# EXAMPLE OF INSTRUMENTED-CODE AND BLOCK-TREE

**Three Kinds of Instrumentations:** (1) Printing "start ⟨block#⟩" at the beginning of each function-block.
(2) Printing "end ⟨block#⟩" before each return/end-statement.
(3) Printing "⟨block#⟩" at the beginning of other blocks.
May need to add some { }-pairs to insert the instrumentation statements.

**Example.** Assume that `IndexOfFirstPositive` is the 2nd function in a source program-file, and thus it is assigned the block# 2.

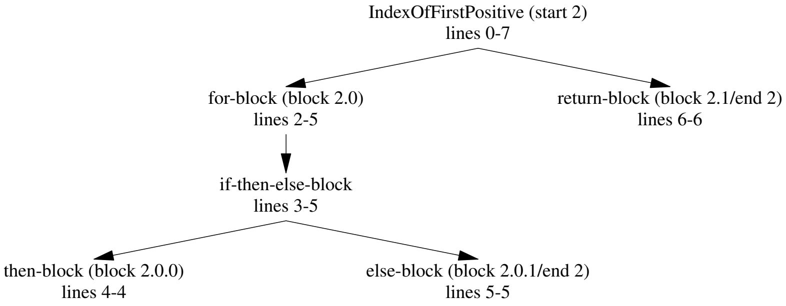
- Note the two layers of { } added for the else-statement.
- Note the blank-space before an added '{' and the statements within the inserted { }.

Input program file

```
0. int IndexOfFirstPositive(int *vector, int size)
1. {int i;

2.  for (i=0; i<size; i++)

3.      if (vector[i] <= 0)
4.         printf("vector[%d]=%d\n", i, vector[i]);


5.      else return(i);





6.  return(-1);




7. }
```

Instrumented program file

```
int IndexOfFirstPositive(int *vector, int size)
{int i;
 fprintf(fpExProf, "start 2\n");
 for (i=0; i<size; i++) {
     fprintf(fpExProf, "block 2.0\n");
     if (vector[i] <= 0) {
        fprintf(fpExProf, "block 2.0.0\n");
        printf("vector[%d]=%d\n", i, vector[i]);
     }
     else {fprintf(fpExProf, "block 2.0.1\n");
           {fprintf(fpExProf, "end 2\n");
            return(i);
           }
     }
 }
  {fprintf(fpExProf, "block 2.1\n");
   {fprintf(fpExProf, "end 2\n");
    return(-1);
   }
  }
}
```




The Block-Tree of IndexOfFirstPositive; the return-blocks are needed for analysis of the instrumentation-output of a recursive function.

| Block-tree is an abstraction of the source-program.<br>(Different source-programs may have the same block-tree.) |
|---|

**EXERCISE**

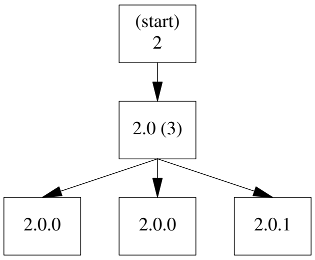
1. Show the instrumented-version of the instrumented code itself shown above; also show the new block-tree. What relationship do you see between this block-tree and that of `IndexOfFirstPositive()`? Is there a way to simplify the former to the latter?

# THE EXECUTION-TREE FOR AN INPUT

**Output of Instrumentation of** `IndexOfFirstPositive()`:

Input data: `vector[] = [-1, -5, 3, 4]` and size = 4.

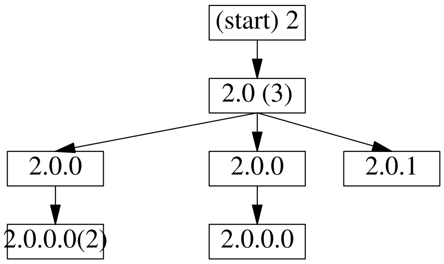
| Regular output | Instrumentation output | Comment |
|---|---|---|
| | `start 2` | |
| | `block 2.0` | For-loop (block#2.0) is iterated 3 times. |
| `vector[0]=-1` | `block 2.0.0` | The first two loop-iterations involved the if-part (block#2.0.0). |
| | `block 2.0` | |
| `vector[1]=-5` | `block 2.0.0` | |
| | `block 2.0` | |
| | `block 2.0.1` | The last loop-iteration involved the else-part (block#2.0.1). |
| | `end 2` | |




Understanding the dynamic behavior is more complex because of:
(1) The shifting of control during an execution because of function-calls, loops, etc.
(2) The execution-tree may be different for different inputs.

**EXERCISE**

1. Show the execution-tree for the input data: `vector[] = [-1, -4, -2, -3]` and `size = 4`.
2. Can the following execution-tree result from some function? If so, explain how (i.e., give the pseudocode/code of a suitable function).




