UNIT–V

MachineIndependentOptimization.TheprinciplesourcesofOptimization,peepholeOptimization,Introductionto Date flowAnalysis.

UNIT5

MACHINEINDEPENDENTOPTIMIZATION

Eliminationofunnecessary instructionsin objectcode, orthereplacementof onesequenceofinstructions by a faster sequence of instructions that does the same thing is usually called "codeimprovement"or "codeoptimization."

Optimizationsareclassifiedintotwocategories.

1. Machineindependentoptimizations:

   Machineindependentoptimizationsareprogramtransformationsthatimprovethetargetcode without taking intoconsideration any properties ofthe target machine

2. Machinedependantoptimizations:

   Machinedependantoptimizationsarebasedonregisterallocationandutilizationofspecialmachine-instructionsequences.

ThePrincipalSourcesofOptimization

A transformation of a program is called local if it can be performed by looking only at thestatementsinabasicblock;otherwise,itiscalledglobal.Manytransformationscanbeperformedatboththelocal and globallevels.

Function-Preserving Transformations: There are a number of ways in which a compiler can improveaprogram without changing thefunction it computes.

: Common sub expression eliminationCopypropagation, Dead-code eliminationConstantfolding

CommonSubexpressionselimination:

An occurrence of an expression E is called a common sub-expression if E was previouslycomputed, and the values of variables in E have not changed since the previous computation. We canavoidrecomputing theexpression if wecan usethepreviously computed value.

- Forexample

  t1: =
  4*it2: = a
  [t1]t3: =
  4*jt4: =
  4*it5:=n
  t6:=b[t4]+t5

Theabove codecanbeoptimized usingthecommon sub-expressioneliminationas

t1:=4*it2:
= a [t1]t3:
=4*jt5:= n
t6:=b[t1]+t5

Thecommonsubexpressiont4:=4*iiseliminatedasitscomputationisalreadyint1andthevalueofiisnot been changedfromdefinition to use.

CopyPropagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind thecopy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.Copypropagation meansuseof onevariable instead of another.

• Forexample:

x=Pi;
A=x*r*r;

Theoptimizationusing copypropagationcan bedoneasfollows:A=Pi*r*r;Herethe variablexis eliminated

Dead-CodeEliminations:

Avariableisliveatapointinaprogramifitsvaluecanbeusedsubsequently;otherwise,itisdeadat that point.

Example:

```
i=0;
if(i==1)
{
a=b+5;
}
```

Here, ‘if’statementisdeadcodebecausethisconditionwillnevergetsatisfied.

Constantfolding:

Deducing at compile time that the value of an expression is a constant and using the constantinstead is known as constant folding. One advantage of copy propagation is that it often turns the copystatementinto dead code.

Forexample,

a=3.14157/2 can be replaced bya=1.570

LoopOptimizations:

Inloops, especiallyinthe innerloops, programstend tospendthe bulk of theirtime. Therunning time of a program may be improved if the number of instructions in an inner loop is decreased,evenif weincreasetheamount of codeoutsidethat loop.

Threetechniques areimportant forloop optimization:

1. Codemotion, which moves codeoutsidealoop;
2. Induction-variable elimination, which we apply to replace variables from inner loop.3.Reductioninstrength,whichreplaces expensiveoperationbyacheaperone,suchas a multiplicationbyanaddition.

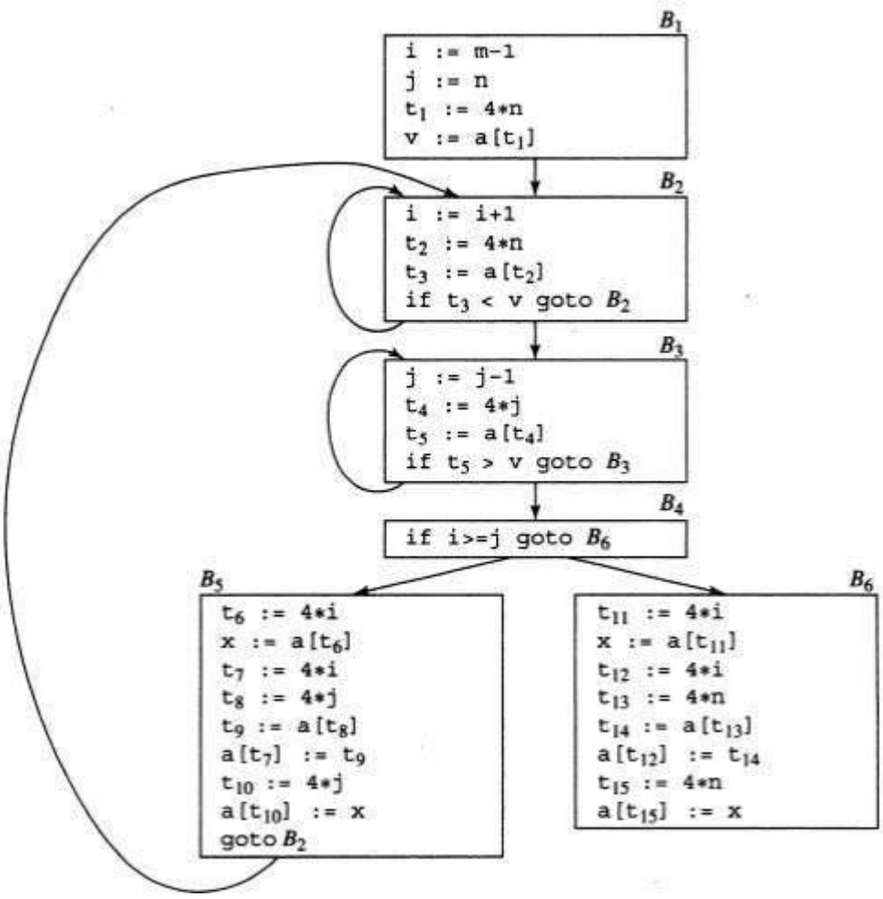

Fig.5.2 Flow graph

CodeMotion:

This transformation takes an expression that yields the same result independent of the number oftimes a loop is executed (a loop-invariant computation) and places the expression before the loop. Notethat the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluationoflimit-2 is a loop-invariant computation in thefollowing while-statement:

while(i <=limit-2)

Codemotion willresultin theequivalent of

```
t= limit-2;
while(i<=t)/*statementdoesnot changelimitort*/
```

InductionVariables:

Loops are usually processed inside out. For example consider the loop around B3. Note that thevalues of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4because4*j is assigned to t4. Suchidentifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all butone, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannotgetrid of either j ort4completely; t4 is usedin B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variableelimination. Eventuallyj willbeeliminated whenthe outerloop ofB2-B5 isconsidered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is notchanged elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1therelationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. Theonly problem is that t4 does not have a value when we enter block B3 for the first time. Since we mustmaintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end oftheblock wherej itselfis initialized, shown bythedashed addition toblock B1 in Fig.5.3.

