COMPILER DESIGN

UNIT - V

PRINCIPAL SOURCES OF OPTIMISATION:

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations:

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation,
- Dead-code elimination
- Constant folding

The other transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Common Sub expressions elimination:

• An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

• For example

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
```

```
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

• For example:

```
x=Pi;
……
A=x*r*r;
```

The optimization using copy propagation can be done as follows: A=Pi*r*r; Here the variable x is eliminated.

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example:

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

Here, 'if' statement is dead code because this condition will never get satisfied. Constant folding: Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 thereby eliminating a division operation.

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- Code motion, which moves code outside aloop;
- Induction-variable elimination, which we apply to replace variables from innerloop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, suchas a multiplication by an addition.

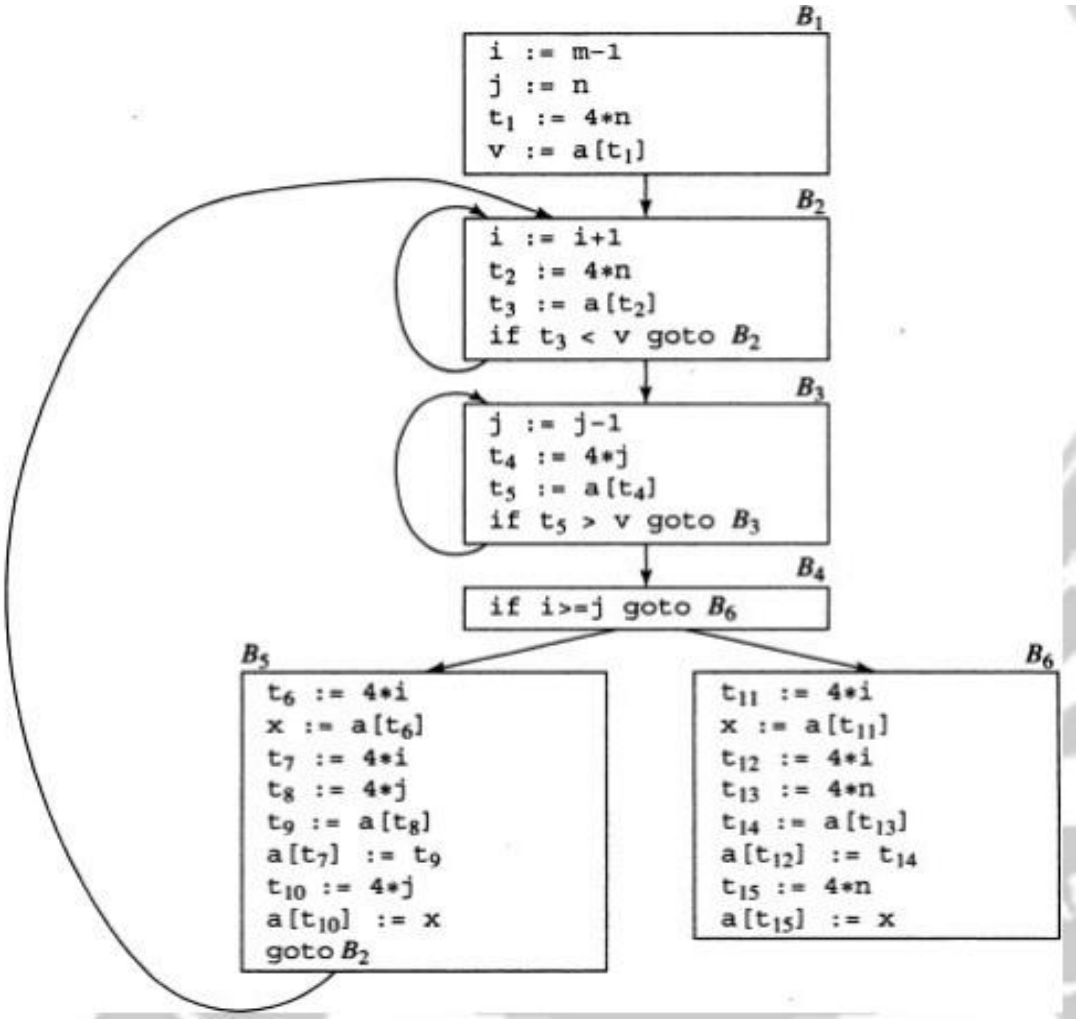


Fig. Flow graph

Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
Code motion will result in the equivalent of
t= limit-2;
while(i<=t) /* statement does not change limit or t*/
```

Induction Variables:

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

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of inductionvariable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 inFig.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. 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 be cheaper.

