Compiler Design

Prepared by

Ch. Chakradhara Rao

Syllabus

UNIT-1

Language Processors: Introduction, the structure of a compiler, the science of building a compiler, programming language basics.

Lexical Analysis: The role of the lexical analyzer, input buffering, recognition of Tokens, the lexical analyzer generator lex program specification, finite automata, from regular expressions to automata, design of a lexical-analyzer generator, optimization of DFA-based pattern matchers.

UNIT-2

Syntax Analysis: Introduction, context-free grammars (CFG), derivation, top-down parsing, recursive and non recursive top down parsers, bottom-up parsing, Operator precedence parser,

Introduction to LR parsing: simple LR parser, more powerful LR parsers, using ambiguous grammars, parser hierarchy, and automatic parser generator YACC tool.

Syllabus

UNIT-3

Syntax-Directed Definitions: Introduction, evaluation orders for SDD's, applications of syntax-directed translation, syntax-directed translation schemes, and implementing L-attributed SDD's.

Intermediate-Code Generation: variants of syntax trees, three-address code, types and declarations, type checking, control flow statements, switch-statement, and procedures.

UNIT-4

Run-Time Environments: Storage organization, stack allocation of space, access to nonlocal data on the stack, heap management, introduction to garbage collection, introduction to trace based collection.

Machine Independent Code optimizations: The principal sources of optimization, introduction to data-flow analysis, foundations of data-flow analysis, constant propagation, partial redundancy elimination, and loop optimization in flow graphs.

Syllabus

UNIT-5

Code Generation: Issues in the design of a code generator, the target language, addresses in the target code, basic blocks and flow graphs, optimization of basic blocks, a simple code generator.

Machine Dependent Code Optimizations: peephole optimization, register allocation and assignment, code generation algorithm.

Unit – 4(Part 1)

Storage Organization

1. Storage organization

2. Stack allocation of space

3. Access to nonlocal data on the stack

4. Heap management

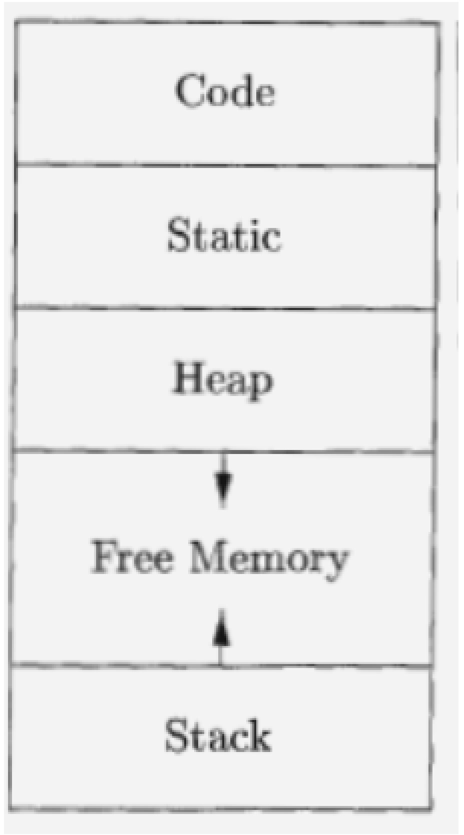
5. Introduction to garbage collection

6. Introduction to trace based collection

Storage language Issues:

Runtime Environment / Runtime Storage Management:

- Always all the programs are stored in hard disk until compilation but CPU / processor executes the program only if it is available in main memory
- OS allocates free memory to the program then compiler uses this in order to store the compiled program
- Main memory can be divided into 4 parts
- Code contains the target executable code(.obj + library files= .exe)
- Static data area stores the static and global variables
- For better utilization of free space we used heap and stack memory
- When a function is called then activation record gets created
- Stack is used to store the activation record information and it is pushed on to the top of the stack and used more often if there are many functions
- Heap is a dynamic data structure which allocates memory at run time and used more often if there are many dynamic, pointer variables
  - In C we used malloc(), calloc(), realloc(), free()
  - In java we used new(), delete()



Activation Record:

- ✓Used to manage the information needed by a single execution of a procedure
- ✓Procedure calls & returns are usually managed by runtime stack called control stack
- ✓When the activation begins then the procedure name will push on to the stack and when the activation ends / returns then it will be popped
- ✓The root of the activation record will be at the bottom of the control stack
- ✓Activation tree represents the sequence & relation between caller & called function

Model of Activation Record / Fields of Activation Record:

Actual Parameters: Holds the actual parameters of the calling function

Returned Values: Stores the result of function call

Control / Dynamic Link: Points to the activation record of calling function

Access / Static Link: Refers to the local data of called function but found in another AR

Saved Machine status: Stores the address of next instruction to be executed (prg' countr)