Thereplacementofamultiplicationbyasubtractionwillspeeduptheobjectcodeifmultiplicationtakes moretime than addition or subtraction

ReductionInStrength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the targetmachine. Certain machine instructions are considerably cheaper than others and can often be used asspecial cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*xthan as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two ischeapertoimplementasashift.Floating-pointdivisionbyaconstantcanbeimplementedasmultiplicationby aconstant, which may becheaper.

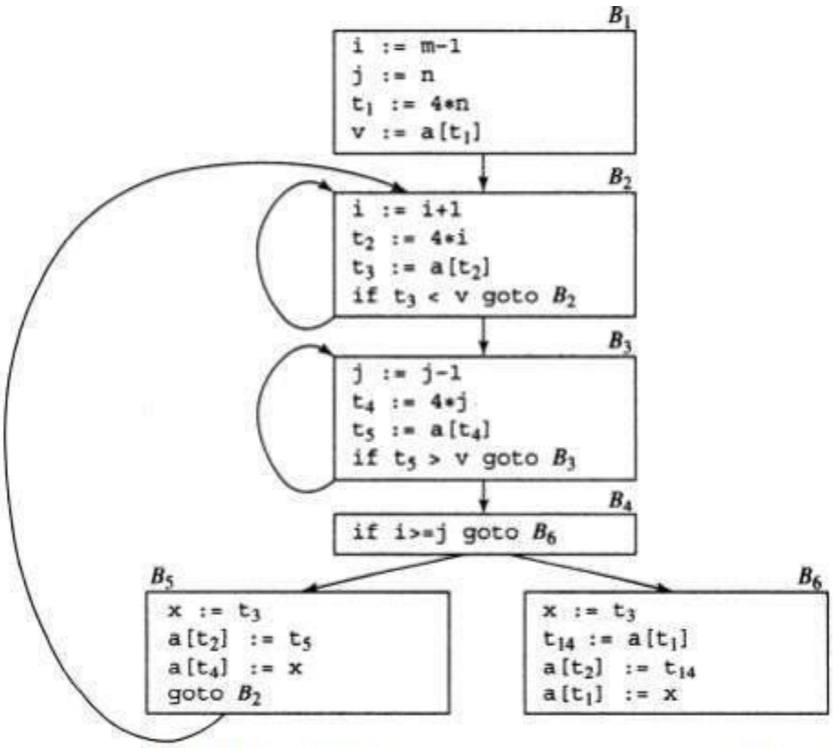

Fig. 5.3 B5 and B6 after common subexpression elimination

Fig.5.3B5and B6aftercommonsubexpressionelimination

PEEPHOLEOPTIMIZATION

A statement-by-statement code-generations strategy often produces target code that containsredundant instructions and suboptimal constructs. The quality of such target code can be improved byapplying"optimizing" transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, Amethod for trying to improving the performance of the target program by examining a short sequence oftarget instructions (called the peephole) and replacing these instructions by a shorter or faster sequence,wheneverpossible.

Thepeepholeisasmall,moving windowonthetargetprogram.

Characteristics of peephole

optimizations: Redundant-
instructions eliminationFlow-of-
control
optimizationsAlgebraicsimplification
s
UseofmachineidiomsU
nreachablecode

Redundant-instructionselimination

seetheinstructionssequence

(1) MOVR0,a

(2) MOVa,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value ofa is already in register R0.If (2) had a label we could not be sure that (1) was alwaysexecutedimmediatelybefore(2)and so wecould not remove(2).

UnreachableCode:

Another opportunity for peephole optimizations is the removal of unreachable instructions. Anunlabeledinstructionimmediatelyfollowinganunconditionaljumpmayberemoved.Thisoperationcan be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a largeprogram may have within it certain segments that are executed only if a variable debug is 1. In C, thesourcecodemight look like:

```
#definedebug0
….

If(debug ) {
Printdebugginginformation

}
```

Intheintermediate representationsthe if-statementmaybetranslatedas:

```
Ifdebug =1 goto L1 goto L2

L1:printdebugginginformationL2:..... (a)
```

Oneobviouspeepholeoptimizationistoeliminatejumpsoverjumps.Thusnomatterwhatthevalueofdebug; (a) can bereplaced by:

```
Ifdebug≠1 gotoL2
Printdebugginginformation
L2:.....(b)

Ifdebug≠0 gotoL2
Printdebugginginformation
L2:.....(c)
```

Astheargumentof thestatement of(c) evaluatesto aconstant trueit canbereplaced

BygotoL2.Thenallthestatementthatprintdebuggingaidsaremanifestlyunreachableandcanbeelimina tedoneat a time.

Flows-Of-ControlOptimizations:

Theunnecessaryjumpscanbeeliminatedineithertheintermediatecodeorthetargetcodebythefollowin g types ofpeephole optimizations. Wecan replacethe jump sequence

gotoL1
….

L1:gotoL2 (d)

bythesequence

gotoL2
….

L1:gotoL2

IftherearenownojumpstoL1,thenitmaybepossibletoeliminatethestatementL1:gotoL2providedit is preceded by an unconditional jump.Similarly, thesequence

if a< b gotoL1
….

