ \mathbf{AVOUT}_p \\ \mathbf{AVOUT}_i &= \mathbf{AVIN}_i - \mathbf{AVKILL}_i\ \cup\ \mathbf{AVGEN}_i \end{aligned}$$

where,

$\mathbf{AVIN}_i/\mathbf{AVOUT}_i$ is availability at entry/exit of i,
$\mathbf{AVKILL}_i$ represents information killed in node i,
$\mathbf{AVGEN}_i$ represents information generated in node i.
$\cap_{\forall p}$ represents $\cap$ over all predecessors.

COMPUTING GLOBAL DATA FLOW INFORMATION

2. Data Flow Equations (Contd.) :

Note the following points concerning the use of the approach based on data flow equations.

1. **The data flow equation for each node of the program graph is different**
   - **the local information** *gen* **and** *kill* **is different for each node.**
   - **predecessors/successors are different for each node.**
2. **The data flow equations have to be solved simultaneously for all nodes in the program graph. The solution of these equations has to satisfy the properties of self-consistency, conservativeness and meaningfulness. (More about this aspect later.)**
3. **Solution of data flow equations is more efficient than use of the MOP approach.**
4. **The MOP solution is of theoretical interest as it indicates the maximum data flow information for a program flow graph.**
5. **MOP approach is impractical due to various reasons (efficiency is only one of them !).**

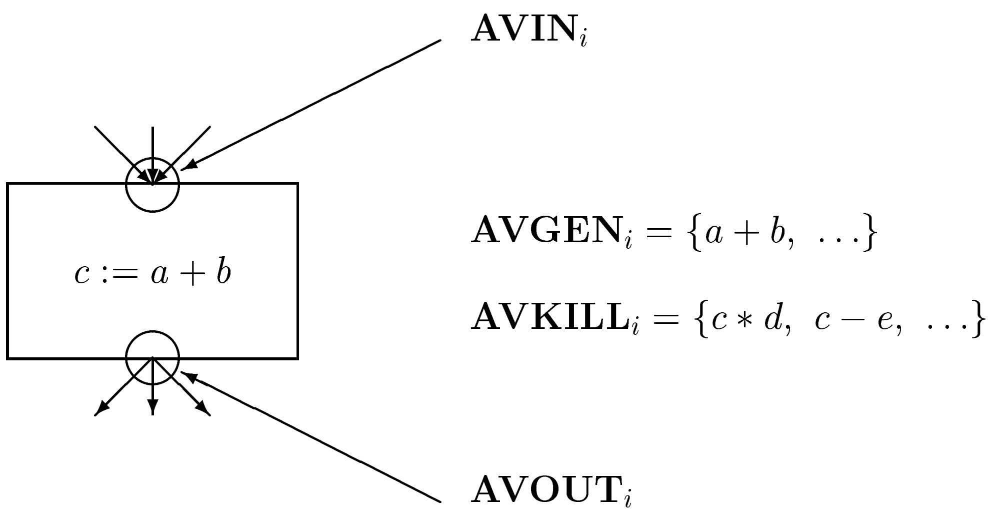
SETTING UP DATA FLOW EQUATIONS

$$\textbf{AVIN}_i = \cap_{\forall p}\ \textbf{AVOUT}_p$$

$$\textbf{AVOUT}_i = \textbf{AVIN}_i - \textbf{AVKILL}_i\ \cup\ \textbf{AVGEN}_i$$

where,

AVIN/AVOUT represent availability at entry/exit,
AVKILL represents information killed in node,
AVGEN represents information generated in node.



- $\forall i$ **AVKILL**$_i$ **and AVGEN**$_i$ **are** *constants* **which can be determined during the preparatory phase.**
- **Using these constants, we define a** *transfer function* $f_i(S)$ **for each basic block, as** $f_i(S) = (S-$ **AVKILL**$_i) \cup$ **AVGEN**$_i$**.**

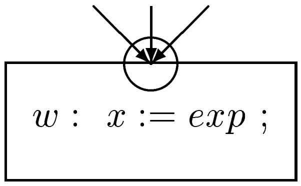
COMPUTING GLOBAL DATA FLOW INFORMATION

Q : **Is the information obtained through data flow equations equivalent to the MOP solution ?**

A: **Depends on *distributivity* of the data flow.**

The Distributivity Condition

$$f(\mu \sqcap v) = f(\mu) \sqcap f(\nu) \quad \forall\ \mu,\ \nu \text{ and } f$$



$$\mathbf{AVIN}_i = \cap_{\forall p}\ \mathbf{AVOUT}_p$$

- **For the MOP solution, we use $f(\mu) \sqcap f(\nu)$ at the point w (here μ and ν represent the data flow information along the paths reaching w, and f is the transfer function).**
- **For solving the data flow equations we use $\mu \sqcap \nu$ at the entry of each basic block (here μ and ν represent $\mathbf{AVOUT}_p$).**
- **The data flow equations can not yield the MOP solution unless the data flow is distributive.**

GENERAL FORM OF DATA FLOW EQUATIONS

1. Forward Data Flow Problems

$$\begin{aligned} \mathbf{IN}_i &= \theta_{\forall p}\ \mathbf{OUT}_p \\ \mathbf{OUT}_i &= \mathbf{IN}_i - \mathbf{KILL}_i\ \cup\ \mathbf{GEN}_i \end{aligned}$$

where,

θ is the confluence operator $\cup$ or $\cap$,
IN/OUT indicate information at entry/exit,
KILL indicates information killed in node,
GEN indicates information generated in node.

2. Backward Data Flow Problems

$$\begin{aligned} \mathbf{OUT}_i &= \theta_{\forall s}\ \mathbf{IN}_s \\ \mathbf{IN}_i &= \mathbf{OUT}_i - \mathbf{KILL}_i\ \cup\ \mathbf{GEN}_i \end{aligned}$$

where,

θ is the confluence operator $\cup$ or $\cap$,
IN/OUT indicate information at entry/exit,
KILL indicates information killed in node,
GEN indicates information generated in node.

