AUTHORS:-
AMOL (CS14B035)
BIKASH (CS14B039)

19th JAN, 2016

Program Analysis

Program Analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at runtime when executing a program on a computer. Program analysis is generally used for two purpose: program optimization and program correctness. Program optimization is the process of modifying a program to make it work more efficiently or use fewer resources without changing the semantics of the program. Program correctness focuses on ensuring that the program does what it is supposed to do.
We can also define program analysis as a technique to identify "*interesting aspects*" of a program's representation for an *end goal*. Here program representation can be anything, for example source code, Abstract Syntax Tree (AST), binary, but we use the predefined program's representation for performing the analysis so that we can use the already available framework.

Following table shows some of the end goals and the corresponding "interesting aspects".

End goal	**Interesting aspect**
Dead code elimination	Reachability
Constant propagation	use-def
Security	Array index range, dangling pointers
Parallelization	Data dependence, SIMD opportunities
Debugging	Slice
Cache performance	Memory access pattern
Memory reduction	Live ranges

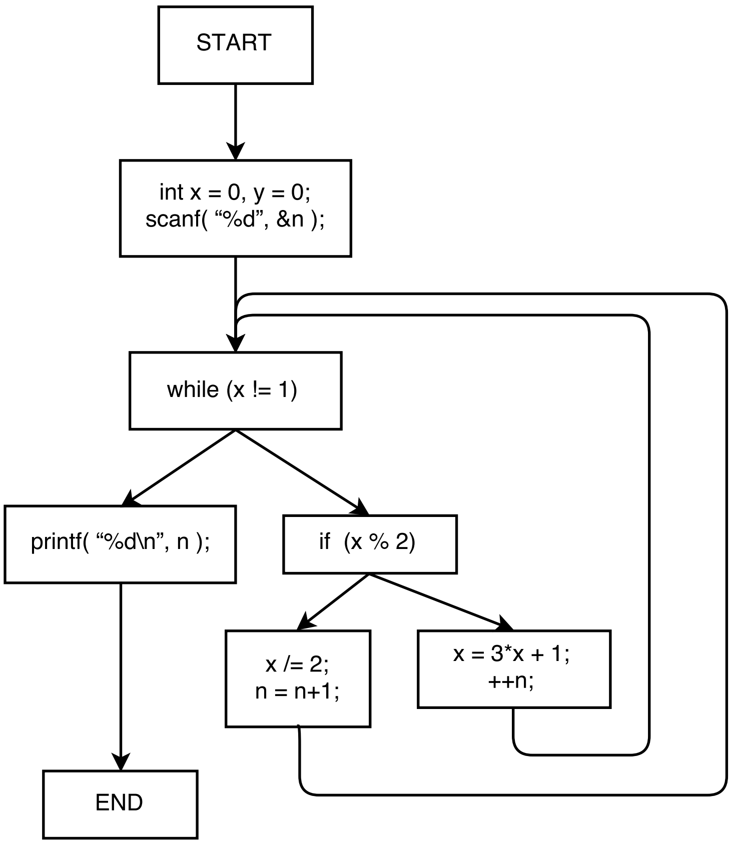
Control Flow Graph (CFG)

Control flow graph is the representation of control flow of a program in graph notation such that it considers all the paths that might be traversed during its execution. Each node of CFG is a basic block. Edge of CFG indicate which basic block can follow which block.
Basic blocks are maximal sequences of consecutive instructions such that flow of control can enter the basic block through the first instruction and it can leave or halt at the last instruction.

Q. Convert the following code into CFG.

```
int x = 0, n = 0;

scanf( “%d”, &n );

while (x != 1) {

        if (x % 2) {

                x = 3*x + 1;

                ++n;

        } else {

                x /= 2

                n = n + 1;

        }

}

printf( “%d\n”, n );
```

Ans:-



Data Flow Analysis (DFA)

Data flow analysis is a technique for gathering information about the possible set of values calculated at various program points. It is a flow-sensitive analysis, which means it is sensitive to the control flow of a program. Different data flow analysis gathers different information. These information are known as data flow facts. The data flow facts gathered from analysis are often used by compilers to optimize the program. For example, constant-propagation analysis seeks to determine whether all assignments to a particular variable that may provide the value of that variable at some particular point necessarily give it the same constant value. If so, a use of the variable at that point can be replaced by the constant and thus reduces the variables for register allocation.

A program can have potentially infinite execution paths. To enumerate all those paths are not possible. So we should approximate in such a way such that the analysis remains conservative. An analysis is said to be conservative if approximations that we make does not change the semantics of the program. For instance, if we use **reaching definitions** for constant folding, it is safe to think a definition reaches when it doesn't (we might think x is not a constant, when in fact it is and could have been folded) , but not safe to think a definition doesn't reach when it does (we might replace x by a constant, when the program would at times have a value for x other than that constant).