L1:gotoL2(e)

canbereplacedby

Ifa < b goto L2

….

L1:goto L2

Ø Finally,supposethereisonlyonejumptoL1andL1isprecededbyanunconditionalgoto. Thenthesequence

gotoL1

L1:ifa < bgotoL2 (f)L3:

maybereplacedby

If a < b goto
L2gotoL3

…….

L3:

Whilethenumberofinstructionsin(e)and(f)isthesame,wesometimesskiptheunconditionaljumpin(f), but never in (e).Thus (f) is superiorto (e) inexecution time

AlgebraicSimplification:

There is no end to the amount of algebraic simplification that can be attempted through peepholeoptimization. Only a few algebraic identities occur frequently enoughthat it is worth consideringimplementingthem. For example, statements suchas

x := x+0
orx := x *
1

areoftenproducedbystraightforwardintermediatecode-generationalgorithms,andtheycanbeeliminatedeasily through peephole optimization.

ReductioninStrength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the targetmachine. Certain machine instructions are considerably cheaper than others and can often be used asspecialcases of moreexpensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiationroutine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift.Floating-point division by a constant can be implemented as multiplication by a constant, which may becheaper.

$X^2 \rightarrow X*X$

Useof MachineIdioms:

The target machine may have hardware instructions to implement certain specific operationsefficiently. For example, some machines have auto-increment and auto-decrement addressing modes.Theseaddorsubtractonefromanoperandbeforeorafterusingitsvalue.Theuseofthesemodesgreatly improves the quality of code when pushing or popping a stack, as in parameter passing. Thesemodescan also be used in codeforstatements like i : =i+1.

i:=i+1→i++

i:=i-1 → i--

IntroductiontoDateflowAnalysis.

1 TheData-FlowAbstraction

2 TheData-FlowAnalysisSchema

3 Data-

FlowSchemasonBasicBlocks4Reachin

g Definitions

5 Live-VariableAnalysis

6 AvailableExpressions

"Data-flow analysis" refers to a body of techniques that derive information about the flow of data alongprogramexecution paths.

1. TheData-FlowAbstraction

The execution of a program can be viewed as a series of transformations of the program state,which consists of the values of all the variables in the program. Each execution of an intermediate-codestatement transforms an input state to a new output state. The input state is associated with the *programpointbefore* the statementand theoutputstateis associated withthe*program pointafter* the statement.

When we analyze the behavior of a program, we must consider all the possible sequences ofprogram points ("paths") through a flow graph that the program execution can take. We then extract,from the possible program states at each point, the information we need for the particular data-flowanalysis problem we want to solve. In more complex analyses, we must consider paths that jump amongtheflow graphs for various procedures, ascalls and returns areexecuted.

Within one basic block, the program point after a statement is the same as the program pointbeforethenext statement.

IfthereisanedgefromblockB1 toblockB2,thentheprogrampointafterthelaststatementofB1may befollowed immediatelyby the program point beforethefirst statement of B_2.

Thus, we may define anexecution path (or just path) from point pito point pn tobea sequenceofpointspi,p2,... ,pn such thatforeach i =1,2, ... ,n -1, either

1. Pi is the point immediately preceding a statement and $pi_{+}i$ is the point immediately followingthatsame statement, or

2. piis theend ofsomeblock and p_{i+1} is thebeginning ofasuccessor block.

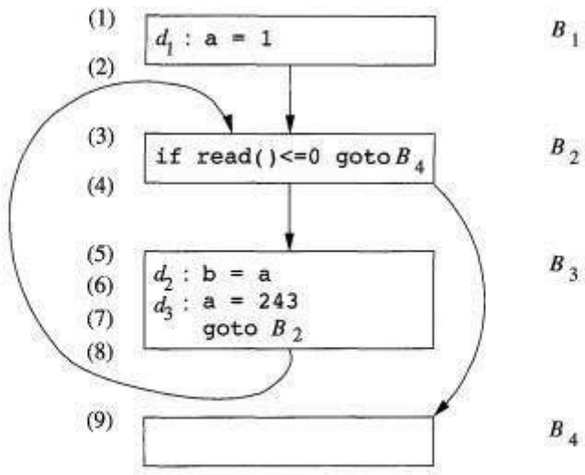

Figure 9.12: Example program illustrating the data-flow abstraction

.Indata-flowanalysis,wedonotdistinguishamongthepathstakentoreachaprogrampoint.Moreover, we do not keep track of entire states; rather, we abstract out certain details, keeping only thedata we need for the purpose of the analysis. Two examples will illustrate how the same program statesmaylead to different information abstractedat a point.

1. To help users debug their programs, we may wish to find out what are all the values a variable mayhave ata program point, and where these values may be defined. For instance, we may summarize allthe program states at point (5) by saying that the value of *a* is one of {1,243}, and that it may be definedbyoneof**{^1,^3}.**Thedefinitionsthat*may*reachaprogrampointalongsomepathareknownas*reaching definitions.*

2. Suppose, instead, we are interested in implementing constant folding. If a use of the variable *x* isreachedbyonlyonedefinition,andthatdefinitionassignsaconstantto *x*, thenwecansimplyreplace *x* by the constant. If, on the other hand, severaldefinitions of *x* may reach a single programpoint, then we cannot perform constant folding on *x*. Thus, for constant folding we wish to find thosedefinitions that are the unique definition of their variable to reach a given program point, no matterwhichexecutionpathistaken.Forpoint(5)ofFig.9.12,thereisnodefinitionthat *must* bethedefinition of *a* at that point, so this set is empty for *a* at point (5). Even if a variable has a uniquedefinitionata point,thatdefinitionmustassigna constanttothevariable.Thus,we maysimplydescribe certain variables as "not a constant," instead of collecting all their possible values or all theirpossibledefinitions.

2. TheData-FlowAnalysisSchema

, we associate with every program point a data-flow value that represents an abstraction of the set of allpossible program states that can be observed for that point.The set of possible data-flow values is thedomain for this application. For example, the domain of data-flow values for reaching definitions is thesetof all subsets of definitions in the program.