COMMON DATA FLOW PROBLEMS

1. FORWARD DATA FLOW PROBLEMS

(a) Available Expressions

$$\begin{aligned}\mathbf{AVIN}_i &= \cap_{\forall p}\ \mathbf{AVOUT}_p \\ \mathbf{AVOUT}_i &= \mathbf{AVIN}_i - \mathbf{AVKILL}_i\ \cup\ \mathbf{AVGEN}_i\end{aligned}$$

where,

AVIN/AVOUT indicate availability at entry/exit,
AVKILL indicates presence of operand definition(s),
AVGEN indicates expressions generated in node.

(b) Reaching Definitions

$$\begin{aligned}\mathbf{DEFIN}_i &= \cup_{\forall p}\ \mathbf{DEFOUT}_p \\ \mathbf{DEFOUT}_i &= \mathbf{DEFIN}_i - \mathbf{DEFKILL}_i\ \cup\ \mathbf{DEFGEN}_i\end{aligned}$$

where,

DEFIN/DEFOUT indicate reaching defs. at entry/exit,
DEFKILL indicates presence of another definition(s),
DEFGEN indicates definitions generated in node.

COMMON DATA FLOW PROBLEMS

2. BACKWARD DATA FLOW PROBLEMS

(a) Busy Expressions

$$\begin{aligned}\textbf{BUSYOUT}_i &= \cup_{\forall s}\ \textbf{BUSYIN}_s \\ \textbf{BUSYIN}_i &= \textbf{BUSYOUT}_i - \textbf{BUSYKILL}_i \ \cup\ \textbf{BUSYGEN}_i\end{aligned}$$

where,

BUSYIN/BUSYOUT indicate busy exps. at entry/exit,
BUSYKILL indicates presence of operand definition(s),
BUSYGEN indicates busy expressions generated in node.

(b) Live Variables

$$\begin{aligned}\textbf{LIVEOUT}_i &= \cup_{\forall s}\ \textbf{LIVEIN}_s \\ \textbf{LIVEIN}_i &= \textbf{LIVEOUT}_i - \textbf{LIVEKILL}_i \ \cup\ \textbf{LIVEGEN}_i\end{aligned}$$

where,

LIVEIN/LIVEOUT indicate variables live at entry/exit,
LIVEKILL indicates presence of another definition(s),
LIVEGEN indicates live variables generated in node.

DATA FLOW ANALYSIS

SETTING UP DATA FLOW EQUATIONS

We will consider the process of setting up data flow equations to collect the information required for an optimisation. Consider the following specification of an optimisation.

Copy Propagation

We can replace a by b in the following statement if a is a *copy* of b at program point w (i.e. if a has the same value as b at w).

$$\boxed{w: \; z := a + y \; ;}$$

Approach :

1. **Decide on *what* information is adequate to perform the desired substitution. (*Note :* You may define your own concepts and notations for this purpose.)**
2. **Analyse the nature of the information to decide *how* it may be collected.**
3. **Design a data flow problem to collect the information. Develop data flow equations for the same.**

SETTING UP DATA FLOW EQUATIONS

Example (Contd.) :

1. **What information is adequate to perform the desired substitution ?**

$$\boxed{w: \; z := a + y \, ;} \longleftarrow \quad \textbf{COPIES}_w = \{\cdots\}$$

Let COPIES be a set of pairs

$\{(x, y) \mid$ **assignment** $x := y$; **exists along some path** $\}$

— **COPIES can be computed for every program point by using C_IN and C_OUT to be the set of copies associated with the entry/exit of nodes.**

2. **Analyse the nature of the information to decide** *how* **it may be collected.**

— *Q :* **When should we add (delete) a pair** (x, y) **to (from) C_IN or C_OUT ?**

3. **Design a data flow to collect the information. Develop data flow equations for the same.**

— **This should be easy once step 2 is performed.**

SETTING UP DATA FLOW EQUATIONS

Example (Contd.) :

2. The nature of the data flow information

$$\boxed{w:\ z := a + y\ ;} \longleftarrow \quad \mathbf{COPIES}_w = \{\cdots\}$$

Q : **What is the nature of the information in COPIES ?**

This is a matter of definition, hence there is no unique answer. For example,

(a) Let the set COPIES$_w$ be

$$\{(a,b),\ (a,c)\ \cdots\}$$

This situation could arise because the statements $a := b$; and $a := c$; may reach the node along different paths, and the confluence operator $\theta = \cup$. In this case, a can not be substituted by either b or c.

(b) Alternatively, we could define the confluence operator $\theta = \cap$. Now, at most one pair (a, v) may exist in COPIES$_w$ for any a, and existence of such a pair indicates correctness of substituting a by v.

SETTING UP DATA FLOW EQUATIONS

Example (Contd.) :

3. The data flow equations

The information flow is *forwards*. Hence we can use the generic form of data flow equations :

$$\begin{aligned} \mathbf{C\_IN}_i &= \theta_{\forall p}\, \mathbf{C\_OUT}_p \\ \mathbf{C\_OUT}_i &= \mathbf{C\_IN}_i - \mathbf{C\_KILL}_i \,\cup\, \mathbf{C\_GEN}_i \end{aligned}$$

where,

θ is the confluence operator $\cup$ or $\cap$,
C_IN/C_OUT indicate information at entry/exit,
C_KILL indicates information killed in node,
C_GEN indicates information generated in node.

Questions :

1. **If θ is chosen to be $\cup$, then is the C_IN / C_OUT data flow the same as the *reaching definitions* data flow ?**
2. **Define the terms C_GEN and C_KILL for the above data flow problem.**

SETTING UP DATA FLOW EQUATIONS

Example (Contd.) :

3. The data flow equations (Contd.)

For the assignment statement :

$$\boxed{w : \; x \; := \; y \; ;}$$

C_GEN $=$ $\{(x, y)\}$**, and**
C_KILL $=$ $\{(x, h), \ (h, x) \ \forall h\}$**.**

Q : **Explain the pairs included in C_KILL above.**

DATA FLOW ANALYSIS

A solution of the data flow equations consists of an assignment of values to the IN and OUT sets of each node in the program graph.