Interprocedural Analysis:-

```
Caller                  Callee
main() {                f() {
        f();                    ....
}                               ....
                                f();
                        }
```

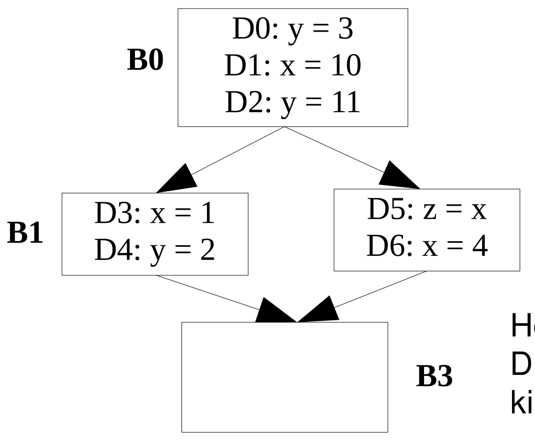
Problems arises in this case:-

1. Function *f()* can return to *f()* itself as well as it can return to function *main()*. It’s not possible to tell which path will take in a static analysis.
2. In this type of recursive function (or in some loops), it’s not possible to tell how many times the function will be called (or the loops will be executed). As we cannot enumerate all the possible paths, we have to make a conservative approximation.

Reaching Definition

Reaching definition is a data-flow analysis which statically determines which definitions may reach a given point in the code. A definition **d** reaches a program point **p** if there exists a path from the point immediately following **d** to **p** such that **d** is not killed along the path. We kill a definition of a variable **x** if there is any other definition of **x** anywhere along the path. The data flow direction in this analysis is forward. It means that the data-flow facts will flow in the direction of control flow.

Example:-



<u>Definitions that can reach B3</u>
D2, D3, D4, D5, D6

Here D1 can't reach B3 as D3 kills D1 on the path B0B1B3 and D6 kills D1 on the path B0B2B3

Some important terms:-

- **IN(B) :-** It is a set of data flow facts that are entering the block B.
- **OUT(B) :-** It is a set of data flow facts that are leaving the block B.
- **GEN(B) :-** It is a set of data flow facts that are being generated in the block B but not being killed internally.
- **KILL(B) :-** It is a set of data flow facts that are being killed in the block B. It includes all the definition of variables in other basic blocks which are redefined in block B.

Transfer functions:-

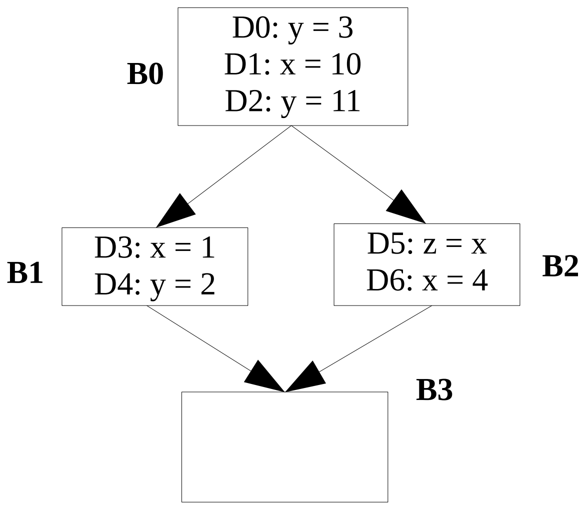
- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) - kill(B))

Algorithm for Reaching Definitions:

```
for each basic block B {
        compute gen(B) and kill(B)
        out(B) = { }
}
do {
        for each basic block B {
                in(B) = U out(P)        where P ∈ pred(B)
                out(B) = gen(B) U (in(B) – kill(B))
        }
} while in(B) changes for any basic block B
```

Q. ***Why should we initialize OUT with empty set*?**
Ans:- We can initialize OUT with a non-empty set, but then we will be ending up computing a superset of the static analysis set, which makes our algorithm imprecise.