A particular data-flow value is a set of definitions, and we want to associate with each point in theprogramthe exactsetof definitionsthatcanreachthatpoint.Asdiscussedabove,the choice ofabstraction depends on the goal of the analysis; to be efficient, we only keep track of information that isrelevant.

Denotethedata-flowvaluesbeforeandaftereachstatementsbyIN[S]andOUT[s],respectively.Thedata-flow problem is to find a solution to a set of constraints on the IN[S]'S and OUT[s]'s, for allstatements s. There are two sets of constraints: those based on the semantics of the statements ("transferfunctions")and thosebased on the flow ofcontrol.

TransferFunctions

The data-flow values before and after a statement are constrained by the semantics of the statement. Forexample, suppose our data-flow analysis involves determining the constant value of variables at points.If variable *a* has value *v* before executing statement **b** = a, then both *a* and *b* will have the value *v* afterthestatement.Thisrelationshipbetweenthedata-flowvaluesbeforeandaftertheassignmentstatementis knownas atransfer function.

Transfer functions come in two flavors: information may propagate forward along execution paths, or itmay flow backwards up the execution paths.In a forward-flow problem, the transfer function of astatement s, which we shall usually denote f(a), takes the data-flow value before the statement andproducesanew data-flow value after thestatement.That is,

$$\text{OUT}[s] = f_s(\text{IN}[s]).$$

Conversely, in a backward-flow problem, the transfer function f(a) for statement 8 converts a data-flowvalueafter thestatementto a new data-flowvaluebeforethestatement. That is,

$$\text{IN}[s] = f_s(\text{OUT}[s]).$$

Control– FlowConstraints

The second set of constraints on data-flow values is derived from the flow of control. Within a basicblock, control flow is simple. If a block *B* consists of statements s1, s_2, • • • ,s_n in that order, then thecontrol-flowvalue out of***Si*** is thesame as the control-flowvalue into ***Si+i.***That is,

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1, 2, \ldots, n-1.$$

However, control-flow edges between basic blocks create more complex constraints between the laststatement of one basic block and the first statement of the following block. For example, if we areinterested in collecting all the definitions that may reach a program point, then the set of definitionsreaching the leader statement of a basic block is the union of the definitions after the last statements ofeachof thepredecessorblocks. Thenext sectiongives thedetails ofhowdata flows amongtheblocks.

3. Data-FlowSchemasonBasicBlocks

While a data-flow schema involves data-flow values at each point in the program, we can save timeand space by recognizing that what goes on inside a block is usually quite simple. Control flows fromthebeginningtotheendoftheblock,withoutinterruptionorbranching.Thus,wecanrestatetheschema in terms of data-flow values entering and leaving the blocks. We denote the data-flow valuesimmediately before andimmediately after each basic block *B* by *m[B]* and 0 U T [S] , respectively.The constraints involving *m[B]* and 0UT[B] can be derived from those involving ***w[s]*** and OUT[s] forthevarious statements ***s*** in *B*as follows.

Suppose block B consists of statements s 1 , . . ., sn, in that order.If si is the first statement of basicblock B, then m[B] =I N [S I] , Similarly, if snis the last statement of basic block B, then OUT[S] =OUT[s,,] . The transfer function of a basic block B, which we denote fB, can be derived by composingthetransferfunctionsofthestatementsinthe
block.Thatis,letfa.bethetransferfunctionofstatementst.Thenof statement si. Then fB = f,sn, o . . . o f,s2, o fsl. . The relationship between thebeginningand end oftheblock is

$$\text{OUT}[B] = f_B(\text{IN}[B]).$$

Theconstraintsduetocontrolflowbetweenbasicblockscaneasily be rewritten bysubstituting *IN[B]* and *OUT[B]* for **IN[SI**] and **OUT[sn],** respectively. For instance, if data-flow valuesare information about the sets of constants that *may* be assigned to a variable, then we have a forward-flowproblemin which

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

When the data-flow is backwards as we shall soon see in live-variable analy-sis, the equations aresimilar,but with the roles oftheIN's and OUT's reversed. Thatis,

$$\text{IN}[B] = f_B(\text{OUT}[B])$$
$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S].$$

Unlike linear arithmetic equations, the data-flow equations usually do not have a unique solution. Ourgoal is to find the most "precise" solution that satisfies the two sets of constraints: control-flow andtransfer constraints. That is, we need a solution that encourages valid code improvements, but does notjustifyunsafetransformations— those that changewhat theprogramcomputes.

4. ReachingDefinitions

"Reaching definitions" is one of the most common and useful data-flow schemas. By knowing where ina program each variable *x* may have been defined when control reaches each point p, we can determinemanythingsabout *x.* Forjusttwoexamples,acompilerthenknowswhether *x* isaconstantatpoint *p,* and a debugger can tell whether it is possible for *x* to be an undefined variable, should *x* be usedat *p.*

Wesayadefinition *dreaches* apoint *p* ifthereisa path from the point immediatelyfollowing *d* to *p,* such that *d* is not "killed" along that path. We *kill* a definition of a variable *x* if there isany other definition of *x* anywhere along the path .if a definition *d* of some variable *x* reaches point p,then*d*might be theplaceat which thevalue of*x*used at*p* was last defined.

Adefinitionofavariable *x* isastatementthatassigns,ormayassign,avalueto *x.* Procedureparameters, array accesses, and indirect references all may have aliases, and it is not easy to tell if astatementisreferringtoaparticularvariable*x.*Programanalysismustbeconservative;ifwedonotnote that the path may have loops, so we could come to another occurrence of *d* along the path, whichdoes not"kill"*d.*

know whether a statement *s* is assigning a value to *x,* we must assume that it *may* assign to it; that is,variable *x* after statement *s* may have either its original value before *s* or the new value created by *s.* Forthe sake of simplicity, the rest of the chapter assumes that we are dealing only with variables that haveno aliases. This class of variables includes all local scalar variables in most languages; in the case of Cand*C++,* local variableswhoseaddresses havebeen computed at somepoint areexcluded.