To be useful for optimisation, the information contained in a solution must satisfy the following conditions :

(a) *Self-consistency :* The values assigned to the different nodes must be mutually consistent (else the solution is incorrect !). Such a consistent solution is known as a *fixed point* of the data flow equations.

(b) *Conservative information :* The data flow information must be conservative in that optimisation using this information should not change the meaning of a program under any circumstances.

(c) *Meaningfulness :* The computed values of the properties should provide meaningful (in fact, maximum) opportunities for optimisation. This is important, since not performing any optimisation is conservative but hardly meaningful in an optimising compiler !

DATA FLOW ANALYSIS

CONSERVATIVE SOLUTION OF DATA FLOW EQUATIONS

During data flow analysis, we consider all paths in the program flow graph (i.e. all *graph theoretic paths*)**. However, during execution, some paths may never be visited, i.e. the set of** *execution paths* **may be different from the set of** *graph theoretic paths***.**

Hence the *computed* **data flow information may be different from the** *actual* **data flow information.**

Q : **Would the optimisation based on the computed data flow information be correct ?**

A: **The optimisation would be correct if the differences between the computed and actual values lie on the safer side, i.e. the differences tend to disallow certain feasible optimisations, but never enable erroneous optimisations.**

Consider available expressions at the statement following an if **statement. Let** $a * b$ **be available prior to the** if **statement, and let the** *then* **branch kill the expression** $a*b$**. Then it is conservative to assume that** $a * b$ **is** *not* **available at the following statement, even if the** *then* **brach is never visited during the program's execution. Use of** $\cap$ **as the confluence operator ensures a conservative solution. Optimisation based on this solution will never be wrong.**

DATA FLOW ANALYSIS

CONSERVATIVE ESTIMATES OF DF PROPERTIES

Making Conservative Assumptions

It is necessary to make conservative assumptions when complete and precise data flow information is not available.

We should use conservative assumptions in

(a) array assignments, viz.

$$a[i] := < exp >;$$

Values of $a[i]$ $\forall$ i are assumed killed.

(b) pointer based assignments, viz.

$$\cdots := a * b;$$
$$*p := \cdots;$$
$$\cdots := a * b;$$

Conservative : $a * b$ is not a CSE !

These assumptions can be relaxed if precise information concerning assignments to i and p is available.

DATA FLOW ANALYSIS

CONSERVATIVE ESTIMATES OF DF PROPERTIES

We should also make conservative assumptions in

(a) procedure calls, viz.

$\cdots := a * b;$

$p(a, x)$;

$\cdots := a * b;$

Conservative : $a * b$ is not a CSE !

(b) procedure bodies, viz.

$procedure\ q(a, b, x);$

$\cdots := a * b;$

$x := < exp >;$

$\cdots := a * b;$

Conservative : $a * b$ is not a CSE !

Conservative estimates make the computed values of data flow properties less precise. This can only be corrected through more analysis, viz. interprocedural analysis, alias analysis, etc.

ALIASING

Identifier v_1 is said to be an alias of identifier v_2, if $v_1,\ v_2$ refer to the same variable/share the same storage location.

Example :

$$\boxed{p\ :=\ \&x;}$$

$*p$ is now an alias of variable x.

The conservativeness associated with an assignment through a pointer variable $*p$ can be relaxed if we determine the set of variables $\{v\}$ such that each v is an alias of $*p$.

Question :

Set up a data flow problem to collect the set of variables whose values can change as a result of the pointer based assignment

$*p := \ldots;$

SOLVING DATA FLOW EQUATIONS

ITERATIVE DATA FLOW ANALYSIS

1. **Set the IN and OUT properties of all nodes in the program flow graph (except the program entry / exit nodes) to some initial values.**
2. **Visit all nodes in the program flow graph and recompute their IN and OUT properties.**
3. **If any changes occur in any IN or OUT properties, then repeat steps 2 and 3.**

Note :

Boundary conditions **hold for the program entry / exit nodes (for forward and backward data flow problems, respectively). Unless interprocedural analysis is performed, no data flow information can be assumed to be available at the boundaries.**

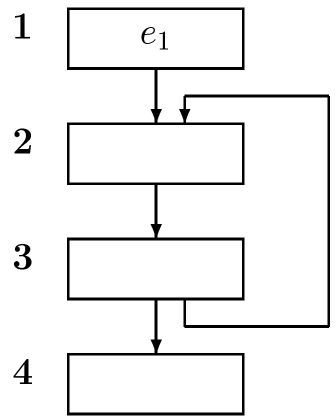
Q : **The solution is a fixed point. Is it unique ?**

SOLVING DATA FLOW EQUATIONS

ITERATIVE DATA FLOW ANALYSIS

The solution of iterative data flow analysis is not unique.

Example :



Let IN$_1$ = {}.

(i) **Let IN = OUT = {e_1} elsewhere.**
Then solution contains IN$_2$ = IN$_3$ = IN$_4$ = {e_1}.

(ii) **Let IN = OUT = {} elsewhere.**
Then solution contains IN$_2$ = IN$_3$ = IN$_4$ = {}.