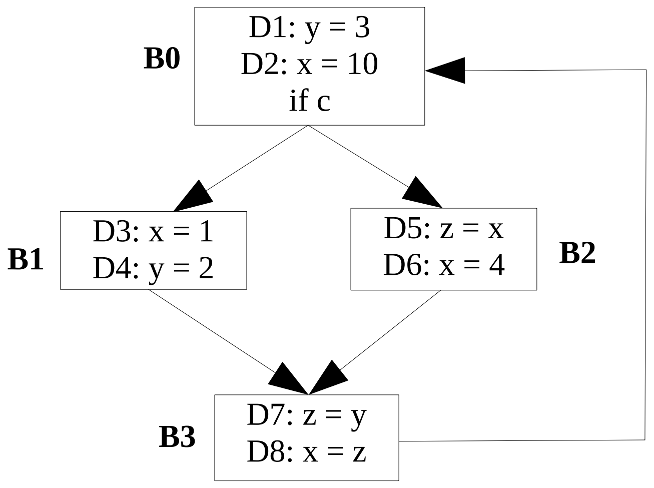
Example 1:-



gen(B0) = {D1, D2}	kill(B0) = {D3, D4, D6}
gen(B1) = {D3, D4}	kill(B1) = {D0, D1, D2, D6}
gen(B2) = {D5, D6}	kill(B2) = {D1, D3}
gen(B3) = { }	kill(B3) = { }

	in1	out1	in2	out2	in3	out3
B0	{}	{D1,D2}	{}	{D1,D2}	{}	{D1,D2}
B1	{}	{D3,D4}	{D1,D2}	{D3,D4}	{D1,D2}	{D3,D4}
B2	{}	{D5,D6}	{D1,D2}	{D2,D5,D6}	{D1,D2}	{D2,D5,D6}
B3	{}	{}	{D3,D4,D5,D6}	{D3,D4,D5,D6}	{D2,D3,D4,D5,D6}	{D2,D3,D4,D5,D6}

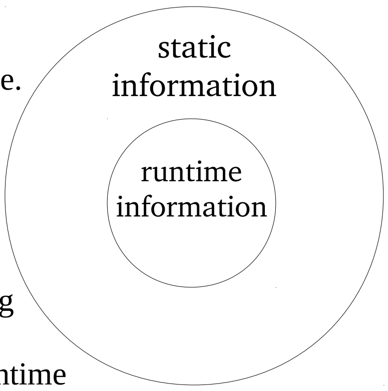
Example 2:-



gen(B0) = {D1, D2}	kill(B0) = {D3, D4, D6, D8}
gen(B1) = {D3, D4}	kill(B1) = {D1, D2, D6, D8}
gen(B2) = {D5, D6}	kill(B2) = {D2, D3, D7, D8}
gen(B3) = {D7, D8}	kill(B3) = {D2, D3, D5, D6}

	in1	**out1**	**in2**	**out2**	**in3**	**out3**	**in4**	**out4**
B0	{}	{D1,2}	{D7, 8}	{D1,2,7}	{D4,7,8}	{D1,2,7}	{D1,4,7,8}	{D1,2,7}
B1	{}	{D3,4}	{D1,2}	{D3,4}	{D1,2,7}	{D3,4,7}	{D1,2,7}	{D3,4,7}
B2	{}	{D5,6}	{D1,2}	{D1,5,6}	{D1,2,7}	{D1,5,6}	{D1,2,7}	{D1,5,6}
B3	{}	{D7,8}	{D3,4,5,6}	{D4,7,8}	{D1,3,4,5,6}	{D1,4,7,8}	{D1,3,4,5,6,7}	{D1,4,7,8}

Static Information Vs Runtime Information:-



Runtime information is what we want in an analysis to be more precise. But as we know it is often not possible to have those information in a static analysis. So what we do is take a conservative approximation, which is basically taking account of less precise but sound information. For example, in reaching definitions, if there is branching in a basic block and from one of the branch a definition reaches a point. In static analysis we include that definition as reaching even though it may not reach in some of the execution path. So the information generated in the static analysis are less precise that the runtime information. As we include more definitions which may not reach the point, it becomes more conservative but becomes less precise.

Q. ***How can we optimize the algorithm for reaching definitions*?**
Ans:- We can optimize the algorithm by having a worklist which keeps track of only those basic blocks, the OUT of whose predecessor gets change. It can be done by including the successor of a basic block if it’s OUT get changes. The algorithm terminates if there are no element in the worklist.

Q. ***Explain whether parenthesization of [IN(B) - KILL(B)] is necessary or not?***
Ans:- If we define KILL to contain definitions only from the other basic blocks, we can get rid of the parentheses. The same equations would work with and without parentheses, as long as GEN is precisely defined (that is, GEN takes care of intra-basic-block KILLs).
However, in a general setting, information from other basic blocks may not be getting tracked by an analysis. For instance, one may track only that the value of a variable is constant or not (without keeping track of where it is defined as a constant), or that the points-to set of a pointer is {x, y, z} without keeping track of where it is coming from. In such cases, we do not want the information (data flow facts) generated in the current basic block to be killed. Hence, we use [IN - KILL] with parenthesization.