The two children 2.0.0 of 2.0 are not combined here because their subtrees are not the same.

3. Show the instrumented-code and the block-tree for the following function (assume this to be function #1 in a source program-file).

```
 1. //returns -1 if x is not found, and returns the array-index (>= 0) corresponding to x
 2. int BinarySearch(int x, int low, int high, int *nums) //0 <= low <= high
 3. {int mid;
 4.  if ((x < nums[low]) || (x > nums[high]))
 5.     return(-1);
 6.  else {mid = (low + high)/2; //mid >= low
 7.        if (x == nums[mid])
 8.           return(mid);
 9.        else if (x < nums[mid])
10.                BinarySearch(x, low, mid-1, nums); //mid-1 >= low
11.        else BinarySearch(x, mid+1, high, nums);   //mid+1 <= high
12.       }
13. }
```

Note that line 10 above amounts to "`return(BinarySearch(...));`" and likewise for line 11. The following instrumentation of a statement "return(function-call)" does not work.

{fprintf(fpExProf, "end..."); return(function-call);}

It incorrectly makes the "end..." appear before the function-call in the execution-profile file. A possible solution is the instrumentation (we have to add declaration for temp within the block):

{temp = function-call; fprintf(fpExProf, "end..."); return(temp);}.

4. Show the instrumentation-output for the input case: $x = 5$, $low = 0$, $high = 5$, and $nums =$ `[-1, 2, 7, 9, 15, 16]`. Also, show the associated execution-tree.

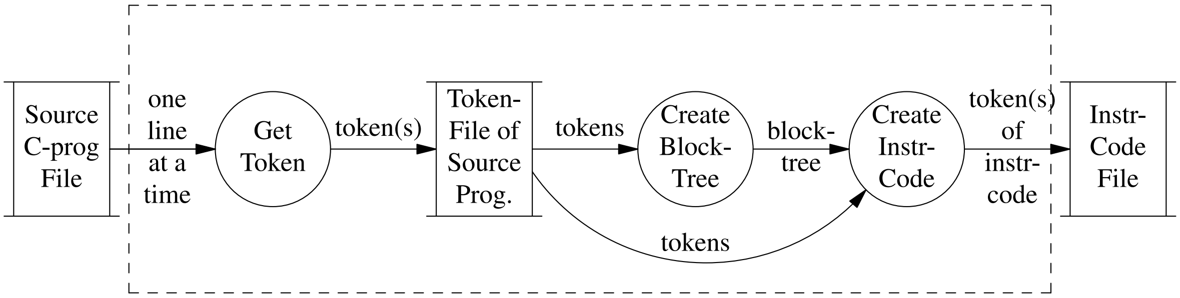
5. What information do we need to keep at *each* node of the block-tree so that we can compute:

   (a) the maximum number of lines of code in that block that can be covered in any execution.

   (b) the percentage of lines of code in that block that is covered in a given execution.

   Use the instrumented code itself as the original source-file to explain how you can compute (a)-(b) using those information. (Do you have to make any assumption about the coding style to make this possible? For example, if we assume that the program is written in C, then do we assume that the if-condition does not spread over two or more lines?) Should we count the number of lines or number of statements? What problems might arise if we want to count the number of expressions evaluated instead of the number of lines of code executed?

# A DATAFLOW DIAGRAM FOR THE CODE-INSTRUMENTATION SOFTWARE




- Not so clear that one block-tree is made from the whole Token-file.
- Not so clear that the whole Token-File is read twice, once by Create-Block-Tree functions and once by Create-Inst-Code function.

**Question:** (1) Is there a way to fix these problems?

(2) What is a simple way to avoid reading Token-File twice?

**Not Represented in DFD:**

The following features of an actual implementation are not captured in this DFD:

- Create-Block-Tree function (which is recursive) calls Get-Tokens every time it needs a token (Get-Token returns one token at a time), and terminates after creating the whole Block-tree.
- Token-File is created by Get-Token function only to help its debugging, and thus its creation can be turned off once the program if debugged.
- Likewise, a Block-Tree-File (not shown here) is created by Create-Block-Tree function to help its deugging.
- The Create-Inst-Code function actually does not exist as a separate entity and its operations are built within the Create-Block-Tree function in that as the Block-tree is created the instrumented code is also generated token-by-token and occassionaly inserting the instrumentation-lines.

**Data Structure for a token:**

In addition to the token-string, we keep:

- Source-line number (this is used for inserting "new-line" in instrumented-code when needed).
- Start-position of the token in the line (this helps in insertion of appropriate number of blanks when needed).

The above DFD suggests a possible alternative (simpler but less efficient) implmentation of an in strumentation-software.

**EXERCISE**

1. What would be the disdavantage of creating a token-file first, then building the whole block-tree, and finally creating the instrumented-code?