Module 29 – Basic Blocks, Flow graphs and Next-use information

After understanding the need for grouping the three-address intermediate code into basic blocks, this module will detail on the construction of basic blocks. A sequence of basic blocks represents a control flow of the program and is called as flow graphs and this module will discuss the construction of a flow graph. This module will conclude with the possible transformations in basic blocks to optimize the three-address code.

29.1 Basic Blocks and Flow graphs

A basic block is a sequence of consecutive instructions with exactly one entry point and one exit point. The exit points may be more than one if we consider branch instructions. A control flow graph (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ if and only if B_j can be executed immediately after B_i. The algorithm for basic block construction was detailed in the previous module and is given below for a quick reference.

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements of basic blocks
    a) The first statement is the leader
    b) Any statement that is the target of a goto is a leader
    c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Using the algorithm for basic block construction let us try an example.

Consider the following high level program in Pascal language. Let us try constructing the basic blocks and flow graph of the same.

```
Begin
    prod := 0
    i := 1;
    do begin
            prod : = prod + a[i] * b[i];
            i = i +1;
    end
    while i< = 20
end
```

The following is the three-address code generated for the above sequence of instructions using the semantic rules discussed in the previous modules.

1. prod := 0
2. i := 1
3. t1 := 4 * i
4. t2 := a[t1]
5. t3 := 4 * i
6. t4 := b [t3]
7. t5 := t2 *t4
8. t6 := prod + t5
9. prod := t6
10. t7 := i+1
11. i := t7
12. If i<= 20 goto (3)

The code is a do-while loop. The first two statements in the high-level program initializes two variables and the same is retained in the three-address code as well. The first statement in the body of the do-while loop, multiplies two array values and adds it with another variable. This is done in lines 3 to 9 of the three-address code based on the semantic rules for array access. The second statement of the do-while block increments the index variable 'i' and this is done in lines 10 to 11. Finally line 12 is the three-address code corresponding to the high-level while statement, which checks for end of iteration and branch accordingly.

Now, to construct the basic block and flow graph the following are the observations:

- Line (1) is the beginning of the three-address code and hence is declared as a leader
- Line (3) is the target of a jump from line (12) and hence is also declared a leader
- Statement following (12), say line number (13) is a leader
- Hence, Lines (1) and (2) will form a basic block
- Lines (3) to (12) will form another basic block.

This is depicted in figure 29.1

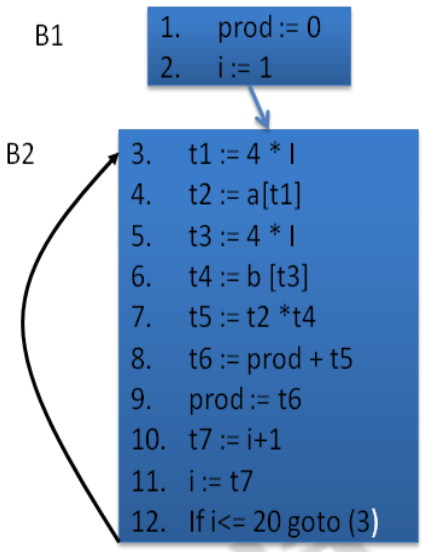


Figure 29.1 Example basic block

The first one is called basic block B1 and the second one as B2. B1 is the predecessor of B2 and so B2 is the successor of B1. The instructions that are part of B1 need to be executed before B2's instructions and hence control flows from B1 to B2.

29.2 Loops

Basic blocks that include a direction of control flow are called as a flow graph. This flow graph can sometimes form a loop that goes in a circular fashion amongst the basic blocks. A loop is a collection of basic blocks, such that

- All blocks in the collection are strongly connected. A strongly connected loop is one in which from any node to any other node, there is a path of length one or more within the loop.
- The collection of blocks has a unique entry, and the only way to reach a block in the loop is through this entry

Loops not containing any other loop is called as Inner loop. Loop that has one or more inner loops is referred to as outer loop.

Consider the basic block and flow graph of figure 29.2. There are 4 blocks B1, B2, B3 and B4.