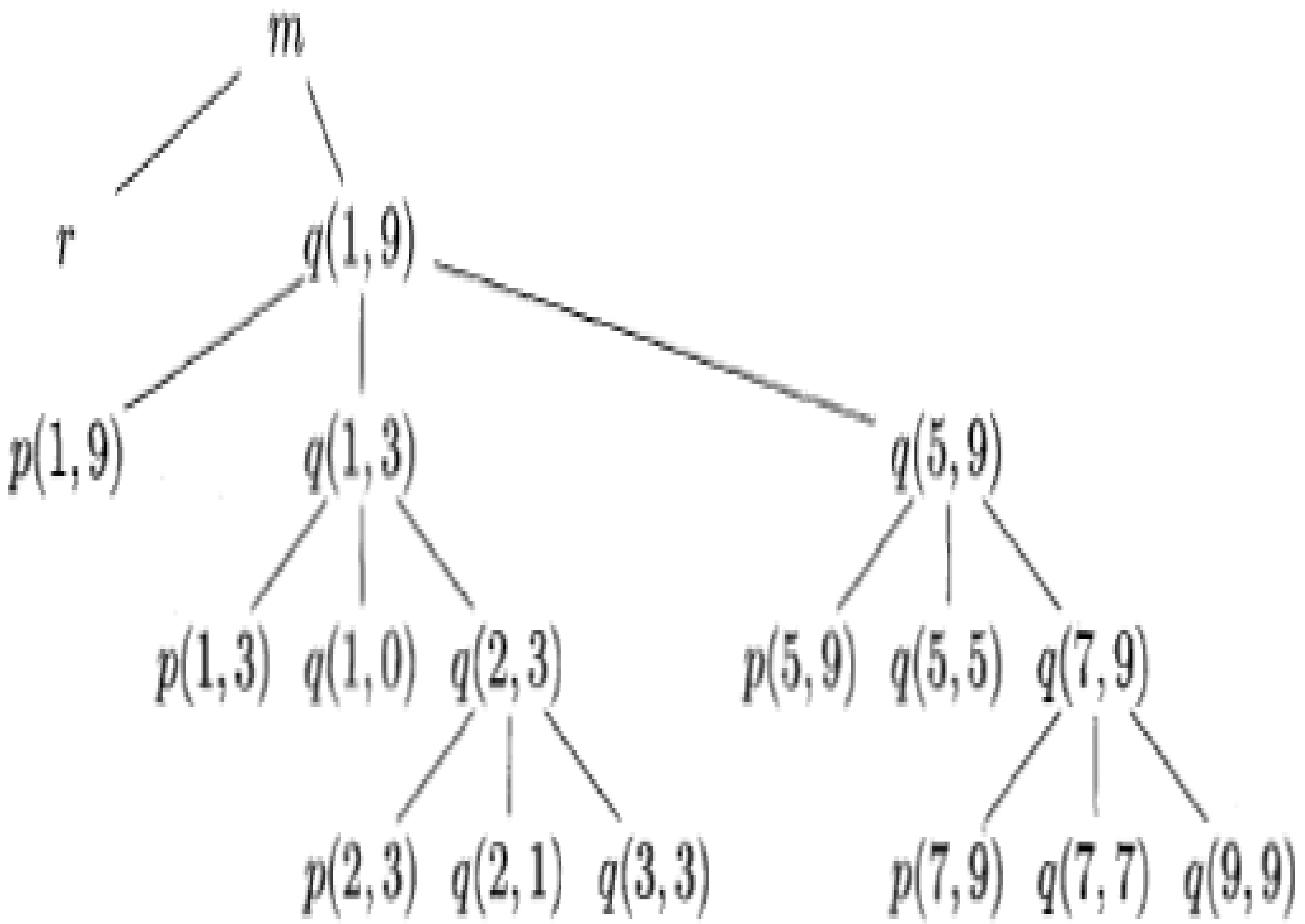
Local Variables: Holds the data that is local to the execution of corresponding function

Temporary Variables: Holds the data that arises during expression evaluation

Activation for QS:

```
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()
```

Activation tree representing calls during an execution of QS:



Storage Allocation Strategies / Storage Organization:

Static Allocation:

✓The allocation of memory can be done at compile time (variables, arrays)

✓We know how much memory is required for the declared variables before program written

✓Supports the dynamic data structure i.e., memory is created at compile time and deallocated after program completion

➢ Drawbacks:

1. Can't change the size of the memory once you fixed
2. Memory wastage if more memory allocated than the required memory
3. Causes problem if less memory allocated than the required memory
4. Insertions and deletions are very expensive (more no. of elements to be shifted)

Dynamic Allocation:

✓The allocation of memory / size can be done at run time

✓Many times we don't know how much memory is needed for the variables / arrays

✓Memory will be allocated / deleted by using new / delete operators

new operator: It creates a memory & returns memory address to the pointer   new datatype;

```
int *p;      p=new int;     *p=10;       cout<<*p;
int *p;      p=new int[5]; //p points to the address of memory location of int type
```

Storage Allocation Strategies / Storage Organization:

delete operator: It creates a memory & returns memory address to the pointer new datatype;

int *p; p=new int; *p=10; cout<<*p;

int *p; p=new int[5]; //p points to the address of memory location of int type

Stack:

- When a function is called then activation record gets created for the corresponding function and will be push down to the stack (for every function there is an AR)
- Activation record contains the locals so that they are bound to fresh storage in each AR
- Size of the memory(var's) is static and variables resides in stack can be accessed directly
- Drawbacks:
    1. Supports dynamic memory allocation but slower than static memory allocation
    2. Supports recursion but references to non-local variables after AR can't be retained

Heap:

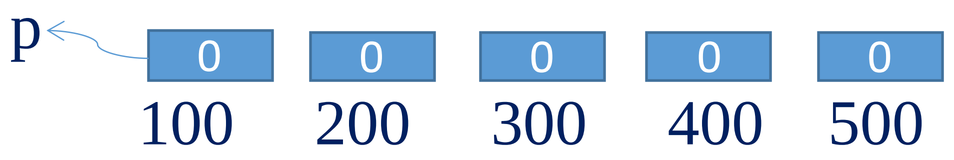
- Implements dynamic memory allocation (at any time allocation & deallocation done)
- Size of the memory(store pointer variables & used by malloc()) is dynamic and variables resides in heap can be accessed indirectly. It supports the recursion process

Dynamic Memory Allocation:

- ✓The allocation of memory can be done at run time / execution
- ✓All these functions are available in alloc.h / stdlib.h

➢**malloc()** – memory allocation

- ➢used to allocate memory for a pointer variable during execution
- ➢Allocates requested size in bytes & returns a pointer to the starting address of memory block
- ➢Allocates only one block of memory
- ➢datatype *ptr = (datatype *) malloc (bytesize); int *p = (int *) malloc (n*sizeof(int));
- ➢Accepts only one argument and returntype of malloc() is void so type casting is required
- ➢Default initial values are garbage (unpredicted) values



➢**calloc()** – contiguous memory allocation

- ➢used to allocate memory for a pointer variable during execution
- ➢Allocates multiple blocks of memory & returns a pointer to the starting / 1st block
- ➢datatype *ptr = (datatype *) calloc (no. of blocks, sizeofblock); int *p = (int *) calloc (n, sizeof(int));
- ➢Requires 2 arguments and default values are 0 (zero)

Dynamic Memory Allocation:

- **realloc()** – modification in memory
  - Used to modify the size of memory which is already allocated
  - realloc(p, bytesize);
  - int *p = (int *) malloc (3*sizeof(int)); → realloc(p, 5*sizeof(int));
- **free()** – delete memory
  - Used to destroy / release the memory which is allocated for the corresponding pointer variable
  - free(p);