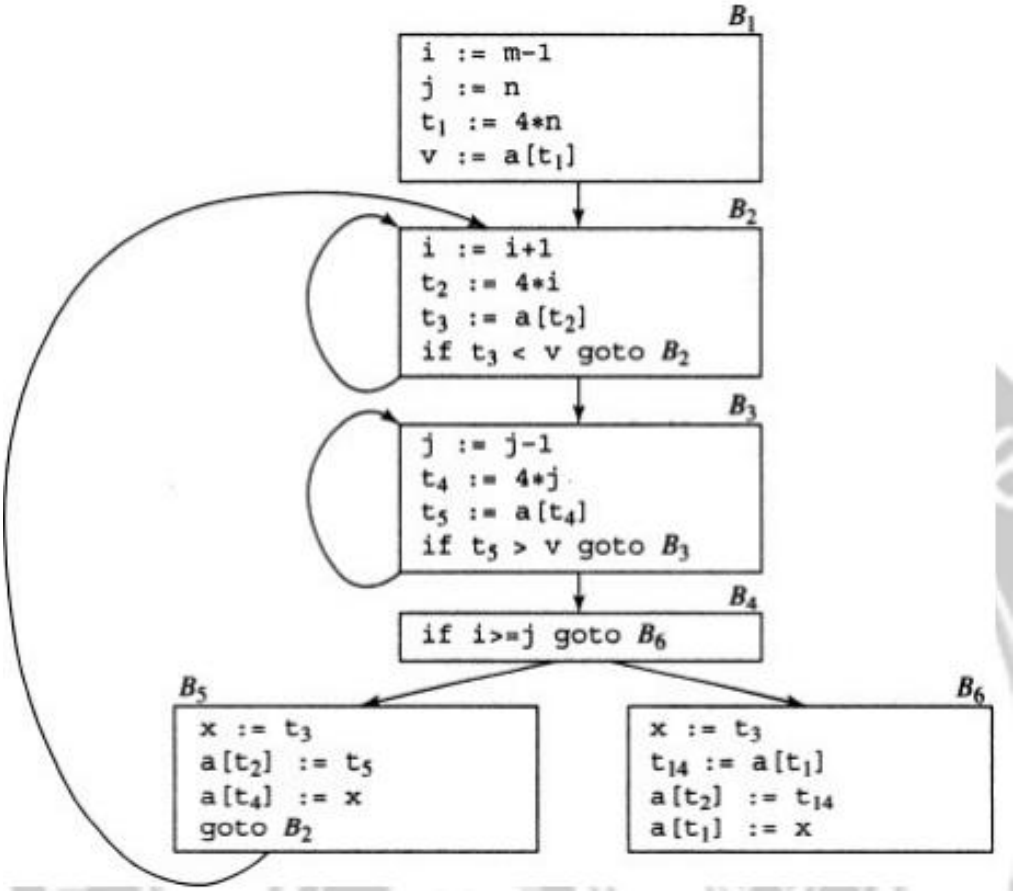


Fig. B5 and B6 after common sub expression elimination

Data flow analysis:

Data flow analysis refers to analyzing the data flow through a program. Data flow analysis helps in optimizing the code. It also helps in increasing efficiency and identifying bugs in the code. In data flow analysis, we track down the variables and how they change over time.

Data flow analysis is a global code optimization technique. The compiler performs code optimization efficiently by collecting all the information about a program and distributing it to each block of its **control flow graph (CFG)**. This process is known as data flow analysis.

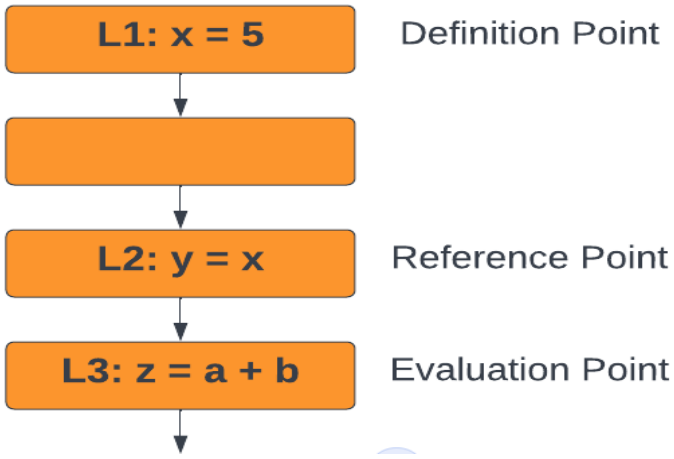
The code optimization associated with data flow analysis uses dead code elimination. It is tacking the variables. It identifies the constant expressions. These measures reduce unnecessary computations.

Basic Technologies

Below are some basic terminologies related to data flow analysis.

- **Definition Point-** A definition point is a point in a program that defines a data item.
- **Reference Point-** A reference point is a point in a program that contains a reference to a data item.
- **Evaluation Point-** An evaluation point is a point in a program that contains an expression to be evaluated.

The below diagram shows an example of a definition point, a reference point, and an evaluation point in a program.



Data Flow Analysis Equation

The data flow analysis equation is used to collect information about a program block. The following is the data flow analysis equation for a statement s-

Out[s] = gen[s] U In[s] - Kill[s]

where