TransferEquationsforReachingDefinitions

Startby examining thedetails of asinglestatement. Consideradefinition

```
d: u = v+w
```

Here,andfrequentlyinwhatfollows,+isusedasagenericbinaryoperator.Thisstatement"generates"adefinition *d* of variable*u* and "kills" all the

otherdefinitionsintheprogramthatdefinevariable *u,* whileleavingthere-maningincomingdefinitionsunaffected. Thetransfer functionofdefinition*d* thuscan beexpressed as

$$f_d(x) = gen_d \cup (x - kill_d) \qquad (9.1)$$

where gend = {d}, the set of definitions generated by the statement, and killd is the set of all otherdefinitionsof u in the program.

Thetransferfunctionofabasicblockcanbefoundbycomposingthetransferfunctionsofthestatements contained therein. The composition of functions of the form (9.1), which we shall refer to as"gen-kill form," is also of that form, as we can see as follows. Suppose there are two functions fi(x) =gen1U (x-kill1) andf2(x) =gen2 U(x —kill2). Then

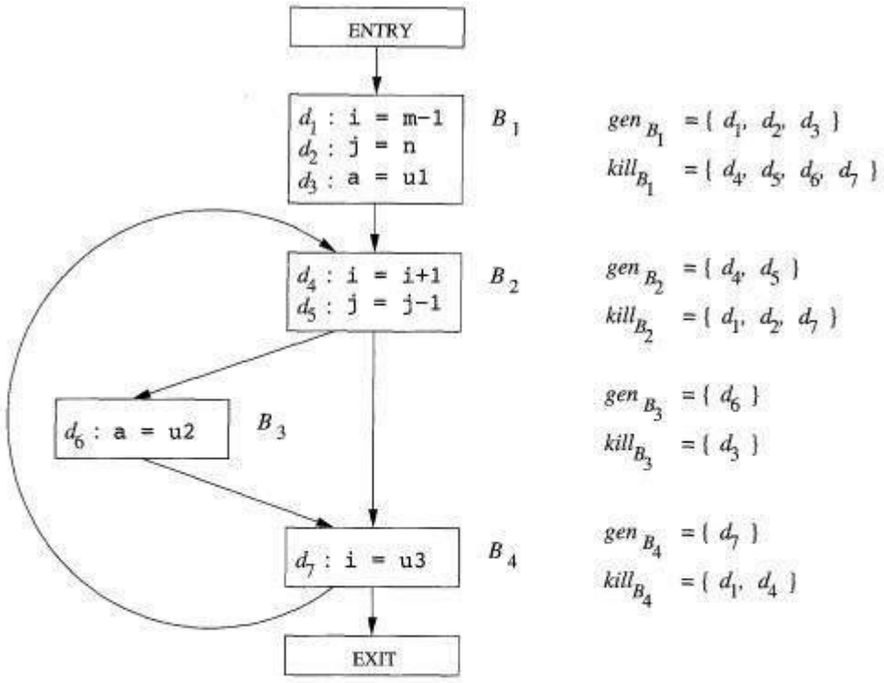

Figure 9.13: Flow graph for illustrating reaching definitions

$$
\begin{aligned}
f_2(f_1(x)) &= gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2) \\
&= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2))
\end{aligned}
$$

This rule extends to a block consisting of any number of statements. Suppose block *B* has *n* statements,withtransfer functions *fi(x) =geni* U *(x —kilh)* for i= 1,2, ... , n.Then the transfer functionforblock*B*may bewrittenas:

$$f_B(x) = gen_B \cup (x - kill_B),$$

where

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

and

$$
\begin{aligned}
gen_B &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\
&\quad \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n)
\end{aligned}
$$

Thus, like a statement, a basic block also generates a set of definitions and kills a set of definitions. Thegen set contains all the definitions inside the block that are "visible" immediately after the block — werefertothemasdownwardsexposed.Adefinitionisdownwardsexposedinabasicblockonlyifitis

not "killed" by a subsequent definition to the same variable inside the same basic block. A basic block'skill set is simply the union of all the definitions killed by the individual statements. Notice that adefinition may appear in both the gen and kill set of a basic block. If so, the fact that it is in gen takesprecedence,becausein gen-kill form, thekillset is applied beforethe gen set.

Example9 . 1 0: Thegen setforthe basic block

d_1: a = 3
d_2: a = 4

is{d2}sinced1isnotdownwardsexposed.Thekillsetcontainsbothd1andd2,sinced1killsd2and vice versa.Nonetheless, since the subtraction of the kill set precedes the union operation with thegenset, theresult of thetransfer functionfor this block always includes definition d2.

Control-FlowEquations

Next, we consider the set of constraints derived from the control flow between basic blocks. Since adefinition reaches a program point as long as there exists at least one path along which the definitionreaches, **O U T** [P] **C** *m[B]* whenever there is a control-flow edge from *P* to *B.* However, since adefinition cannot reach a point unless there is a path along which it reaches, *w[B]* needs to be no largerthantheunion ofthereaching definitions ofall the predecessor blocks.That is, it issafeto assume

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

We refer to union as the *meet operator* for reaching definitions. In any data-flow schema, the meetoperatoristheoneweusetocreateasummaryofthecontributionsfromdifferentpathsattheconfluenceofthosepaths.

IterativeAlgorithmforReachingDefinitions

Weassumethateverycontrol-flowgraphhastwoemptybasicblocks,anENTRYnode,whichrepresentsthestartingpointofthegraph,andanEXITnodetowhichallexitsoutofthegraphgo.Since no definitions reach the beginning of the graph, the transfer function for the ENTRYblock is asimpleconstant function that returns0 as ananswer.Thatis, OUT [ENTRY]=0.

Thereachingdefinitionsproblemis definedbythefollowingequations:

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

and for all basic blocks B other than ENTRY,

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

These equations can be solved using the following algorithm. The result of the algorithm is the *leastfixedpoint* of the equations, i.e., the solution whose assigned values to the **IN** ' s and OUT's is containedin the corresponding values for any other solution to the equations. The result of the algorithm below isacceptable, since any definition in one of the sets **IN** or OUT surely must reach the point described. It isadesirable solution, sinceitdoes not includeanydefinitionsthat wecan besuredo not reach.