SOLVING DATA FLOW EQUATIONS

ITERATIVE DATA FLOW ANALYSIS

AVAILABLE EXPRESSIONS

/* Initialisations */
$\mathbf{AVIN}_{n_0}$:= {}; $\mathbf{AVOUT}_{n_0}$:= $\mathbf{AVGEN}_{n_0}$;
$\forall\ i \in N - n_0$ **$\mathbf{AVOUT}_i$:= U − $\mathbf{AVKILL}_i \cup$ $\mathbf{AVGEN}_i$;**
/* U is the universal set of expressions */

```
/* Iteration */
change := true;
while change do begin
        change := false;
        ∀ i ∈ N − n0 do begin
                AVIN_i := ∩_∀p AVOUT_p;
                oldout_i := AVOUT_i
                AVOUT_i := AVGEN_i ∪ (AVIN_i − AVKILL_i);
                if AVOUT_i ≠ oldout then change := true;
                end;
        end;
```

Q : **What is the complexity of iterative df analysis ?**

A: $O(n)$ **iterations, where n is the number of nodes.**

SOLVING DATA FLOW EQUATIONS

MEANINGFUL SOLUTION OF DATA FLOW EQUATIONS

For the problem of available expressions, initialising IN and OUT properties of all nodes to {} leads to a trivial solution of the data flow equations. We must avoid trivial solutions and try to obtain the most meaningful solution, i.e. the largest possible solution to the equations (also called the *maximum fixed point* **(MFP)).**

Initialisation for the Maximum Fixed Point

(a) For $\cap$ problems initialise all nodes to the universal set.

(b) For $\cup$ problems initialise all nodes to {}.

DATA FLOW ANALYSIS

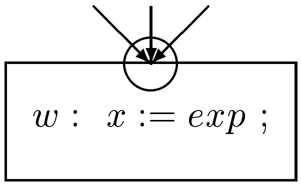
CONVERGENCE OF ITERATIVE DATA FLOW ANALYSIS

Q : **Is the process of iterative data flow analysis guaranteed to converge ?**

A: **Depends on** *monotonicity* **of the data flow.**

The Monotonicity Condition

$$\mu \leq \nu \textbf{ implies } f(\mu) \leq f(\nu) \quad \forall\ \mu,\ \nu \textit{ and } f$$



$$\textbf{AVIN}_i = \cap_{\forall p}\ \textbf{AVOUT}_p$$

$$\textbf{AVOUT}_i = \textbf{AVGEN}_i \cup (\textbf{AVIN}_i - \textbf{AVKILL}_i)$$

- **Let ν be the initial value of AVIN$_i$, and let μ be the value of AVIN$_i$ after the first iteration. Hence $\mu \leq \nu$.**
- **Since $\mu \leq \nu$, from monotonicity AVOUT$_i$ after the first iteration $\leq$ the initial value of AVOUT$_i$.**
- **Hence the values assumed by AVIN$_i$ (AVOUT$_i$) form a non-increasing sequence. This guarantees convergence.**

Note : **Most practical data flow problems are monotone !**

SOLVING DATA FLOW EQUATIONS

COMPLEXITY OF ITERATIVE DATA FLOW ANALYSIS

Depth first numbering

A depth first numbering (dfn) of the nodes of a graph is the reverse of the order in which we last visit each node in a pre-order traversal of the graph.

Depth first numbering has the following properties :

- $\forall i \in dominators(j), dfn(i) < dfn(j)$,
- $\forall$ *forward edges* (i, j), $dfn(i) < dfn(j)$,

In a *reducible flow graph* **(Refer to A-S-U for definition), an edge** (i, j) **is a loop forming edge (also called a** *back edge***) if** $dfn(j) < dfn(i)$.

Iterative analysis in depth first order :

Fewer iterations are required if we visit the nodes of the graph in depth first order (or reverse depth first order) during each iteration, rather than in some random order (example on next transparency).

SOLVING DATA FLOW EQUATIONS

COMPLEXITY OF ITERATIVE DATA FLOW ANALYSIS

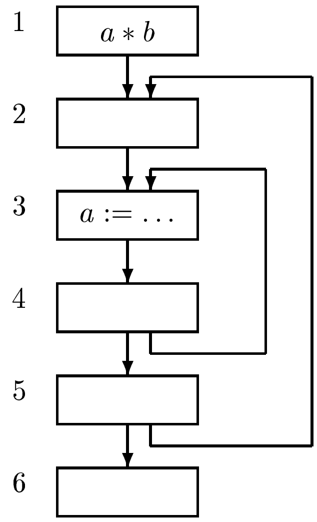
Depth of a Program Flow Graph (d)

Depth (d) **is the maximum number of back edges in any acyclic path in the program flow graph (Note that this is not the same as the nesting depth !).**

Complexity of iterative analysis

When the nodes of a graph are visited in a depth first order (reverse depth first order) for a forward data flow problem (backward data flow problem), $d + 1$ **iterations are sufficient to reach a fixed point.**

Example :



depth = 1

nesting depth = 2

SOLVING DATA FLOW EQUATIONS

COMPLEXITY OF ITERATIVE DATA FLOW ANALYSIS

$d+1$ iterations are adequate for data flow analysis

- **For a forward data flow problem, we visit the nodes in the increasing order by depth first numbers. If there are no loops in the program (i.e. $d = 0$), the data flow information computed in the first iteration would be a fixed point of the data flow equations.**

- **If a single loop exists in the program, $d = 1$. Now, the new value of the loop exit node computed in the first iteration can influence the property of the loop entry node. This can only be achieved in the next iteration. Hence 2 iterations are necessary.**

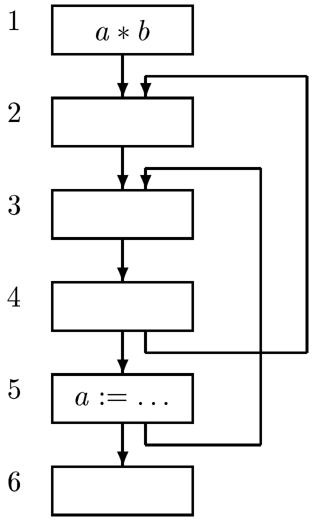
- **If another back-edge starts on some node of the loop, such that the loop formed by it is *not* contained in the outer loop, then yet another iteration is required (see the next transparency for an example).**

SOLVING DATA FLOW EQUATIONS

COMPLEXITY OF ITERATIVE DATA FLOW ANALYSIS

$d + 1$ *iterations are adequate for data flow analysis*

Example



depth = 2

The effect of the assignment $a := \ldots$ is felt on the availability of $a * b$ at the entry of block 2 only in the third iteration.