Out[s] is the information at the end of the statement s.

gen[s] is the information generated by the statement s.

In[s] is the information at the beginning of the statement s.

Kill[s] is the information killed or removed by the statement s.

Data Flow Properties

Some properties of the data flow analysis are-

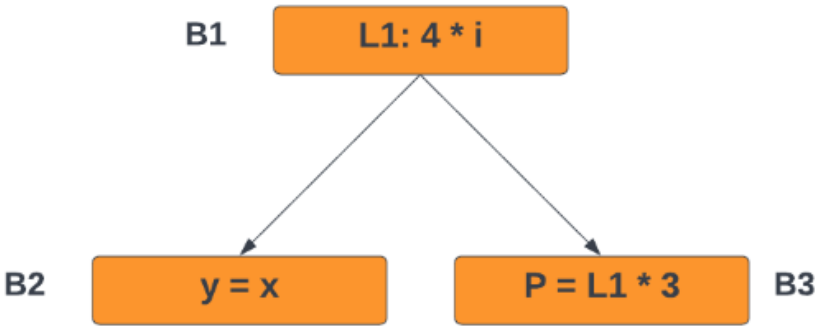
- Available expression
- Reaching definition
- Line variable
- Busy expression

Available Expression

An expression **a + b** is said to be available at a program point **x** if none of its operands gets modified before their use. It is used to eliminate common subexpressions.

An expression is available at its evaluation point.

Example:

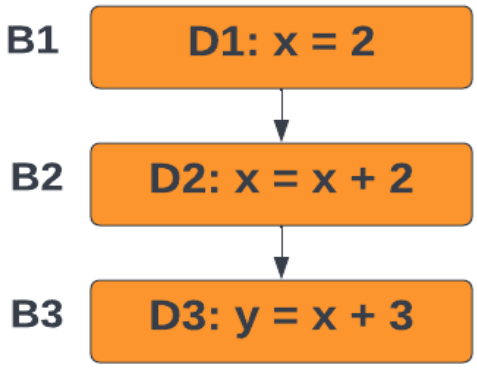


In the above example, the expression L1: 4 * i is an available expression since this expression is available for blocks B2 and B3, and no operand is getting modified.

Reaching Definition

A definition **D** is reaching a point **x** if D is not killed or redefined before that point. It is generally used in variable/constant propagation.

Example:



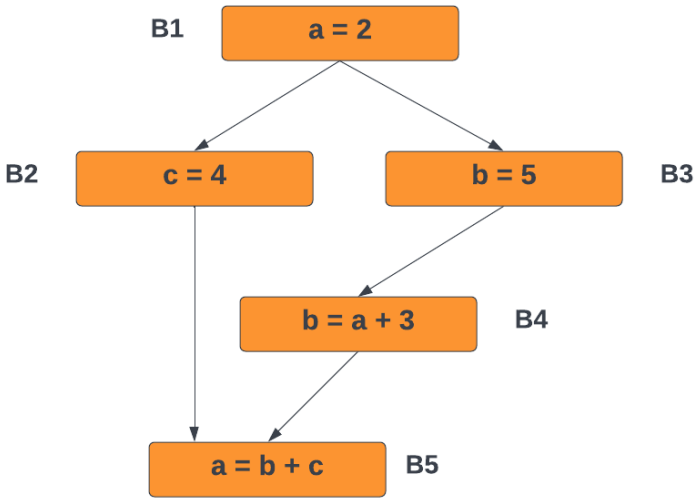
In the above example, D1 is a reaching definition for block B2 since the value of x is not changed (it is two only) but D1 is not a reaching definition for block B3 because the value of x is changed to x + 2. This means D1 is killed or redefined by D2.

Live Variable

A variable **x** is said to be live at a point **p** if the variable's value is not killed or redefined by some block. If the variable is killed or redefined, it is said to be dead.

It is generally used in register allocation and dead code elimination.

Example:



In the above example, the variable a is live at blocks B1,B2, B3 and B4 but is killed at block B5 since its value is changed from 2 to b + c. Similarly, variable b is live at block B3 but is killed at block B4.

Busy Expression

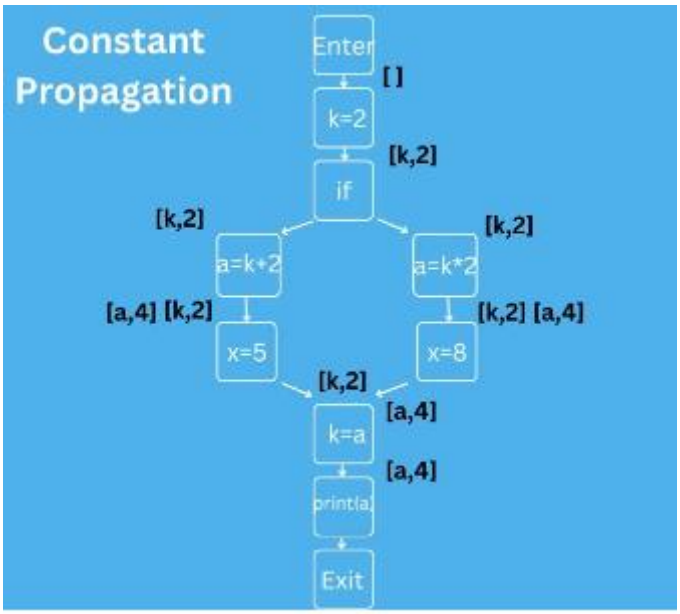
An expression is said to be busy along a path if its evaluation occurs along that path, but none of its operand definitions appears before it.

It is used for performing code movement optimization.

Constant Propagation Analysis:

It determines the point where the program variable guarantees a constant value. The value computed is represented as a pair of (variable, value). If we have (x,v) as a pair, then x is guaranteed to have a value v whenever that block is reached during program execution.

Below is the CFG that can be used to understand the constant propagation information.



Here, we have the variable and their associated values listed down adjoining the block. We can see that for block 2 where k=2 the variable k is propagated with value 2 until associated with a new value.

Partial-Redundancy Elimination:

PRE (Partial Redundancy Elimination) is a compiler optimization method. PRE aims to remove redundant computations from the program. Redundant computations are those that are achieved in more than one instance. However, it produces the same result every time. By removing these redundancies, PRE can enhance the overall performance of the program by reducing the number of instructions carried out.

Redundancy is basically unnecessary or repetitive code present in a program that does not contribute to the functionality of the program. When a computation is performed more than once along some of the execution paths

but not along all execution paths, this condition is known as "partial redundancy."

An example of full redundancy is given below.