Al g ori th m 9 .1 1 :Reachingdefinitions.

INPUT: Aflow graphfor which*kills* and gen_Bhavebeencomputedfor each block *B.*

OUTPUT:IN[B]and OUT **[B],** thesetofdefinitionsreachingtheentryandexitofeachblock*B*of theflowgraph.

METHOD: Weuseaniterativeapproach,inwhichwestartwiththe"estimate" OUT[JB]=0 forall *B* and converge to the desired values of **IN** and OUT. As we must iterate until the **IN** ' s (and hencethe OUT's) converge, we could use a boolean variable *change* to record, on each pass through theblocks, whether any OUT has changed. However, in this and in similar algorithms described later, weassumethat theexact mechanismforkeeping trackof changesis understood,and weelidethose details.

The algorithm is sketched in Fig. 9.14. The first two lines initialize certain data-flow values.[4] Line (3)starts the loop in which we iterate until convergence, and the inner loop of lines (4) through (6) appliesthedata-flowequations to every block otherthanthe entry. •

Algorithm 9.11 propagates definitions as far as they will go with-out being killed, thus simulating allpossible executions of the program. Algo-rithm 9.11 will eventually halt, because for every *B*, OUT[B]never shrinks; once a definition is added, it stays there forever. (See Exercise 9.2.6.) Since the set of alldefinitions is finite, eventually there must be a pass of the while-loop during which nothing is added
toanyOUT,andthealgorithmthenterminates.Wearesafeterminatingthenbecauseiftheout'shavenotchange d, the**IN**' s will

```
1)  OUT[ENTRY] = ∅;
2)  for (each basic block B other than ENTRY) OUT[B] = ∅;
3)  while (changes to any OUT occur)
4)          for (each basic block B other than ENTRY) {
5)                  IN[B] = ⋃_{P a predecessor of B} OUT[P];
6)                  OUT[B] = gen_B ∪ (IN[B] − kill_B);
            }
```

Figure 9.14: Iterative algorithm to compute reaching definitions

not change on the next pass. And, if the IN'S do not change, the OUT's cannot, so on all subsequentpassesthere can beno changes.

The number of nodes in the flow graph is an upper bound on the number of times around the while-loop. The reason is that if a definition reaches a point, it can do so along a cycle-free path, and thenumberofnodesinaflowgraphisanupperboundonthenumberofnodesinacycle-freepath.Each

time around the while-loop, each definition progresses by at least one node along the path in question,andit often progressesby morethan onenode, depending on theorder inwhichthe nodes arevisited.

In fact, if we properly order the blocks in the for-loop of line (5), there is empirical evidence that theaverage number of iterations of the while-loop is under 5 (see Section 9.6.7). Since sets of definitionscan be represented by bit vectors, and the operations on these sets can be implemented by logicaloperationson thebit vectors,Algorithm 9.11 is surprisingly efficientin practice.

Example9.12:Weshallrepresentthesevendefinitionsd1,d2,•••,d>jintheflowgraphofFig.
9.13 by bit vectors, where bit i from the left represents definition d{. The union of sets is computed bytaking the logical OR of the corresponding bit vectors. The difference of two sets S — T is computed bycomplementing the bit vector of T, and then taking the logical AND of that complement, with the bitvectorfor S.

Shown in the table of Fig. 9.15 are the values taken on by the IN and OUT sets in Algorithm 9.11. Theinitialvalues,indicatedbyasuperscript0,asinOUTfS]0,areassigned,bytheloopofline(2)ofFig.
9.14. They are each the empty set, represented by bit vector 000 0000. The values of subsequent passesof the algorithm are also indicated by superscripts, and labeled IN [I?]1 and OUTfS]1 for the first passandm[Bf and OUT[S]2 for thesecond.

Supposethefor-loop of lines(4)through(6)is executedwith Btakingon thevalues

$$B_1, B_2, B_3, B_4, \text{EXIT}$$

in that order. With B = B1, since OUT [ENTRY] = 0, [IN B1]-Pow(1) is the empty set, and OUT[P1]1isgenBl. This valuediffers from the previous value OUT[Si]0 , so

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Figure 9.15: Computation of IN and OUT

wenow knowthereisa changeon thefirst round (and willproceed

toasecondround).Thenweconsider$B = B_2$and compute

$$\begin{aligned}
\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\
&= 111\ 0000 + 000\ 0000 = 111\ 0000 \\
\text{OUT}[B_2]^1 &= gen[B_2] \cup (\text{IN}[B_2]^1 - kill[B_2]) \\
&= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100
\end{aligned}$$

This computation is summarized in Fig. 9.15. For instance, at the end of the first pass, OUT [5 2] 1 =001 1100, reflecting the fact that d4 and d5 are generated in B2, while d3 reaches the beginning of B2andis not killed in B2.

Notice that after the second round, OUT [B2] has changed to reflect the fact that d& also reaches thebeginning of B2 and is not killed by B2. We did not learn that fact on the first pass, because the pathfromd6totheendofB2,whichisB3-»B4->B2,isnottraversedinthatorderbyasinglepass.Thatis,bythetimewelearnthatd$reachestheendofB4,wehavealreadycomputedIN[B2]andOUT[B2] onthe first pass.

There are nochanges in any of the OUTsetsafter the secondpass.Thus, after a third pass, thealgorithmterminates, with theIN's and OUT's asin the final twocolumns of Fig. 9.15.

5. Live-VariableAnalysis

Some code-improving transformations depend on information computed in the direction opposite to theflow of control ina program; we shall examine one such example now. In *live-variable analysis* wewish to know for variable *x* and point *p* whether the value of *x* at *p* could be used along some path in theflowgraph starting at *p.*Ifso, wesay*x*is*live*at *p;*otherwise, *x*is *dead*at *p.*

An important use for live-variable information is register allocation for basic blocks. Aspects of thisissue were introduced in Sections 8.6 and 8.8. After a value is computed in a register, and presumablyused within a block, it is not necessary to store that value if it is dead at the end of the block. Also, if allregisters are full and we need another register, we should favor using a register with a dead value, sincethatvalue does not haveto bestored.