Parameter Passing:

- ✓ Mechanism which is used to pass parameters to a procedure (subroutine) or function
- ✓ The communication medium among procedures is known as parameter passing
- ✓ The values of the variables from a calling procedure are transferred to the called procedure by some mechanism

r-value

- ➢ The value of an expression is called its r-value and r-values can always be assigned to some other variable
- ➢ The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator

l-value

- ➢ The location of memory (address) where an expression is stored is known as the l-value of that expression

It always appears at the left hand side of an assignment operator

Ex:

```
day = 1;    week = day * 7;        month = 1;    year = month * 12;
```

- ➢ Here the constant values like 1, 7, 12 and variables like day, week, month and year are r-values
- ➢ variables have l-values as they also represent the memory location assigned to them

```
7 = x + y;
```

- ➢ is an l-value error, as the constant 7 does not represent any memory location

Parameter Passing (Cont…):

Formal Parameter:

1. Variables that take the information passed by the caller procedure are called formal parameters
2. These variables are declared in the definition of the called function

Actual Parameter:

1. Variables whose values and functions are passed to the called function are called actual parameters
2. These variables are specified in the function call as arguments

➢**Parameter passing techniques**

1. call by value 2. call by reference 3. call by value result 4. call by name

call by value

➢The calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record

➢Formal parameters then hold the values passed by the calling procedure

➢If the values held by the formal parameters are changed, it should have no impact on the actual parameters

call by reference

➢The l-value of the actual parameter is copied to the activation record of the called procedure

➢The called procedure has the address of actual parameter & formal parameter refers to the same address

➢If the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value

Parameter Passing (Cont…):

call by value result

- It also works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends
- On function call, the values of actual parameters are copied in the activation record of the called procedure
- Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters

```
int a=1, b=2;
procedure(a,b);
Procedure(x,y) { x = x + 2;  a = x * y; x = x + 1;  }
```

- Value of a = 7 when we use call by reference
- Value of a = 4 when we use call by value result

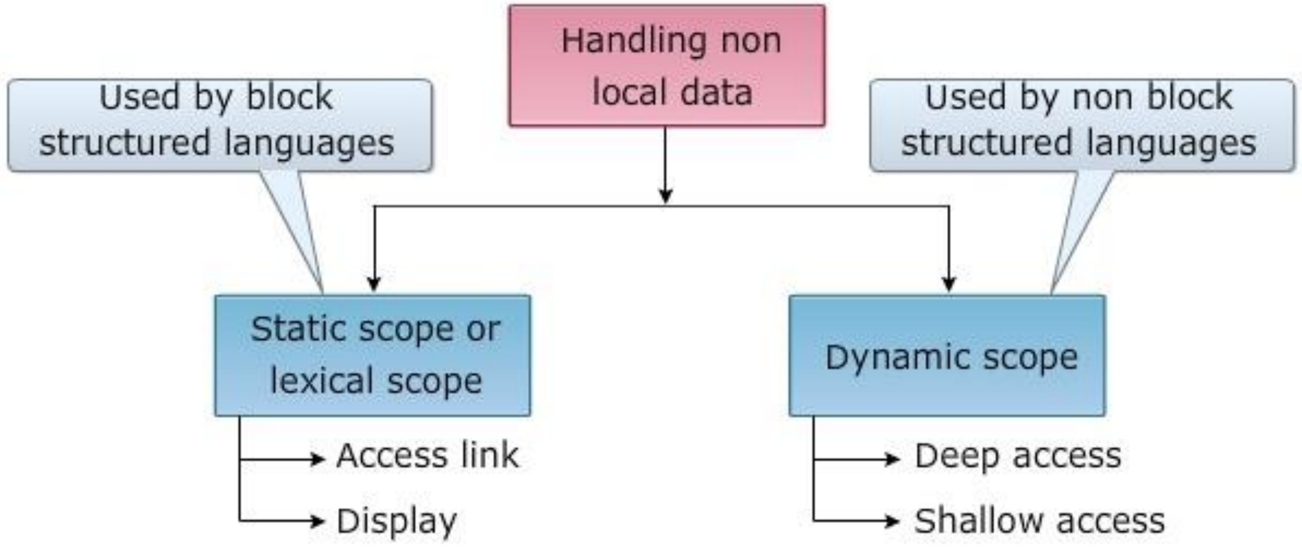
call by name

- The name of the procedure being called is replaced by its actual body
- Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so it can work on actual parameters, much like pass-by-reference

```
void procedure(int x, int y){   for(int k=0; k<10; k++) { y = 0; x++; }       }
main() { int j = 0, A[10];   procedure(j, A[j]);}
```

Access to non-local data on the stack:

- When a procedure refer to variables that are not local to it, then such variables are called nonlocal variables.
- Mechanism for finding the data used within procedure but doesn't belong to that procedure
- There are two types of scope rules for the non-local names. They are 1. Static 2.Dynamic


Handling non local data
Used by block structured languages
Used by non block structured languages
Static scope or lexical scope
Dynamic scope
Access link
Display
Deep access
Shallow access

- There are various situations or conditions to access the nonlocal data. These include:

Lexical Scope for Procedures

  - If a procedure is declared inside another procedure then it is called as nested procedure
  - A procedure pi, can call any procedure, i.e., its direct ancestor or older siblings of its direct ancestor

Access to non-local data on the stack:

1. Data Access without Nested Procedures

- ➢Generally, a variable has a scope of existence or use only within the function or procedure where it is defined.
- ➢After being declared globally, if the same variable is declared elsewhere, then this new definition overrides the global definition and is used as its value.
- ➢For non-nested procedures, variables are accessed as follow:

1. Global variables are declared statically as their values and locations are fixed at compile time.
2. Other variables (local) are declared locally at the top of the stack and can be accessed by top_sp pointer.

1.1. Issues with Nested Procedures

1. Access becomes more complicated when procedure declarations are nested and uses normal static scoping rule.
2. The nested declarations does not specify the relative positions of the Activation Records at runtime.