```
if(condition1) {
x = a + b;
}
else{
x = a + b;
}
```

As you can see in the above example, the computation of A+B is fully redundant, as it is performed along both execution paths.
But the condition of partial redundancy arrived if we modified the code as follows.

```
if(condition1) {
x = a + b;
}
if(condition2) {
y = a + b;
}
```

As you can see in the above example, A+B is only partially redundant because it is performed along some execution path(“given that both conditions are true”) but not along all execution paths.

The Source of Redundancy

Redundancy is due to the presence of one or multiple expressions from the following expressions.

1. Partially redundant expressions: It is an expression in which the value that is computed by the expression is already available on some of the paths but is not available on all the paths through a program to that expression.
2. Loop invariant expression: It is a condition that is necessarily true immediately before and immediately after each iteration of the loop.
3. Common sub-expression: A common sub-expression is an expression that has appeared and computed before and appears again during the computation of the code.

Loops in Flow Graph:

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

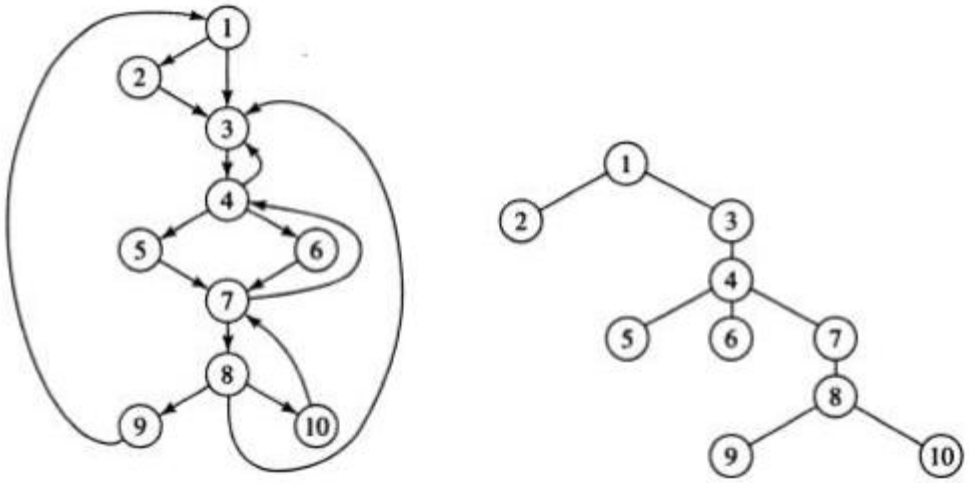


Fig. Flow graph (b) Dominator tree

The way of presenting dominator information is in a tree, called the dominator tree, in which

- The initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.

The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.

D(1)={1}
D(2)={1,2}
D(3)={1,3}
D(4)={1,3,4}

D(5)={1,3,4,5}
D(6)={1,3,4,6}
D(7)={1,3,4,7}
D(8)={1,3,4,7,8}
D(9)={1,3,4,7,8,9}
D(10)={1,3,4,7,8,10}

Natural Loops:

One application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of loops:

- Ø A loop must have a single entrypoint, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
- Ø There must be at least one way to iterate the loop(i.e.)at least one path back to the headerOne way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of

edges are called as back edges.

Example:

In the above graph,

7→4 4 DOM 7

10 →7 7 DOM 10
4→3
8→3
9 →1

The above edges will form loop in flow graph. Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Inner loops:

If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the
other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.

When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheader. The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header. Edges from inside loop L to the header are not changed. Initially the pre-header is empty, but transformations on L may place statements in it.

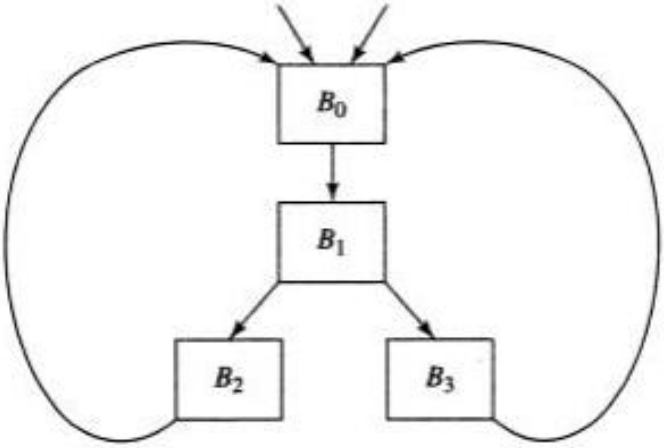


Fig. 5.4 Two loops with the same header

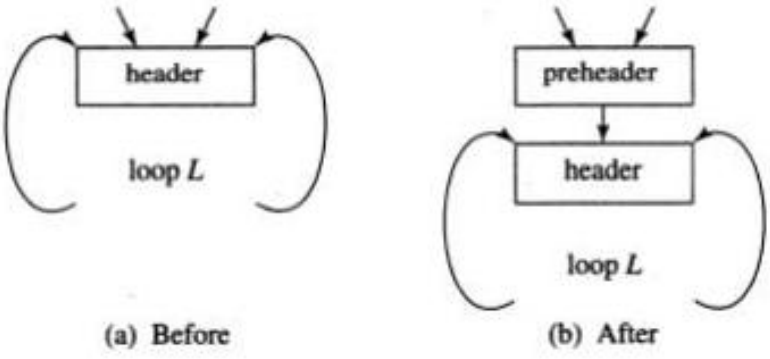


Fig. Introduction of the preheader