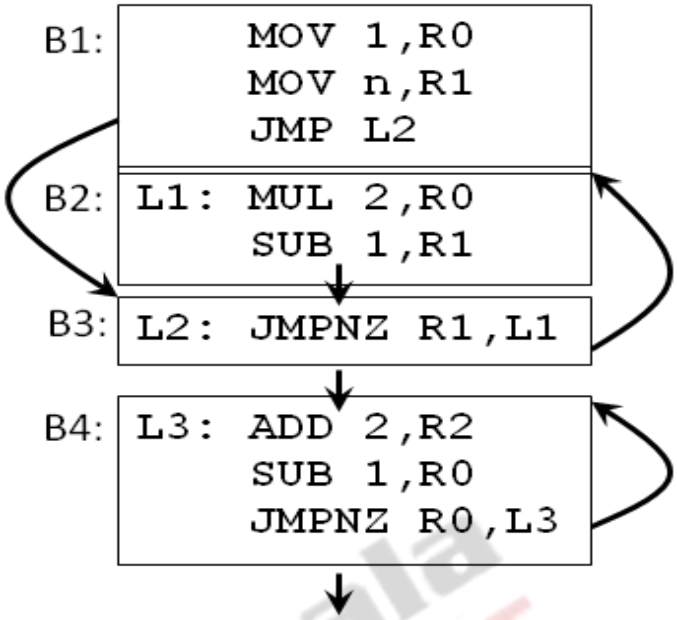
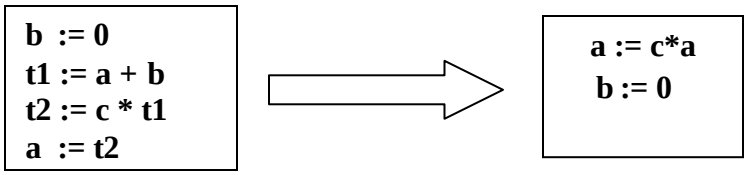


Figure 29.2 Example for understanding types of loops

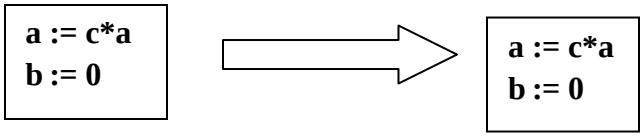
In figure 29.2, B2 is connected to all other blocks with a path length of more than 1and hence is a strongly connected component. So, is the case with B3 and B4.

29.3 Transformations on Basic Blocks

In the previous module, we questioned on the need for basic blocks. Typically, any basic block, computes a set of expression. Values of the variables outside the block is decided by the computation inside the block. Two basic blocks are equivalent if they compute the same set of expressions. Consider the following basic block, which computes the addition of ‘a’ and ‘b’ and multiply with ‘c’ and assigns it to ‘a’. Though this block adds ‘a’ and ‘b’, since ‘b’ is initialized to ‘0’ to start with, the addition of ‘a’ and ‘b’ does not have any effect. Hence, this block can be thought of as simply multiplying ‘a’ and ‘c’ and assigning it to ‘a’ and the value of ‘b’ which is set to ‘0’ and the value of ‘c’ is unaltered.



Alternately, consider the following basic block which multiplies ‘a’ and ‘c’ and assigns it to ‘a’ and initializes ‘b’ to ‘0’. Hence, this basic block can also be thought of as the same as the previous one.



As can be observed from this discussion, basic blocks could be analyzed on a simpler note to verify the validity of some instructions. A code-improving transformation is a code optimization to improve speed or reduce code size. Global transformations are performed across basic blocks. Local transformations are only performed on single basic blocks. The transformations that are carried out must be safe and preserve the meaning of the code. We can say, that a local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form.