2. A Language with Nested Procedure Declarations

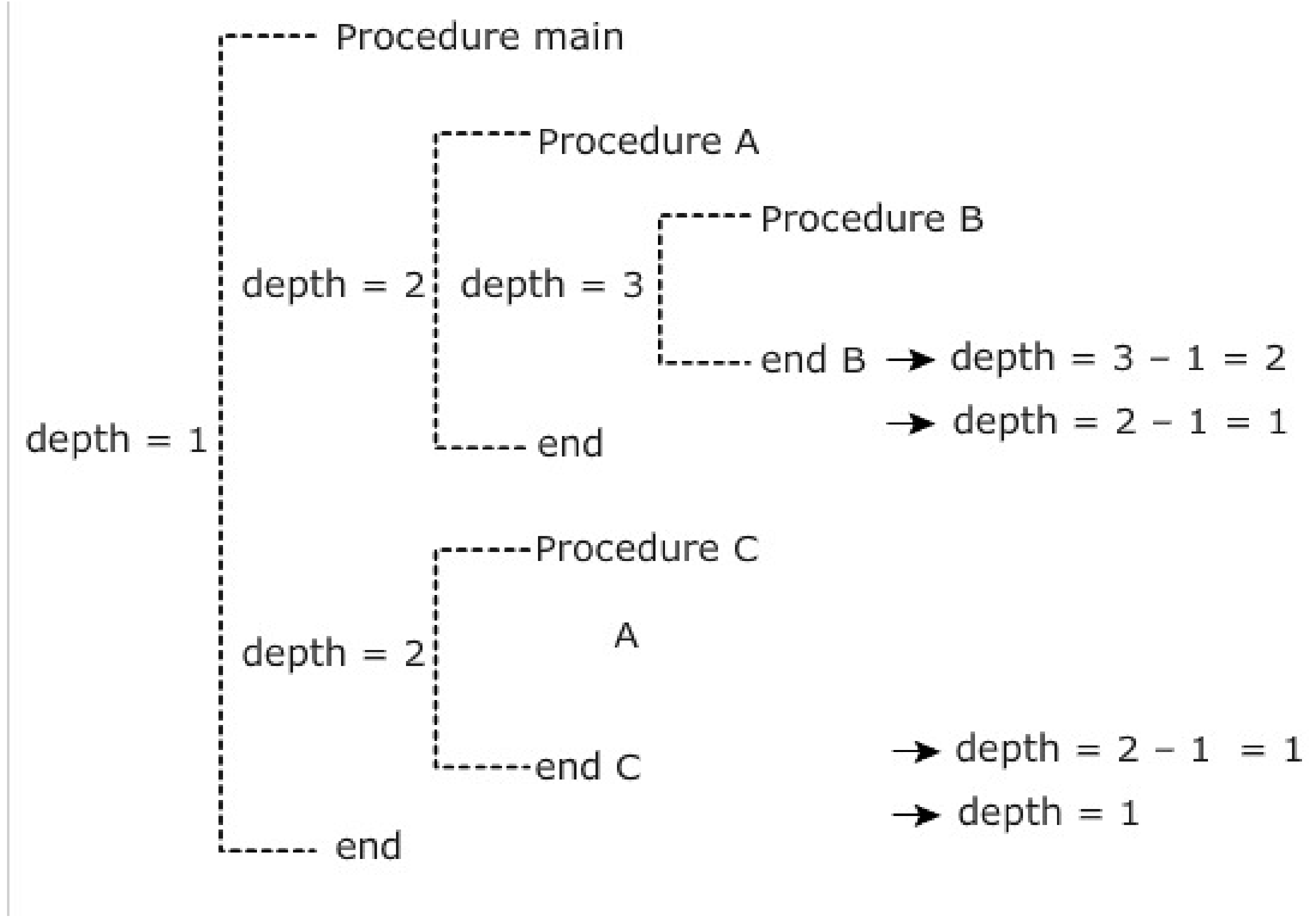
1. ML is a functioanal Language supports nested procedure declarations
2. val<name> = <expression
3. fun<name>(<arguments>) = <body>
4. let<list of definitions>in <statements>end

Access to non-local data on the stack (Cont...):

3. Nesting Depth

1. Lexical scope can be implemented by using nesting depth of a procedure
2. The main procedure nesting depth is 1
3. When a new procedure begins, add ‘1’ to nesting depth each time
4. When you exit from a nested procedure, subtract ‘1’ from depth each time

```
f1()             ---->    1
{
    f2();        ---->    2
    f3();        ---->    2
    f4()         ---->    2
    {
        f5();    ---->    3
    }
}
```