Here, wedefinethedata-flowequationsdirectly intermsof IN [5]and OUTpB], which representthesetofvariablesliveatthepointsimmediatelybeforeandafterblockB,respectively.Theseequations can also be derived by first defining the transfer functions of individual statements andcomposingthem to createthe transfer function of abasic block.Define

1.defBas the set of variablesdefined(i.e., definitely assigned values)in B prior to any use of thatvariable in B, and useB as the set of variables whose values may be used in B prior to any definition ofthevariable.

Example 9 . 1 3 : For instance, block B2in Fig. 9.13 definitely uses i. It also uses j before anyredefinition of j, unless it is possible that i and j are aliases of one another.Assuming there are noaliases among the variables in Fig. 9.13, then uses2={i,j}- Also,B2clearly defines iand j.Assumingthereareno aliases, defB2=aswell.

As a consequence of the definitions, any variable in useB must be considered live onentrance to blockB, while definitionsof variables in defBdefinitely are deadatthe beginning of B.Ineffect,membershipin defB"kills" any opportunityforavariableto belivebecauseof pathsthat begin atB.

Thus,the equationsrelating defand useto theunknowns INand OUTaredefinedas follows:

$$\text{IN}[\text{EXIT}] = \emptyset$$

and for all basic blocks B other than EXIT,

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$
$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

The first equation specifies the boundary condition, which is that no variables are live on exit from theprogram. The second equation says that a variable is live coming into a block if either it is used beforeredefinition in the block or it is live coming out of the block and is not redefined in the block. The thirdequation says that a variable is live coming out of a block if and only if it is live coming into one of itssuccessors.

The relationship between the equations for liveness and the reaching-defin-itions equations should benoticed:

Both sets of equations have union as the meet operator. The reason is that in each data-flowschema we propagate information along paths, and we care only about whether *any* path with desiredpropertiesexist, ratherthan whethersomething is truealong*all* paths.

• However, information flow for liveness travels "backward," opposite to the direction of control flow,becauseinthisproblemwewanttomakesurethattheuseofavariable*x* atapoint*p* istransmittedtoallpointsprior to *p* inan execution path,sothatwe may know atthe prior pointthat *x* willhave itsvalueused.

To solve a backward problem, instead of initializing O U T [E N T R Y] , we initialize I N [EXIT] .SetsI NandO U Thave their rolesinterchanged,anduse anddefsubstitute for genandkill,respectively. As for reaching definitions, the solution to the liveness equations is not necessarily unique,and we want the so-lution with the smallest sets of live variables. The algorithm used is essentially abackwardsversion ofAlgorithm 9.11.

Algorithm9.14:Live-variableanalysis.

INPUT:Aflowgraphwithdef anduse computedforeachblock.

OUTPUT: m[B] and O U T [£] ,the set of variables live on entry and exit of each block B of the flowgraph.

```
IN[EXIT] = ∅;
for (each basic block B other than EXIT) IN[B] = ∅;
while (changes to any IN occur)
        for (each basic block B other than EXIT) {
                OUT[B] = ⋃S a successor of B IN[S];
                IN[B] = useB ∪ (OUT[B] − defB);
        }
```

Figure 9.16: Iterative algorithm to compute live variables

6.Available Expressions

An expression x + y is available at a point p if every path from the entry node to p evaluatesx + y, andafterthelastsuchevaluationpriortoreachingp,therearenosubsequentassignmentstoxory.5Forthe available-expressions data-flow schema we say that a block kills expression x + y if it assigns (ormay 5 N o te that, as usual in this chapter, we use the operator + as a generic operator, not necessarilystanding foraddition.

assign) x or y and does not subsequently recompute x + y. A block generates expression x + y if itdefinitelyevaluates x+yand does not subsequently definex ory.

Note that the notion of "killing" or "generating" an available expression is not exactly the same as thatfor reaching definitions. Nevertheless, these notions of "kill" and "generate" behave essentially as theydofor reaching definitions.

The primaryuse ofavailable-expression information is for detecting global common subexpressions.For example, in Fig. 9.17(a), the expression 4 * i in block Bs will be a common subexpression if 4 * i isavailable at the entry point of block B3. It will be available if i is not assigned a new value in block B2,orif, as in Fig. 9.17(b),4 * iis recomputed afteri is assigned in B2.

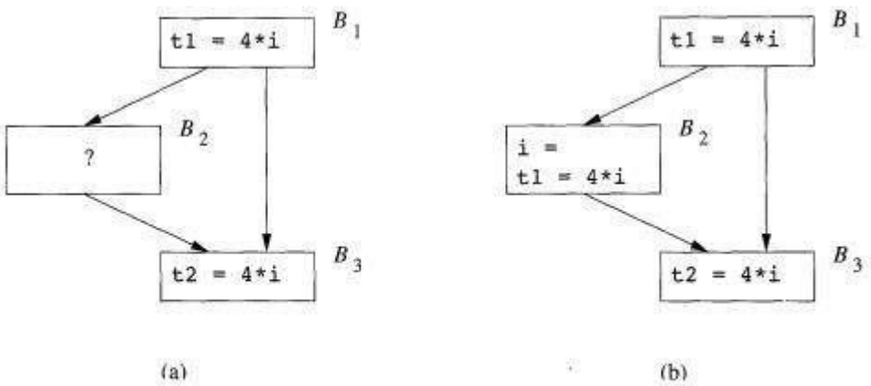

Figure 9.17: Potential common subexpressions across blocks

We can compute the set of generated expressions for each point in a block, working from beginning toendoftheblock.Atthepointpriortotheblock,noexpressionsaregenerated.IfatpointpsetSof

expressions is available, and *q* is the point after p, with statement x = y+**z** between them, then we formtheset of expressions available at*q* bythe following two steps.

Addto *S* theexpression*y*+ *z*.

DeletefromSanyexpressioninvolvingvariable*x*.