The transformations that are possible with basic block can be grouped into the following:

- Structure-Preserving Transformation – In this transformation, the syntactic structure of the statements in the basic blocks are not altered.
- Algebraic Transformation–In this type of transformation, mathematical identity are applied on the statements to perform the transformation and thus altering the syntactic structure

Let us look into these two types of transformations in detail in the next sections

29.3.1 Structure-Preserving Transformation

- Common Sub-expression Elimination – Within a block, if a particular expression is repeated the expression is said to be common. This could be eliminated to avoid unnecessary computation.
- Dead-code elimination – In a basic block, if a particular variable or assignment is not going to be used, then the instruction is said to be dead.
- Renaming of temporary variables – As we know, to convert instructions to three-address code, temporary variables are defined. These variables are renamed to effectively optimize the final code.
- Interchange of two independent adjacent statements – As we have already discussed in the previous module, reordering of instructions will result in an optimized code and this transformation allows swapping of two independent adjacent statements.

29.3.1.1 Common Sub-expression Elimination.

If an expression is computed repeatedly and in between the repeated computations, if the values of the variables of the expression is unaltered then the expression is said to be common.

Consider the following sequence of instructions:

1. a := b+c
2. b := a–d
3. c := b+c
4. d := a-d

In the above set of instruction, the RHS of the expression in line number (1) and (3) has "b+c" and so is "a-d" in line number (2) and (4). But, between line numbers (1) and (3) the value of the variable 'b' changes and hence "b+c" is not a common expression. However, the value of 'a' or'd' doesn't change between line numbers (2) and (4) and hence "a-d" is considered as a common sub-expression. The value of 'd' changes after line number (4) and hence the above instructions could be replaced as follows:

1. a := b+c
2. b := a – d
3. c := b + c
4. d := b

Thus common sub-expression elimination involves, identifying the common-expression and replacing the RHS with the computed LHS variable at the previous line number.

Consider one more example:

1. t1 := b * c
2. t2 := a - t1
3. t3 := b * c
4. t4 := t2 + t3

Here, the common expression is "b*c" and hence this could be replaced as follows:

1. **t1 := b * c**
2. **t2 := a - t1**
3. **t3 := t1**
4. **t4 := t2 + t3**

Here, the instruction at line number (3) is not used and hence, we could also eliminate that and replace t3 with t1. This is referred to as copy propagation. Thus removing common sub-

expressions results in copy code propagation, where the same value is available at multiple variables. This can be removed by replacing the LHS variable with the RHS till the LHS / RHS values change. In this example, we could replace t3 with t1, as both will have the same value and thereby the instruction at line number (3) could be eliminated. Hence the code could be reduced to the following:

1. **t1 := b * c**
2. **t2 := a - t1**
3. **t4 := t2 + t1**

29.3.1.2 Dead code elimination

Any code that is not used is said to be dead. In the previous example, t3 is dead and hence we eliminated the instruction t3:= t1. In a similar fashion, if a particular variable is not going to be used later, then the instruction involving that variable could be eliminated. Consider the following sequence of instructions:

b := a + 1
a := b + c

If the variable 'a' is not going to be used in any part of the later set of instructions, then 'a' is said to be dead and hence the above sequence of instructions can be reduced to just the single statement:

b := a + 1

Unreachable code will also account for dead code and hence they could also be eliminated. Consider a branch instruction in which a particular path is never going to be taken. Then the instructions belonging to this branch could be completely eliminated as they are never going to be computed.

29.3.1.3 Renaming Temporary variables

Temporary variables that are dead at the end of a block can be safely renamed and can be eliminated if necessary. Consider the following sequence of instructions:

1. **t1 := b + c**
2. **t2 := a - t1**
3. **t1 := t1 * d**
4. **d := t2 + t1**

In this scenario, t1 is being reused at line numbers (1) and (3) and this could be conveniently renamed to another variable as shown in the instruction sequence below:

1. **t1 := b + c**
2. **t2 := a - t1**
3. **t3 := t1 * d**
4. **d := t2 + t3**

Here, t1 in the LHS of line (3) is renamed as t3 and this value is used at line number (4).

29.3.1.4 Independent statements can be reordered

For improving on the register usage, certain statements could be reordered. The algorithm to do this will be seen in later modules. However, consider the following example involving 4 instructions.

1. **t1 := b + c**
2. **t2 := a - t1**
3. **t3 := t1 * d**
4. **d := t2 + t3**

The above instructions could be changed as follows which doesn't alter the syntactic and semantic structure of the computation.

