Introduction to Global Data Flow Analysis

Global Data Flow Analysis

- **To efficiently optimize the code** compiler **collects all the information about the program and distribute this information to each block of the flow graph.** This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

- Based on the local information a compiler can perform some optimizations. For example, consider the following code:
- x = a + b;
- x = 6 * 3
- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

For example, consider the following code:

```
a = 1;
b = 2;
c = 3;
if (....)
    x = a + 5;
else
    x = b + 4;
c = x + 1;
```

- In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

- Global data flow analysis is used to solve a specific problem "**User definition chaining**".
- Problem: Given that the identifier A is used at a point p in the program. At what point could the value of A have been defined.
- **Reaching definition:** Determination of each variable whenever they are declared in the program.

For this it follows following steps-

1. Assign a distinct number to each definition such as d1, d2, d3,…..
2. For each variable i, make a list of all definition in entire diagram where it is used.

For i, definitions d1 and d3 are used.

For j, definitions d2, d4 and d5 are used.

3. For each basic block calculate the following:

a. **GEN[B]-** The set GEN[B] consists of all definitions generated in block B**.**

GEN[B1] – d1, d2

b. **KILL[B]-** The set of all the definition outside block B that define the same variables having definitions in block B also.

KILL[B1]- d3, d4, d5

4. For all basic block compute the following:

a. IN[B]: The set of all definitions reaching the point just before the first statement of block B.

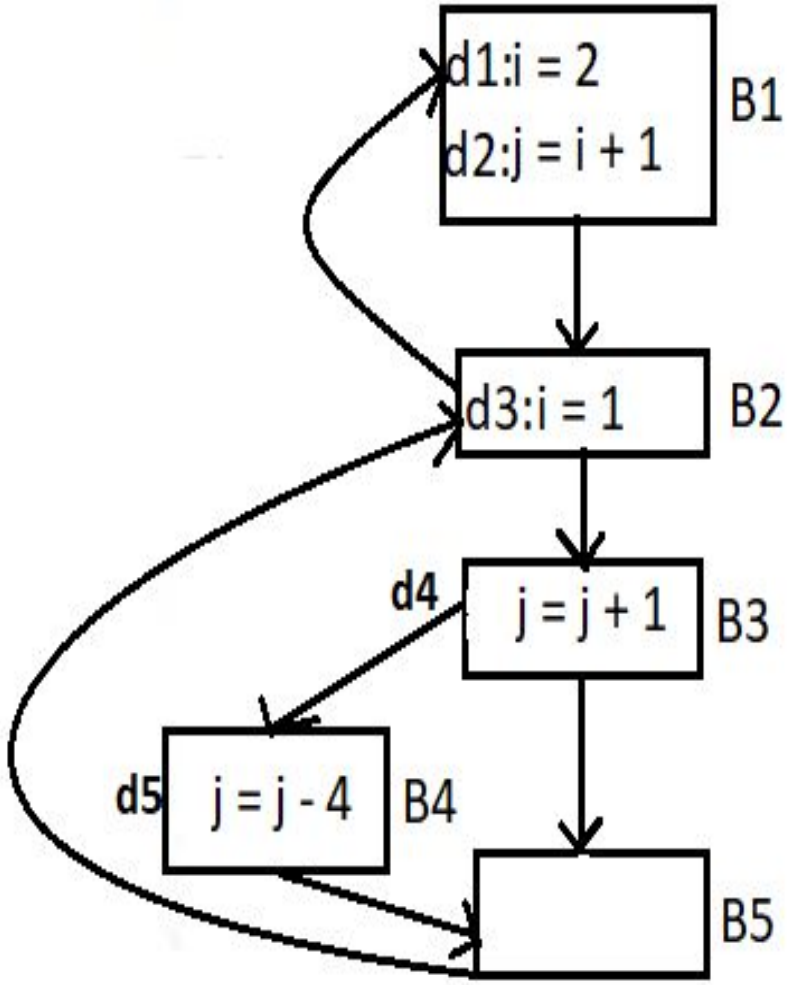
b. OUT[B]: The set of all definitions reaching the point just after the last statement of block B.