REGISTER ASSIGNMENT & ALLOCATION

A note on terminology

Register *assignment* **and** *allocation* **are two distinct phases in the work aimed at making effective utilisation of registers of the target machine (by reducing the number of Load/Store instructions). The distinction between the two terms has not always been maintained in the literature.**

We will, however, make a distinction between the two terms and use them with the meanings described in the following two transparencies.

REGISTER ASSIGNMENT & ALLOCATION

Register assignment

- **Managing the use of *hypothetical* registers.**
  - **A hypothetical register is assumed to be available for every data item / expression. During register assignment, we decide where to place Load / Store instructions so that the value of the data item / expression is available in a register at every usage point, and is available in the memory cell allocated for that data item / expression at all other points.**
  - **An unbounded number of hepothetical registers is assumed to be available for the purpose of assignment.**

REGISTER ASSIGNMENT & ALLOCATION

Register allocation

- **Managing the use of *real* registers in a target machine.**
  - **The use of hypothetical registers is mapped into the use of real registers existing in the target machine.**
  - **The motivation is to hold frequently used data values in registers instead of memory locations in the interests of execution efficiency of the target program.**

Scope :

(i) *Local Register Allocation* **: Allocation of registers to data items / expressions within a basic block.**

(ii) *Global Register Allocation* **: Allocation of registers to data items / expressions over regions of a program.**

LOCAL REGISTER ALLOCATION

... allocation of registers to date items / expressions within a basic block (i.e. *in straight line code***).**

Note the following points in this context

- **Code generation precedes register allocation.**
- **Register assignment may have been performed during code generation, e.g. the code generation algorithm may assume the presence of a large number of registers, possibly one for every data item / expression.**
- **Some register allocation may have been performed during code generation, e.g. the code generation algorithm may save / restore partial results from registers when it runs out of registers to use for expression evaluation. (The Aho-Johnson and Sethi-Ullman algorithms even try to do this 'optimally'.)**
- **Register assignment / allocation performed for a source statement may have to be modified to improve register usage within a basic block, e.g. if a variable** *var* **is used in two consecutive statements, holding** *var* **in a register accross the statements would save a Load instruction !**

LOCAL REGISTER ALLOCATION

Register Reference String

... **represents the sequence in which hypothetical registers are used in a basic block. This provides the basis for allocation decisions. (** *Note :* **We use r_1, r_2, etc. to represent hypothetical registers, and R_1, R_2, etc. to represent machine, i.e.** *real*, **registers.)**

Example

$$\boxed{r_1, r_2^*, r_3, r_1, r_2}$$

Here, r_2^* indicates that the hypothetical register is referenced-and-modified in this step.

LOCAL REGISTER ALLOCATION

Managing the registers over a basic block

When all machine registers hold useful values and a new register is required for calculations, we free one of the machine registers (say, R_i). This is called *pre-emption* of a register.

Let R_i contain the value represented by a hypothetical register r_j.

- If r_j (i.e. the data item represented by it) has been modified since it was last loaded from the memory cell, then we need to store its value in the memory cell while freeing it for another purpose.

  (a) Belady proposed that the value whose next use lies farthest in the register reference string should be pre-empted from the machine register.

  (b) Kennedy, Horowitz, Fischer differentiate between

  - a reference of a hypothetical register, and
  - a reference-and-modification of a hypothetical register

  during pre-emption, since the latter requires a Store while the former does not. They consider alternative pre-emption decisions and compute the total cost of the register allocation for the basic block. The lowest cost alternative is selected.

LOCAL REGISTER ALLOCATION

Example :

Consider local register allocation in a 2-register machine for the reference string :

$$\boxed{r_1, r_2^*, r_3, r_1, r_2}$$

Belady's Algorithm : 'Maximum distance' pre-emption.

Register reference	Machine Register R_1	R_2	Cost
r_1	r_1	—	**1**
r_2^*	r_1	r_2^*	**1**
r_3	r_1	r_3	**2**
r_1	r_1	r_3	**0**
r_2	r_1	r_2	**1**

total cost = 5

Note :

Load and Store instructions are introduced at appropriate points, i.e. whenever the contents of a machine register are changed. Both are assumed to cost 1 unit each.

LOCAL REGISTER ALLOCATION

Example :

Consider local register allocation in a 2-register machine for the reference string :

$$\boxed{r_1, r_2^*, r_3, r_1, r_2}$$

Kennedy *et al* **Algorithm :**

Register reference	Machine Register R_1	R_2	Cost
r_1	r_1	—	**1**
r_2^*	r_1	r_2^*	**1**
r_3	r_3	r_2	**1**
r_1	r_1	r_2	**1**
r_2	r_1	r_2	**0**

total cost = 4

Note :

Load and Store instructions are introduced at appropriate points, i.e. whenever the contents of a machine register are changed. Both are assumed to cost 1 unit each.

Comparison of algorithms

The algorithm by Kennedy *et al* is an optimal algorithm, in that it always finds the least cost allocation. However, this algorithm is computationally expensive since it considers the alternative pre-emption decisions and selects the best one.

Belady's algorithm is not an optimal algorithm in that it does not guarantee least cost allocation. However, it it computationally efficient.

The manner of variation of the register allocation expenses with the size of the register reference string is very important from the viewpoint of compilation efficiency. As programs are continually increasing in size, it is useful that an algorithm used in the compiler should be *linear* in nature. If this is not possible, it should at least *not* be exponential in its behaviour. Belady's algorithm is more practical from this viewpoint.

GLOBAL REGISTER ALLOCATION

Notation :

R	:	**Set of machine registers**
D	:	**Set of data items**
$\mathbf{R}^k$	:	**Set of registers allocated to** $d_k \in$ **D over the entire program**
$\mathbf{D}_l$	:	**Set of data items to which** $r^l \in$ **R is allocated over the program**
$\mid \ldots \mid$	:	**Cardinality of a set**

For high profits, register utilisation should be maximised. Any allocation of registers in R to data items in D must satisfy the following conditions :

(i) *Non-interference :* **Data items to which the same register has been allocated should not be simultaneously live at any point, i.e.** $|\mathbf{D}_l| = 1 \; \forall l$.