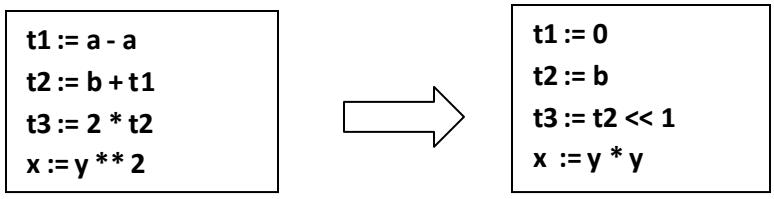
1. **t1 := b + c**
2. **t3 := t1 * d**
3. **t2 := a - t1**
4. **d := t2 + t3**

Here, we have swapped line number (2) and (3) so that the computed value of t1 could be used effectively.

29.3.2 Algebraic transformations

In this transformation, the syntactic structure is altered to apply the mathematical identities. Instructions involving addition by '0' or multiplication by '1' could be eliminated. Similarly, multiplying by '2' could be considered as a left-shift or addition of a variable with itself. Exponentiation by '2' could be considered as multiplying a variable with itself.

Following is the sequence of instructions and their corresponding algebraic transformed instructions.



The first computation results in '0' and hence t1 is assigned '0'. As t1 is '0' the addition operation in the next instruction is removed. The third instruction of multiplying by 2 is changed to left-shift by 1. The fourth instruction of computing y^2 is replaced with multiplying 'y' with itself.

29.4 Next-Use Information.

Next-use information is needed for dead-code elimination and register allocation. Next-use is computed by a backward scan of a basic block. The next-use information will indicate the statement number at which a particular variable that is defined in the current position will be reused. The following are the steps involved in generating next-use information:

Consider a statement number 'i' with the following instruction.

- *i*: *x* := *y* op *z*

We need to add liveness/next-use information for the variables *x*, *y*, and *z* to statement *i* using the following rules:

  - Set *x* to "not live" and "no next use" – As 'x' is defined here it's set to not live.
  - Set *y* and *z* to "live" and the next uses of *y* and *z* to *i*. The variables 'y' and 'z' are being used in this statement and hence has their next use at 'i' and as they are used here these variables are set to live.

Table 29.1 gives an example sequence of instructions and their nextuse and liveness computation. Consider statement 'i' precedes statement 'j'. We start computing the information from statement 'j'

Table 29.1 Example live and nextuse() computation

Statement number	**Instruction**	**Liveness and Nextuse initialized**	**Modified liveness and nextuse after statement 'j'**	**Modified liveness and nextuse after statement 'i'**	**Comments**
i	**a := b + c**	*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none	*live*(**a**) = true *nextuse*(**a**) = *j* *live*(**b**) = true *nextuse*(**b**) = *j* *live*(**t**) = false *nextuse*(**t**) = none	*live*(**a**) = false *nextuse*(**a**) = none *live*(**b**) = true *nextuse*(**b**) = *i* *live*(**c**) = true *nextuse*(**c**) = *i* *live*(**t**) = false *nextuse*(**t**) = none	'a' is said to false and no nextuse. 'b' and 'c' are said to live at statement 'i' and their next use is at 'i' . 't''sliveness

					and nextuse remains as it is computed in statement 'j'
j	**t := a + b**	*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none	*live*(**a**) = true *nextuse*(**a**) = *j* *live*(**b**) = true *nextuse*(**b**) = *j* *live*(**t**) = false *nextuse*(**t**) = none	*live*(**a**) = true *nextuse*(**a**) = *j* *live*(**b**) = true *nextuse*(**b**) = *j* *live*(**t**) = false *nextuse*(**t**) = none	'a' and 'b' are used here and hence their nextuse is at 'j' and they are said to be live. 't' is computed and hence nextuse is none and live information is false

The usage of nextuse and live information will be discussed in the subsequent modules along with the code generation algorithm.

Summary: In this module we discussed the conversion from three-address code into basic blocks. The structure preserving and algebraic transformations that are possible in basic blocks also discussed. The module was completed with the computation of next-use and live information.