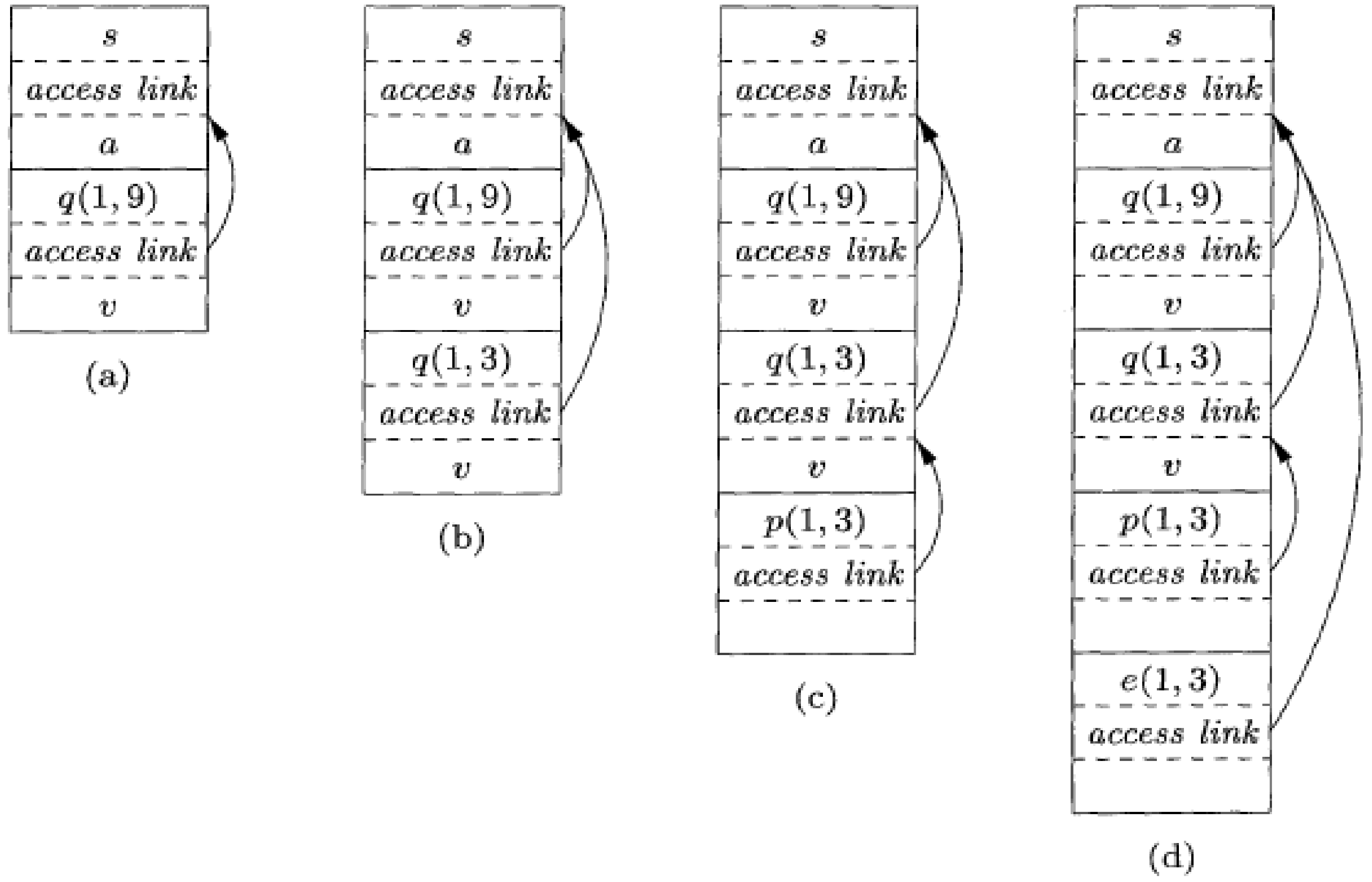
Access to non-local data on the stack (Cont...):

4. Access Link

1. A direct implementation of lexical scope for nested procedures is obtained by adding a pointer called an access link to each activation record.

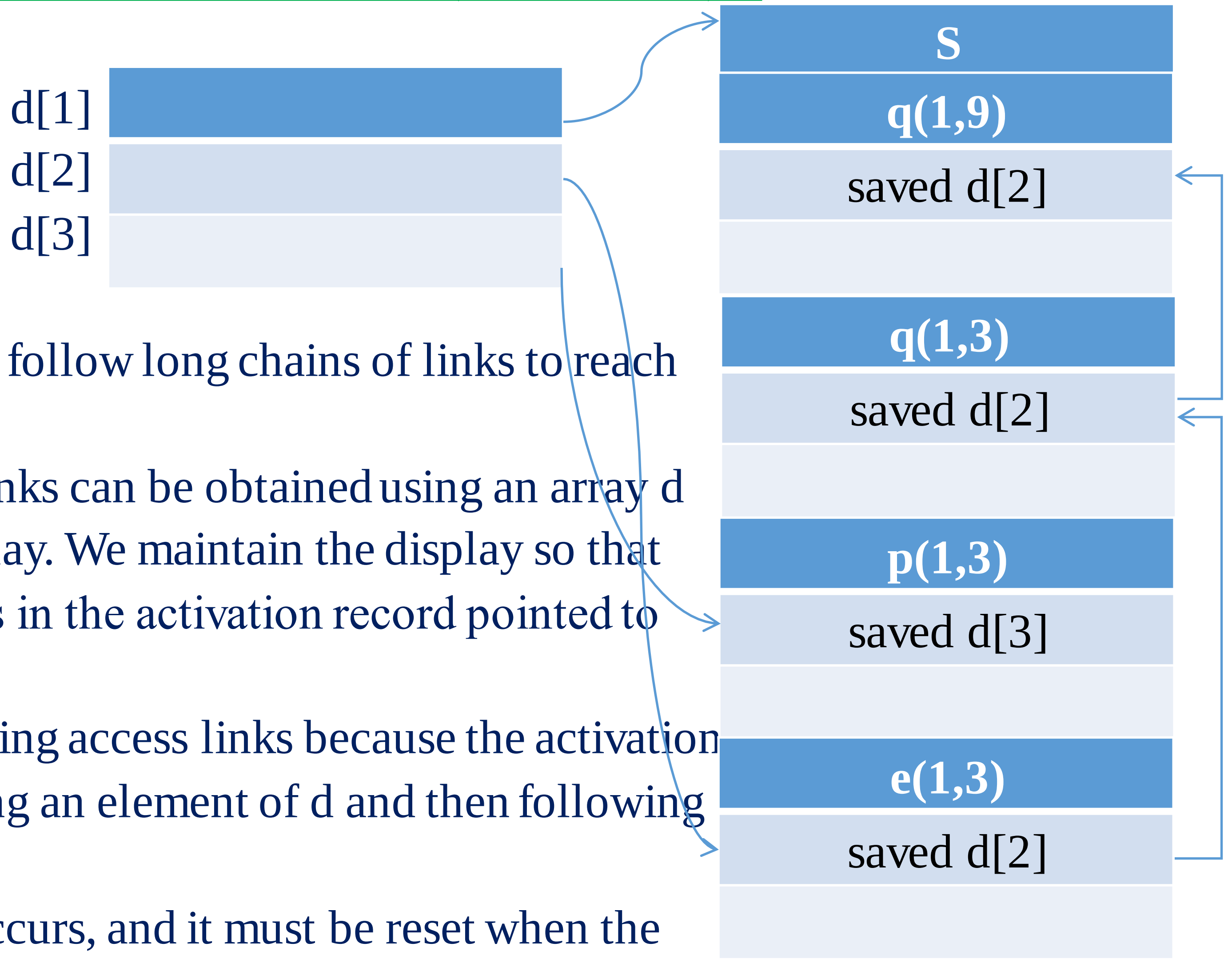
2. If procedure p is nested immediately within q in the source text, then the access link in an activation record for p points to the access link in the record for the most recent activation of q.