(ii) *Consistency :* **At most one register should be allocated to a data item at any program point.**

GLOBAL REGISTER ALLOCATION

Nature of global allocation

(a) *one-one allocation* : **Each machine register is allocated exclusively to one data value and a value is allocated to at most one register.**

(b) *many-one allocation* : **At least one machine register is shared between more than one data values.**

(c) *many-few allocation* : **At least one register is shared between more than one data values, and at least one data value is resident in different registers in diferent regions of the program.**

Q : **Characterise many-few allocation formally.**

GLOBAL REGISTER ALLOCATION

Allocation of registers across basic block boundaries.

Preliminaries

When a value is allocated to a machine register, it is necessary that :

(i) The value is available in a register at all points in the program where it is used (i.e. at all its reference/definition points), and

(ii) The value is available in a memory location at all points in the program where it is not contained in a register.

Load and Store instructions have to be inserted at strategic points in the program to ensure this.

Live range of a value

The program region over which a value x needs to reside in a register is called the *live range* **of the value x (i.e. lr_x). A live range is often represented as a set of basic blocks $\{b\}$ of a program.**

Identification of the live range of a value constitutes *register assignment* **for the value.**

GLOBAL REGISTER ASSIGNMENT

The profit of a live range

The *profit* of a live range lr_x is the number of executions of Load/Store instructions of x eliminated by holding its value in a register.

Example : **Static estimation of profits**

Consider a live range lr consisting of a set of basic blocks $\{b\}$. We have

$$MP_{lr} = \sum_{b \in lr} \#occ_b \cdot w^{nb}$$
$$P_{lr} = MP_{lr} - \text{cost of inserted Loads/Stores.}$$

where

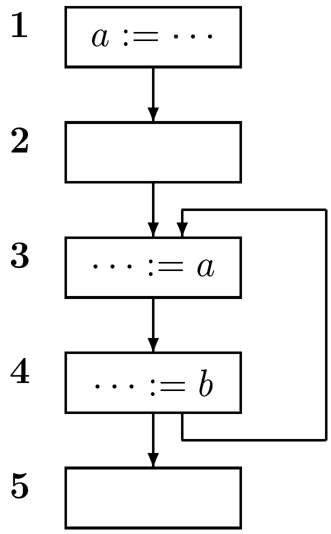
MP_{lr} **is the maximum profit for live range** lr,
P_{lr} **is the realisable profit for live range** lr,
$\#occ_b$ **is the number of Loads/Stores in** b,
nb **is the static nesting level of** b, **and**
w **is the nesting weightage (usually 5 or 10). (w^{nb} indicates how many times** b **may execute in a run)**

Profit of a register allocation

The *profit of a register allocation* is the sum of the profits of all live ranges to which registers have been allocated.

GLOBAL REGISTER ASSIGNMENT

Live Range Examples



Live range of a : $\{1, 2, 3, 4\}$

— **Store in block 1.**

— MP_a **= 11,** P_a **= 10 for** w **= 10.**

Live range of b : $\{2, 3, 4\}$

— **Load at exit of block 2.**

— MP_b **= 10,** P_b **= 9 for** w **= 10.**

Many methods for identifying the live range of a data item have been designed. We see one such method in the following.

GLOBAL REGISTER ASSIGNMENT

METHODS OF LIVE RANGE IDENTIFICATION

Chow-Hennessy Approach

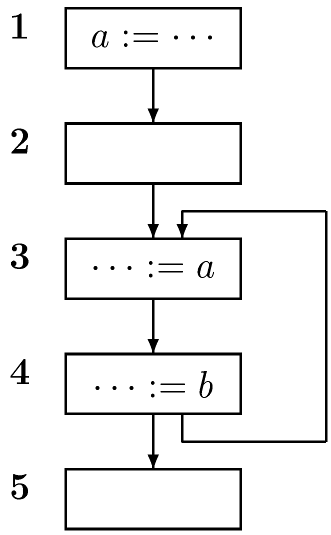
A basic block of the program belongs to the live range of a value if the value is *live* within that basic block and a reference or a definition of the value *reaches* it. The live range is the set of such basic blocks.

1. *Value is live :* This implies that the value is used along some path through this block, hence it is meaningful to hold it in a register.
2. *A reference is reaching and the value is live :* This implies that the value is currently in a register (it would have been loaded at the reference that is reaching), and is required along some path through this block.
3. *A defintion is reaching and the value is live :* The value is currently in a register (it would have been put there by the definition that is reaching), and is required along some path through this block.

METHODS OF LIVE RANGE IDENTIFICATION

Chow-Hennessy Approach (Contd.)

Example :



Live range of a : $\{1, 2, 3, 4\}$

Live range of b : $\{3, 4\}$

Load instructions are inserted (if necessary) in all entry blocks of the live range. Store instructions are inserted (where necessary) in all exit blocks of the live range.

Note that block 2 is not contained in the live range of b.

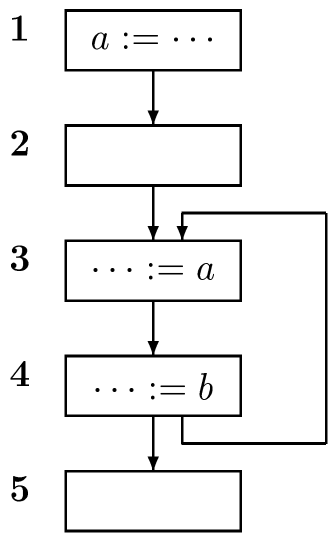
THE BASIS FOR REGISTER ALLOCATION

INTERFERENCE OF LIVE RANGES

If some basic block b of the program belongs to live ranges lr_1 and lr_2, then live ranges lr_1, lr_2 are said to interfere (in that block).

The same machine register can not be allocated to interfering live ranges.

Example



Live range of a : $\{1, 2, 3, 4\}$
Live range of b : $\{3, 4\}$

Live ranges lr_a, lr_b interfere in blocks 3 and 4, hence the same register can not be allocated to variables a and b.