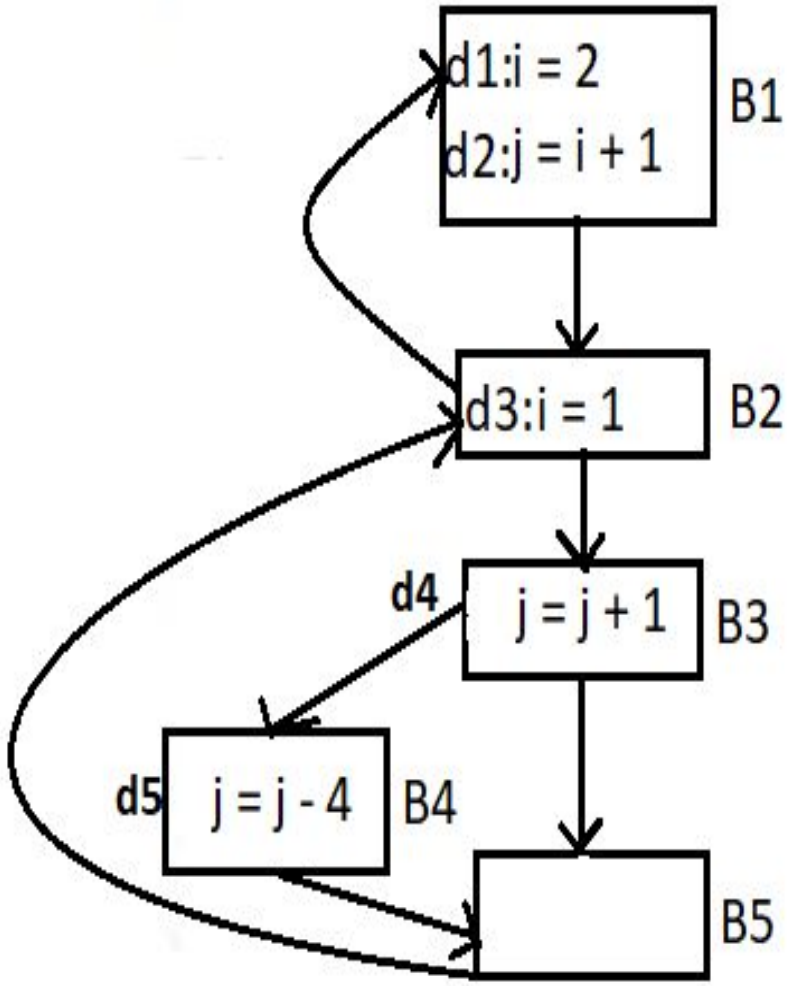
Data Flow Equation:

OUT[B] = IN[B] – KILL[B] U GEN[B]

OUT[B] = { IN[B] AND (~KILL[B]) U GEN[B]

IN[B] = U OUT[B] (Union of all out B from all previous blocks)





1. Find GEN and KILL for each block.
2. Find IN and OUT for reaching definitions.

Block B	GEN [B]	Bit vector d1 d2 d3 d4 d5	KILL [B]	Bit vector d1 d2 d3 d4 d5
B1	[d1, d2]	11000	[d3, d4, d5]	00111
B2	[d3]	00100	[d1]	10000
B3	[d4]	00010	[d2, d5]	01001
B4	[d5]	00001	[d2, d5]	01010
B5	Φ	00000	Φ	00000

Initially Pass 0,

- IN[B] = Φ, OUT[B] = GEN[B]

Block [B]	IN [B]	OUT [B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass 1:

Block [B]	IN [B]	OUT [B]
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

For Pass 1:
IN [B1] = OUT [B2] = 00100 (refer Pass 0)
OUT [B1] = {IN [B1] AND (~KILL [B1])} OR GEN [B1]
= (00100 AND 11000) OR 11000
= 11000

IN [B2] = OUT [B1] UNION OUT [B5]
= 11000 UNION 00000 = 11000
OUT [B2] = {IN[B2] AND (~KILL [B2])} OR GEN [B2]
= 11000 AND 01111 OR 00100
= 01100

Similarly calculate the values of Pass 2, Pass 3, Pass 4……., until all the values of two passes are same.

Parameter Passing

- All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments.

- In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition).

1. Call By Value

- In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable).
- The value is placed in the location belonging to the corresponding formal parameter of the called procedure.
- This method is used in C and Java, and is a common option in C++, as well as in most other languages.
- Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

2. Call- by-Reference

- In call- by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
- Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller.
- Changes to the formal parameter thus appear as changes to the actual parameter.
- If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own.
- Changes to the formal parameter change this location, but can have no effect on the data of the caller.

3 Call By Name

- A third mechanism - call-by-name - was used in the early programming language Algol 60.
- It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct).
- When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

Storage Allocation strategies

1 Static Versus Dynamic Storage Allocation:

- The layout and allocation of data to memory locations in the run-time environment are key issues in storage management.
- The two adjectives static and dynamic distinguish between compile time and run time, respectively.
- We say that a storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
- Conversely, a decision is dynamic if it can be decided only while the program is running.

- Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. Stack storage. Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.

2. Heap storage. Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage. The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

- To support heap management, "garbage collection" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly

2 Stack Allocation of Space:

- Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.
- This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

2.1 Activation Tree

- Stack allocation would not be feasible if procedure calls, or activations of procedures, did not nest in time.
- We can represent the activations of procedures during the running of an entire program by a tree, called an *activation tree.*
- Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- Notice that one child must finish before the activation to its right can begin.

- The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

2. The sequence of returns corresponds to a postorder traversal of the activation tree.

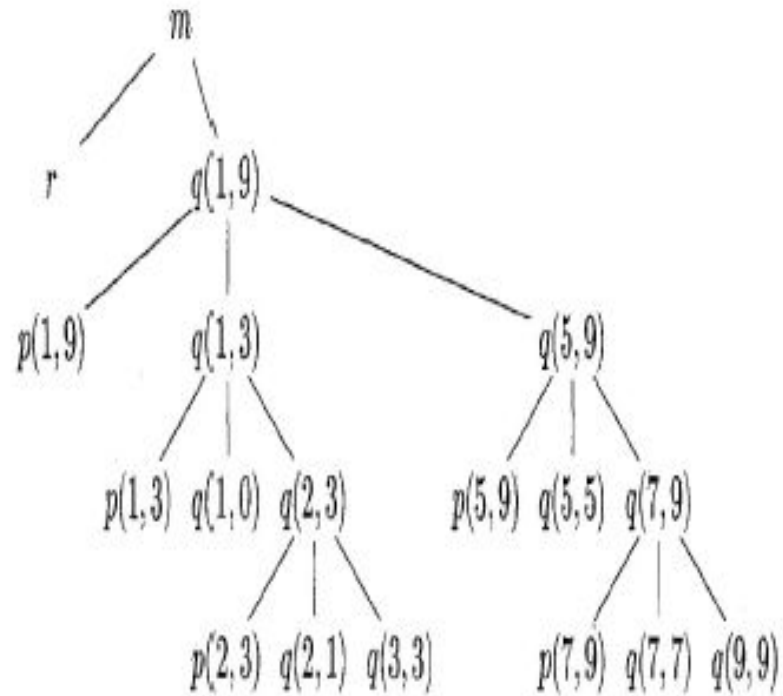
3. Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree. Then the activations that are currently open (live) are those that correspond to node N and its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root, and they will return in the reverse of that order.

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] s
       a[m..p-1] are less than v, a[p] = v, and a[p+1..
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Possible activations for the program

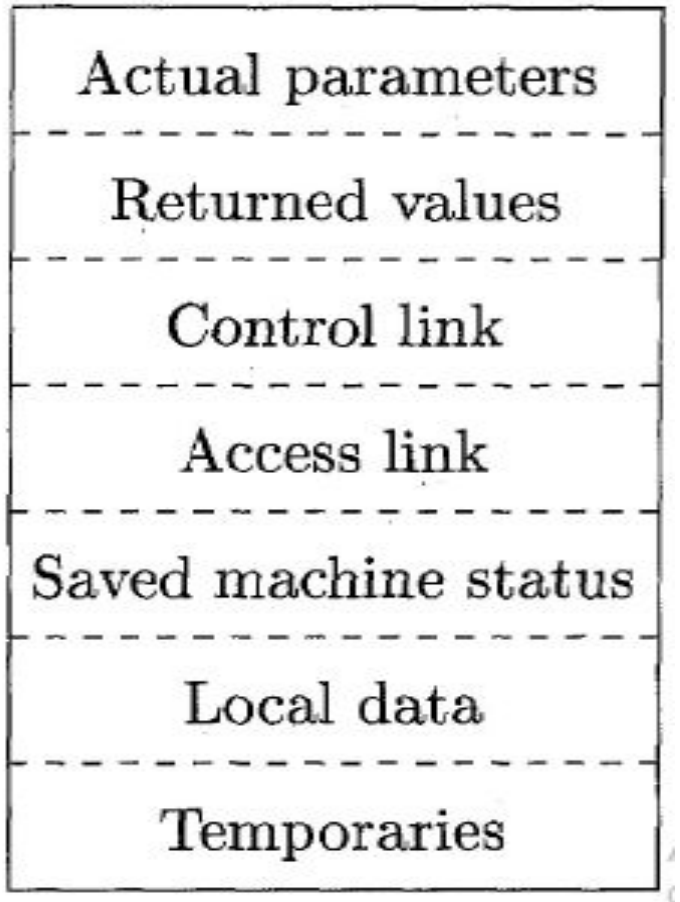


Activation tree representing calls during an execution of quicksort

2.2 Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack.