Access Links for finding nonlocal data →



Access to non-local data on the stack (Cont…):

5. Displays



1. If the nesting depth gets large then we have to follow long chains of links to reach the data
2. Faster access to non-locals than with access links can be obtained using an array d of pointers to activation records, called a display. We maintain the display so that storage for a nonlocal ‘a’ at nesting depth ‘i’ is in the activation record pointed to by display element d[i].
3. Using a display is generally faster than following access links because the activation record holding a nonlocal is found by accessing an element of d and then following just one pointer.
4. The display changes when a new activation occurs, and it must be reset when the control returns from the new activation.

Introduction to garbage collection:

- ✓ It is a technique used to automatically manage memory allocation and deallocation, in order to prevent memory leaks and improve program reliability.
- ✓ Garbage collection is used to automatically reclaim memory that is no longer in use by the program.
- ✓ A garbage collector works by keeping track of all objects that are currently in use by the program, and periodically checking to see if any of these objects are no longer accessible or needed.
- ✓ If an object is determined to be no longer needed, the memory it occupies is freed and can be reused by the program.
- ✓ There are several different algorithms for performing garbage collection, each with its own advantages and disadvantages. Some common algorithms include reference counting, mark-and-sweep, and copying collectors.

Reference Counting: It involves maintaining a count of the number of references to each object in memory, and freeing objects that have a reference count of zero. This approach is simple to implement, but can suffer from performance problems and is not effective for detecting and freeing cyclic data structures.

Mark-and-Sweep: It involves starting from a set of root objects and marking all objects that are reachable from these roots. Once all reachable objects have been marked, the memory occupied by objects that are not marked is freed. This approach is more effective at freeing cyclic data structures, but can cause temporary pauses in program execution while the collection is performed.

Copying Collectors: It divide memory into two areas, called the "from" and "to" spaces. During a collection, all live objects are copied from the "from" space to the "to" space, and the "from" space is then cleared and reused. This approach can be more efficient in terms of memory utilization, but requires more memory to be reserved for the "to" space.

Introduction to trace based collection:

- ✓ Instead of collecting garbage as it is created, trace-based collectors run periodically to find unreachable objects and reclaim their space. Typically, we run the trace-based collector whenever the free space is exhausted or its amount drops below some threshold.
- ✓ We can begin with "mark-and-sweep" garbage collection algorithm. Then we describe the variety of trace-based algorithms in terms of four states that chunks of memory can be put in. This also contains a number of improvements on the basic algorithm, including those in which object relocation is a part of the garbage-collection function.

A Basic Mark-and-Sweep Collector: Mark-and-sweep garbage-collection algorithms are straightforward, stop-the-world algorithms that find all the unreachable objects, and put them on the list of free space.

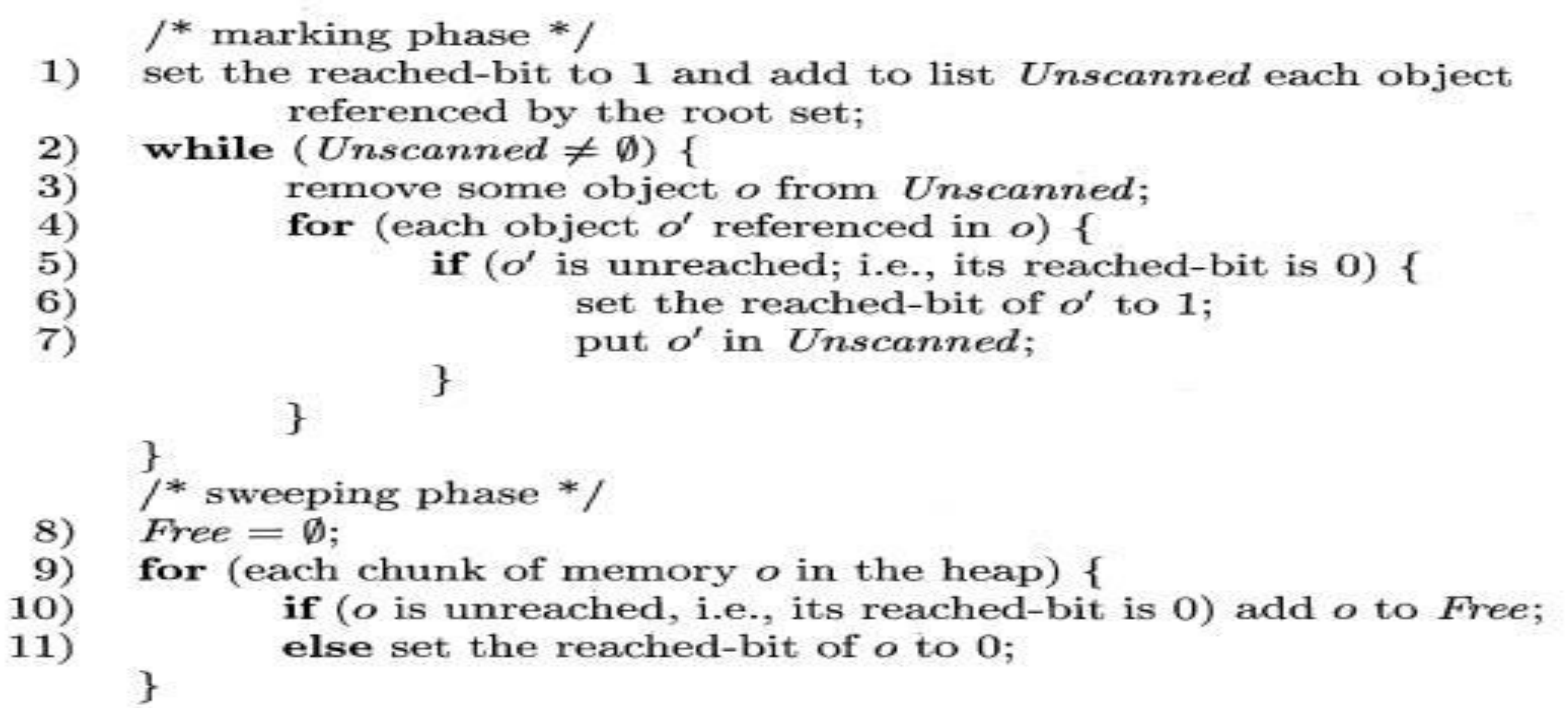
INPUT: A root set of objects, a heap, and a free list, called Free, with all the unallocated chunks of the heap. All chunks of space are marked with boundary tags to indicate their free/used status and size.