GLOBAL REGISTER ALLOCATION

REGISTER INTERFERENCE GRAPH

Register Interference graph is an undirected graph

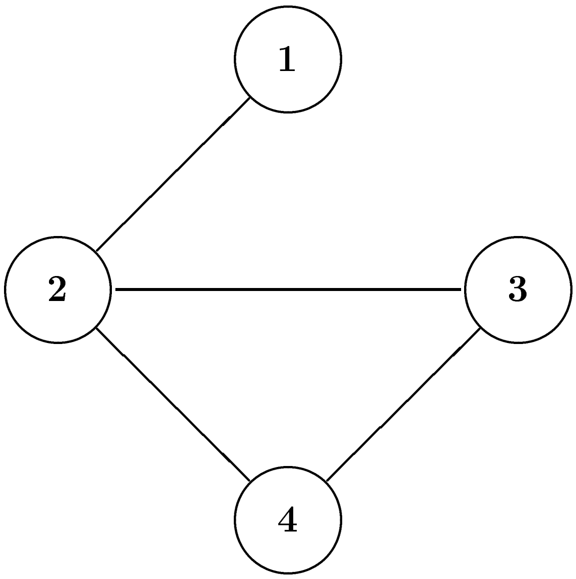
$$IG = (L, IE)$$

where

(i) L **is the set of live ranges for the values which are candidates for register allocation**

(ii) IE **is the set of edges** (lr_i, lr_j) **such that live ranges** lr_i **and** lr_j **interfere, i.e.** $lr_i \cap lr_j \neq \phi$**.**

Example



GLOBAL REGISTER ALLOCATION

THE GRAPH COLOURING APPROACH

Graph Colouring

The problem of graph colouring is defined as :

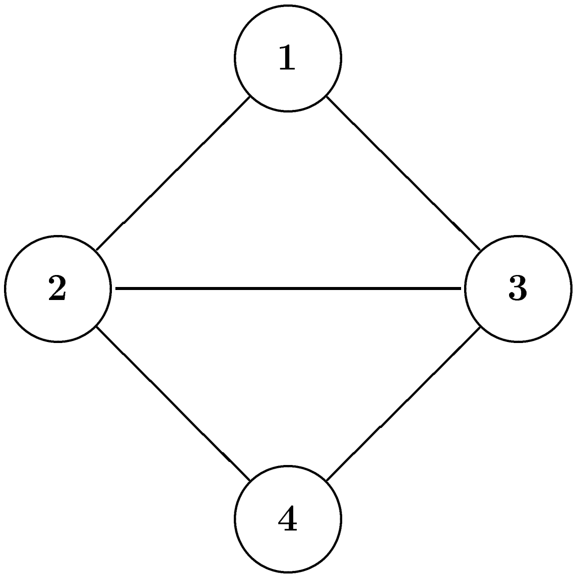
(a) give different colours to nodes lr_i, lr_j if the edge (lr_i, lr_j) exists in IG,

(b) use minimum number of colours

Register allocation

$\ldots$ **can be looked upon as a colouring of IG !?**

Example

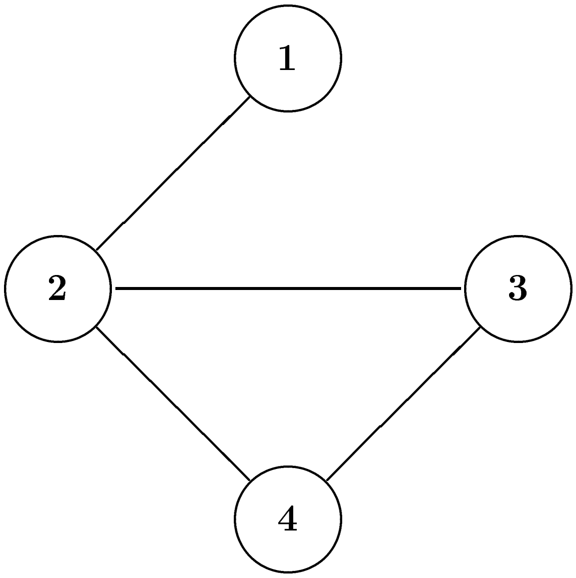


GLOBAL REGISTER ALLOCATION

ONE-ONE & MANY-ONE ALLOCATION

. . . is feasible when # colours $\leq$ # registers.

Example



Allocation for a 3 register machine :

register 1 : live ranges 1, 4
register 2 : live range 2
register 3 : live range 3

GLOBAL REGISTER ALLOCATION

MANY-FEW ALLOCATION

Q : **What if # colours > # registers ?**

Constrained live ranges

A constrained live range is a live range whose degree in IG $\geq r$, the number of registers.

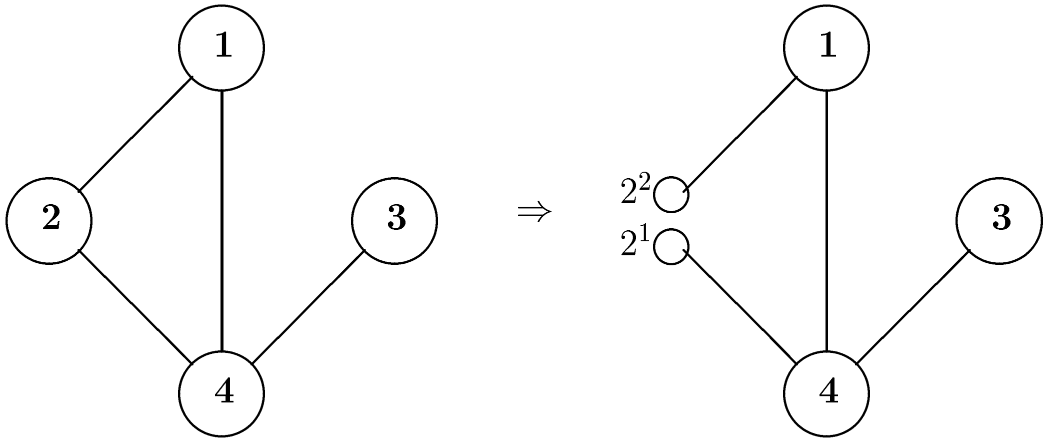
(a) An unconstrained live range can always be coloured. It may or may not be possible to colour a constrained one. (Refer to interference graph on previous transparency.)