Note the steps must bedone in the correct order, as *x* could be the sameas *y* or *z*. After we reachtheend of the block, *S* is the set of generated expressions for the block. The set of killed expressions is allexpressions, say *y* + *z*, such that either *y* or *z* is defined in the block, and *y* + *z* is not generated by theblock.

E x a m p l e 9.15 : Consider the four statements of Fig. 9.18. After the first, ***b* + *c*** is available. After thesecondstatement,a— ***d*** becomesavailable,but ***b+c*** isnolongeravailable,because ***b*** hasbeenredefined.Thethirdstatementdoesnotmake ***b+c*** availableagain,becausethevalueofcisimmediatelychanged.

After the last statement, *a* — *d* is no longer available, because *d* has changed. Thus no expressions aregenerated,and all expressions involving*a, b,* c, or*d* arekilled.

Statement	Available Expressions
	$\emptyset$
`a = b + c`	
	$\{b+c\}$
`b = a - d`	
	$\{a-d\}$
`c = b + c`	
	$\{a-d\}$
`d = a - d`	
	$\emptyset$

Figure 9.18: Computation of available expressions

Wecanfindavailableexpressionsinamannerreminiscentofthewayreach-ingdefinitionsarecomputed. Suppose *U* is the "universal" set of all expressions appearing on the right of one or morestatementsoftheprogram.Foreachblock*B,* let***IN****[B]* bethesetofexpressionsin*U*thatareavailableatthepointjustbeforethebeginningof *B.* Let**OUT***[B]* bethesameforthepointfollowingtheendof*B.* Define***e.genB***tobetheexpressionsgeneratedby*B*and *eJnills*tobethesetofexpressionsin *U* killed in *B.* Note that **I N , O U T ,** e_#en, and *eJkill* can all be represented by bit vectors. Thefollowingequations relatethe unknowns

IN and OUT to each other and the known quantities *e_gen* and *e_kill*:

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

and for all basic blocks B other than ENTRY,

$$\text{OUT}[B] = e\_gen_B \cup (\text{IN}[B] - e\_kill_B)$$
$$\text{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \text{OUT}[P].$$

T he above equations look almost identical to the equations for reaching definitions. Like reachingdefinitions, the boundary condition is OUT [ENTRY] = 0, because at the exit of the E N T R Y node,thereareno available expressions.

The most important difference is that the meet operator is intersection rather than union. This operator isthe proper one because an expression is available at the beginning of a block only if it is available at theend of all its predecessors. In contrast, a definition reaches the beginning of a block whenever it reachesthe end ofany oneormoreof its predecessors.

The use of D rather than U makes the available-expression equations behave differently from those ofreaching definitions. While neither set has a unique solution, for reaching definitions, it is the solutionwith the smallest sets that corresponds to the definition of "reaching," and we obtained that solution bystartingwiththeassumptionthatnothingreachedanywhere,andbuildinguptothesolution.Inthatway,wenev erassumedthatadefinition *d* couldreachapoint *p* unlessanactual pathpropagating *d* to *p* could be found. In contrast, for available expression equations we want the solutionwith the largest sets of available expressions, so we start with an approximation that is too large andworkdown.

It may not be obvious that by starting with the assumption "everything (i.e., the set *U)* is availableeverywhere except at the end of the entry block" and eliminating only those expressions for which wecan discover a path along which it is not available, we do reach a set of truly available expressions. Inthe case of available expressions, it is conservative to produce a subset of the exact set of availableexpressions.Theargumentforsubsetsbeingconservativeisthatourintendeduseoftheinformationisto replace the computation of an available expression by a previously computed value. Not knowing anexpres-sion is available only inhibits us from improving the code, while believing an expression isavailablewhen it is not could causeus to change what theprogram computes.

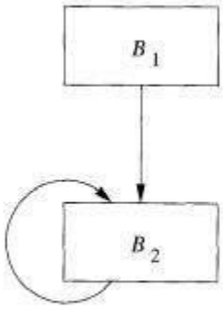

Figure 9.19: Initializing the OUT sets to $\emptyset$ is too restrictive.

Example9.16:Weshallconcentrateonasingleblock,B2inFig.9.19,toillustratetheeffectofthe initial approximation of OUT[B2]on IN[B 2] -Let G and K abbreviate e.genB2 and e-killB2,respectively.Thedata-flow equations forblock B2are

$$\text{IN}[B_2] = \text{OUT}[B_1] \cap \text{OUT}[B_2]$$

$$\text{OUT}[B_2] = G \cup (\text{IN}[B_2] - K)$$

These equations may be rewritten as recurrences, with I^j and O^j being the jth approximations of IN$[B_2]$ and OUT$[B_2]$, respectively:

$$I^{j+1} = \text{OUT}[B_1] \cap O^j$$

$$O^{j+1} = G \cup (I^{j+1} - K)$$

Starting with $O^0 = \emptyset$, we get $I^1 = \text{OUT}[B_1] \cap O^0 = \emptyset$. However, if we start with $O^0 = U$, then we get $I^1 = \text{OUT}[B_1] \cap O^0 = \text{OUT}[B_1]$, as we should. Intuitively, the solution obtained starting with $O^0 = U$ is more desirable, because it correctly reflects the fact that expressions in OUT$[B_1]$ that are not killed by B_2 are available at the end of B_2. □

Algorithm9.1 7: Availableexpressions.

INPUT:Aflowgraphwith e-killsand e.genscomputedforeachblock B.Theinitialblock isB1.

OUTPUT: IN [5] and O U T [5] , the set of expressions available at the entry and exit of each block Bof theflowgraph.

```
OUT[ENTRY] = ∅;
for (each basic block B other than ENTRY) OUT[B] = U;
while (changes to any OUT occur)
        for (each basic block B other than ENTRY) {
                IN[B] = ⋂_{P a predecessor of B} OUT[P];
                OUT[B] = e_gen_B ∪ (IN[B] − e_kill_B);
        }
```

Figure 9.20: Iterative algorithm to compute available expressions

Figure9.20: Iterativealgorithmtocomputeavailableexpressions