- The contents of activation records vary with the language being implemented.
- Here is a list of the kinds of data that might appear in an activation record:



1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

5. A control link, pointing to the activation record of the caller.

6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

2.3 Calling Sequences

- Procedure calls are implemented by what are known as calling sequences, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

3 Heap Management

- The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.
- While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them.
- For example, both C++ and Java give the programmer ***new*** to create objects that may be passed - or pointers to them may be passed - from procedure to procedure, so they continue to exist long after the procedure that created them is gone.
- Such objects are stored on a heap.

3.1 The Memory Manager

- Memory manager is the subsystem that allocates and deallocates space within the heap; it serves as an interface between application programs and the operating system.
- The memory manager keeps track of all the free space in heap storage at all times.

It performs two basic functions:

1. Allocation.

- When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size.
- If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system.
- If space is exhausted, the memory manager passes that information back to the application program.

2. Deallocation.

- The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.
- Memory managers typically do not return memory to the operating system, even if the program's heap usage drops.

Memory management would be simpler if

(a) all allocation requests were for chunks of the same size, and

(b) storage were released predictably, say, first-allocated first-deallocated.

- There are some languages, such as Lisp, for which condition (a) holds; pure Lisp uses only one data element - a two pointer cell - from which all data structures are built.
- Condition (b) also holds in some situations, the most common being data that can be allocated on the run-time stack.

- However, in most languages, neither (a) nor (b) holds in general.
- Rather, data elements of different sizes are allocated, and there is no good way to predict the lifetimes of all allocated objects.
- Thus, the memory manager must be prepared to service, in any order, allocation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

Here are the properties we desire of memory managers:

1. Space Efficiency: A memory manager should minimize the total heap space needed by a program.

- Doing so allows larger programs to run in a fixed virtual address space.
- Space efficiency is achieved by minimizing "fragmentation.“

2. Program Efficiency: A memory manager should make good use of the memory subsystem to allow programs to run faster.

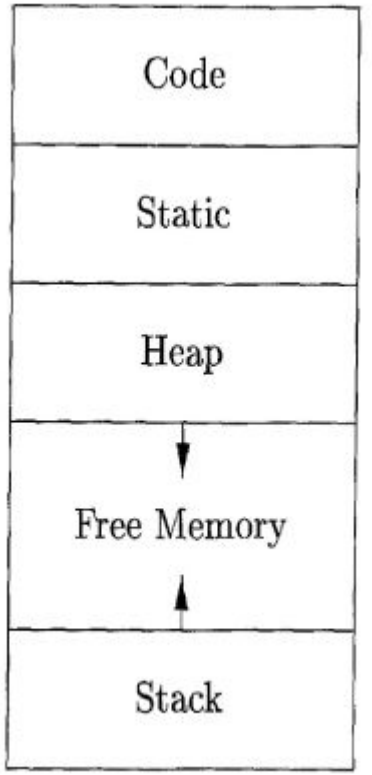
- The time taken to execute an instruction can vary widely depending on where objects are placed in memory.
- Fortunately, programs tend to exhibit "locality," a phenomenon which refers to the nonrandom clustered way in which typical programs access memory.
- By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster.

3. Low Overhead: Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible.

- That is, we wish to minimize the *overhead* - the fraction of execution time spent performing allocation and deallocation.

Storage Organization

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.
- The run-time representation of an object program in the logical address space consists of data and program areas.
- A compiler for a language like C++ on an operating system like Linux might subdivide memory in this way.



Subdivision of run-time memory into code and data areas

- The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area Code, usually in the low end of memory.

- Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called Static.

- One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.

- To maximize the utilization of space at run time, the other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space.

- These areas are dynamic; their size can change as the program executes.
- These areas grow towards each other as needed. The stack is used to store data structures called activation records that get generated during procedure calls.
- An activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs.
- When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.
- Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- For example, C has the functions **malloc** and **free** that can be used to obtain and give back arbitrary chunks of storage.
- The heap is used to manage this kind of long-lived data.

Runtime Environments

- A compiler must accurately implement the abstractions embodied in the source language definition.
- These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs.
- The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.

- To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

- This environment deals with a variety of issues such as the

- layout and allocation of storage locations for the objects named in the source program,
- the mechanisms used by the target program to access variables,
- the linkages between procedures,
- the mechanisms for passing parameters,
- and the interfaces to the operating system, input/output devices, and other programs.