OUTPUT: A modified Free list after all the garbage has been removed.

METHOD: The algorithm, shown in Fig. 1, uses several simple data struc-tures. List Free holds objects known to be free. A list called Unscanned, holds objects that we have determined are reached, but whose successors we have not yet considered. That is, we have not scanned these objects to see what other objects can be reached through them. The Unscanned list is empty initially. Additionally, each object includes a bit to indicate whether it has been reached (the reached-bit). Before the algorithm begins, all allocated objects have the reached-bit set to 0.

Introduction to trace based collection (Cont...):

Mark-and-Sweep Collector Algorithm:



```
    /* marking phase */
1)  set the reached-bit to 1 and add to list Unscanned each object
        referenced by the root set;
2)  while (Unscanned ≠ ∅) {
3)      remove some object o from Unscanned;
4)      for (each object o′ referenced in o) {
5)          if (o′ is unreached; i.e., its reached-bit is 0) {
6)              set the reached-bit of o′ to 1;
7)              put o′ in Unscanned;
            }
        }
    }
    /* sweeping phase */
8)  Free = ∅;
9)  for (each chunk of memory o in the heap) {
10)     if (o is unreached, i.e., its reached-bit is 0) add o to Free;
11)     else set the reached-bit of o to 0;
    }
```

Figure 7.21: A Mark-and-Sweep Garbage Collector

Unit – 4(Part 2)

Machine Independent Code optimizations

7. The principal sources of optimization

8. Introduction to data-flow analysis

9. Foundations of data-flow analysis

10. Constant propagation

11. Partial redundancy elimination

12. Loop optimization in flow graphs

Principal Sources of Code Optimization:

- ✓It optimizes the code by removing unnecessary lines of code so we can use less amount of memory to store it and execute it in a fast manner
- ✓Various Code Optimization Techniques

1. Common sub expression elimination	2. Code Motion / Code movement
3. Compile Time Evaluation	→ a. Constant folding b. Constant propagation
4. Dead Code Elimination	5. Induction Variable & Strength Reduction

1. Common Subexpression Elimination

- ✓ It is an expression which appears repeatedly in the program, which is computed previously but the values of variables in expression doesn't changed
- ✓ It replaces the redundant expression each time it is encountered

unoptimized code	**optimized code**
a = b + c	a = b + c
b = a – d	b = a – d
c = b + c	c = b + c
d = a – d	d = b

Principal Sources of Optimization (Cont…):

2. Compile Time Evaluation

- ✓ Evaluation is done at compile time instead of run time & It can be done in 2 ways
  - a. **Constant folding**
    - ➢ Evaluate the expression and submit the result
    - ➢ area=(22/7)*r*r here 22/7 is calculated and result 3.14 is replaced so area = 3.14*r*r
  - b. **Constant propagation**
    - ➢ Constant replaces a variable
    - ➢ pi=3.14, r=5, area=pi*r*r → area=3.14*5*5

3. Code Motion / Code Movement

- ✓ It is a technique which moves the code outside the loop if it won't have any difference, if it executes inside or outside the loop

unoptimized code

```
for(i=0; i<n; i++) {
        x=y+z;
        a[i] = 6*i;     }
```

optimized code

```
    x = y + z;
    for(i=0; i<n; i++) {
            a[i] = 6*i;    }
```

Principal Sources of Optimization (Cont…):

4. Dead Code Elimination

✓ It eliminates the statements which are never executed or if executed its output is never used

unoptimized code

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

optimized code

```
i = 0;
```

unoptimized code

```
int add(int x, int y)
{
  int z; z=x+y;
  return z;
  printf("%d",z);
}
```

optimized code

```
int add(int x, int y)
{
    int z; z=x+y;
    return z;
}
```

5. Induction Variable & Strength Reduction

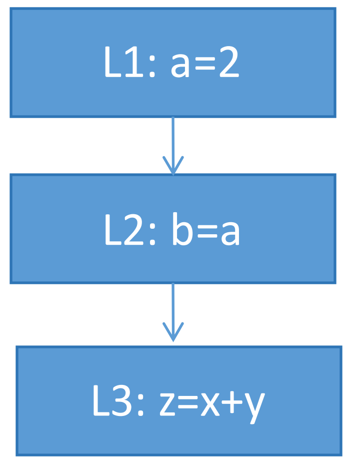
Replacement of expensive operator with cheaper operator (* >>>> +)

Ex: b = a*2; is replaced with b = a+a

Introduction to Data Flow Analysis:

- ➢ It is an analysis that determines the information regarding the definition and use of data in the program
- ➢ Global Optimization can be done with this

❖ **Basic Terminology:**



→ Definition Point - Define the variable a and assign 2 in block 1

→ Reference Point - Reference of a is assigned to the variable b

→ Evaluation Point - expression eveluation is assigned to the variable z

❖ **Advantages:**

- ❖ Code quality improvement
- ❖ Better identification of errors
- ❖ Understand behaviour

Introduction to Data Flow Analysis (Cont…):

❖ Data Flow Properties

- ❖ **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.

  Ex: **B1**: L1 = 4*i **B2**: y = x + L1 **B3**: p = L1 * 3

  here L1 i.e., 4*i is available for B2 and B3

- ❖ **Reaching Definition:** A definition D is reaching to a point 'x', if D is not killed or redefined before that point. It is used in constant / variable propogation.

  Ex: **B1**: D1: x = 4 **B2**: D2: x = x+2 **B3**: D3: y = x+2

  here D1 is reaching definition for B2 but not for B3 since it is killed by D2

- ❖ **Live Variable:** If a value or an expression is used for future computation.

  Ex: **B1**: a = 4 **B2**: c = 6 **B3**: b = 8 **B4**: d = a+b **B5**: a = c+d

  here a is live variable for B1, B3 and B4 but not for B5 since it is modified at B5

- ❖ **Busy Expression:** An expression is busy along the path, if its eveluation exists along that path and none of its operand definition exists before its eveluation along the path.