(b) For constrained live ranges, one-one or many-one allocation may not be feasible. In that case, *live range splitting* may be used to perform many-few allocation.

GLOBAL REGISTER ALLOCATION

LIVE RANGE SPLITTING

Example



Allocation for a 2 register machine :

Live ranges 1,2 and 4 are constrained. Live range 2 can be split to facilitate allocation. Hence :

register 1 : live ranges 1, 3, 2^1
register 2 : live range 4, 2^2

where 2^1 **and** 2^2 **are parts of live range 2, which do not interfere with nodes 1 and 4 respectively.**

PRACTICAL LIVE RANGE SPLITTING

In practice, it is not always possible to find live range partitions as shown in previous transparency. Hence, live range splitting is performed as follows :

(a) A colourable live range partition is found. This partition of the live range has a degree $\leq n$, where n is the number of machine registers.

(b) The remainder of the live range is represented by another node in the interference graph. This node may have a degree as much as the original live range. (Hence, it may be necessary to split this partition of the live range further.)

In the previous transparency, live range 2^2 may be identified so that it only interferes with live range 1. Live range 2^1 may interfere with nodes 1 and 4. Hence, it can not be allocated a colour without further splitting.

GLOBAL REGISTER ALLOCATION

CHOW-HENNESSY'S PRIORITY BASED COLOURING

- **Priority P_{lr} = (profits / # blocks in live range)**
- *Forbidden set* **for each node is the set of colours which have been given to neighbouring nodes.**

Algorithm outline

1. **Separate constrained and unconstrained live ranges.**
2. **While a constrained live range and an allocatable register exists**
   (a) **Compute the priorities P_{lr} for all live ranges, if not already done.**
   (b) **Find the lr with highest P_{lr}**
      - **Colour this lr with a colour not present in its forbidden set.**
      - **Update the forbidden set for the neighbours of this live range in IG.**
      - **Split the neighbours of this lr, if necessary. (This is done when the forbidden set of a neighbour is 'full', i.e. it contains all possible colours.)**
3. **Colour the unconstrained live ranges.**

Q : **Should we identify constrained live ranges again in step 2? Explain why.**

GRADED EXERCISES

1. Study the program flow graph given below and indicate whether any of the following optimising transformations can be applied to it

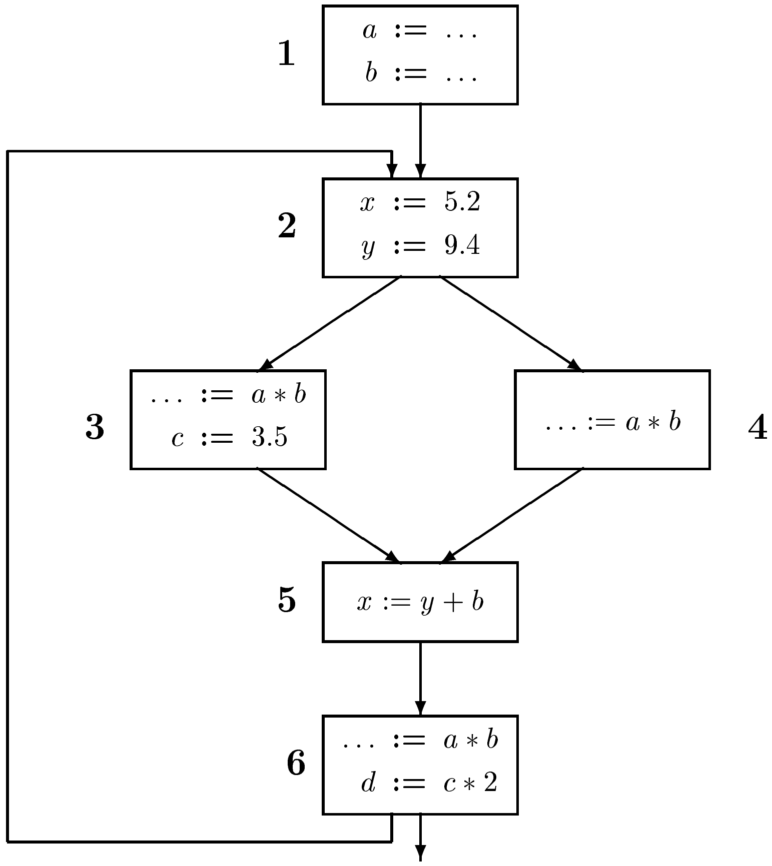
   (a) common subexpression elimination

   (b) elimination of dead code

   (c) constant propagation, and

   (d) frequency reduction

   Clearly justify your answers.



GRADED EXERCISES

2. Specify the necessary and sufficient conditions for performing
   (a) Constant propagation
   (b) dead code elimination, and
   (c) loop optimisation
3. Develop complete algorithms for the following optimisations
   (a) common subexpression elimination
   (b) elimination of dead code
   (c) constant propagation, and
   (d) frequency reduction

   Apply these algorithms to the program flow graph of question 1, and compare the answers with your own answers in question 1.
4. Write a note justifying the need for $d+1$ iterations, where d is the depth of a graph, for the iterative solution of a data flow problem.

GRADED EXERCISES

5. Given an assignment statement of the form

$a := b;$

copy propagation **implies substituting** b **for** a **at every usage point of** a **reached by the definition** $a := b;$**.**

(a) Develop a complete algorithm for copy propagation. (*Hint :* Refer to the discussion of copy propagation in this module.)

(b) Can copy propagation be performed transitively ? For example, in

$a := b;$

$c := a;$

Can c **be replaced by** b **? If so, explain how you will modify your algorithm to perform this enhanced copy propagation.**