  Ex:

  here In the path itself we are evaluating and defining the expression

Foundations of Data-Flow Analysis:

- ✓ A data-flow analysis framework (D,V,A:F) consists of a direction of the data flow D, which is either F O R W A R D S or B A C K W A R D S .
- ✓ A semilattice includes a do-main of values V and a meet operator A.
- ✓ A family F of transfer functions from V to V. This family must include functions suitable for the boundary conditions, which are constant transfer functions for the special nodes E N T R Y and E X I T in any flow graph.

1. Semilattices

- ✓ A semilattice is a set V and a binary meet operator A such that for all x, y, and z in V:

1. x A x — x (meet is idempotent).
2. x A y = y A x (meet is commutative).
3. x A (y A z) = (x A y) A z (meet is associative).

A semilattice has a top element, denoted T, such that for all x in V, T A x — x.

A semilattice has a bottom element, denoted ±, such that for all x in V, _L A x = ±.

Foundations of Data-Flow Analysis (Cont...):

2. Transfer Functions

✓ A semilattice is a set V and a binary meet operator A such that for all x, y, and z in V:

1. x A x — x (meet is idempotent).
2. x A y = y A x (meet is commutative).
3. x A (y A z) = (x A y) A z (meet is associative).

A semilattice has a top element, denoted T, such that for all x in V, T A x — x.

A semilattice has a bottom element, denoted ±, such that for all x in V, _L A x = ±.

3. The Iterative Algorithm for General Frameworks

✓ We can generalize Algorithm to make it work for a large variety of data-flow problems.

Algorithm: Iterative solution to general data-flow frameworks.

INPUT: A data-flow framework with the following components:

1. A data-flow graph, with specially labeled E N T R Y and E X I T nodes
2. A direction of the data-flow D

Foundations of Data-Flow Analysis (Cont...):

3. A set of values V

4. A meet operator A

5. A set of functions F, where fs in F is the transfer function for block B and A constant value GENTRY or f EXIT in V, representing the boundary condition for forward and backward frameworks, respectively

OUTPUT: Values in V for IN[B] and OUT [B] for each block B in the data-flow graph.

4. Meaning of a Data-flow Solution

- ✓ We now know that the solution found using the iterative algorithm is the maximum fixedpoint
- ✓ The solution of a data-flow framework (D, F, V, A), let us first describe what an ideal solution to the framework would be.
- ✓ We show that the ideal cannot be obtained in general, but the above Algorithm approximates the ideal conservatively.

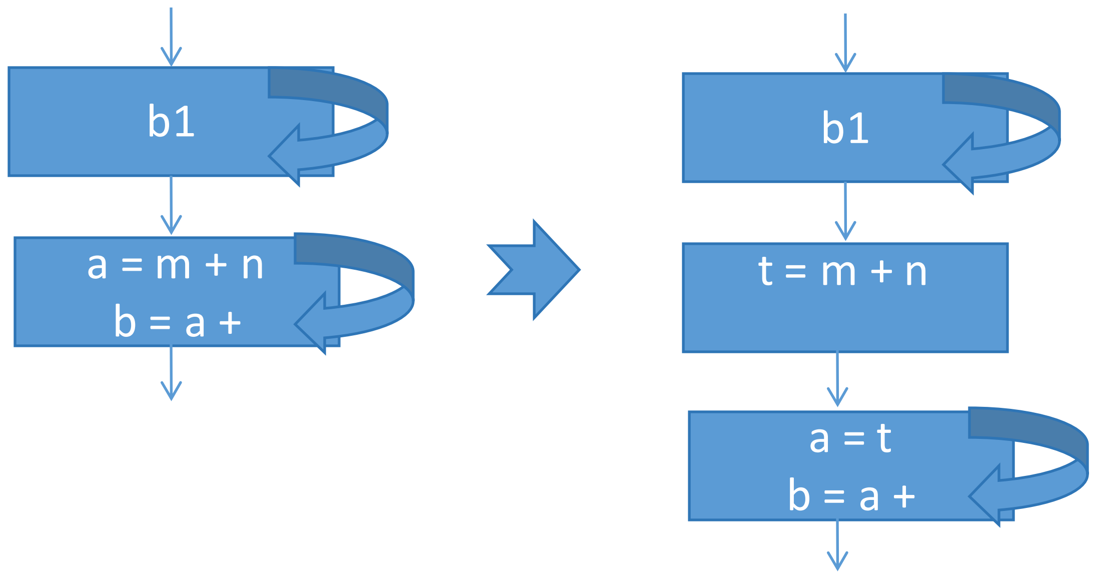
Constant Propagation:

Constant Propagation is one of the local code optimization technique

- ✓ It can be defined as the process of replacing the constant value of variables in the expression
- ✓ If some value is assigned a known constant, than we can simply replace the that value by constant. Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.
- ✓ Constant propagation is executed using reaching definition analysis results in compilers, which means that if reaching definition of all variables have same assignment which assigns a same constant to the variable, then the variable has a constant value and can be substituted with the constant.
- ✓ Suppose we are using pi variable and assign it value of 22/7 i.e., pi = 22/7 = 3.14
- ✓ In the above code the compiler has to first perform division operation, which is an expensive operation and then assign the computed result 3.14 to the variable pi.

Partial Redundancy Elimination:

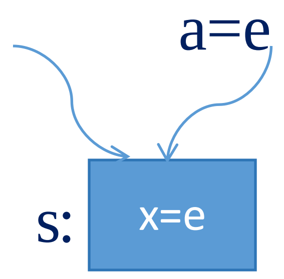
- ✓ It is also called as lazy code motion
- ✓ All redundant computations of expressions that can be eliminated without code duplication are eliminated
- ✓ The optimized program does not perform any computations that were not already present in the original program execution
- ✓ Expression are computed at the latest possible time



Partial Redundancy Elimination(Cont...):

Full Vs Partial Redundancy

- ✓ e is fully redundant if it is available on all incoming paths
- ✓ e is partially redundant if it is available on some(but not all) incoming paths
- ✓ In the given example, e is partially redundant at s



Loop Optimization:

✓ We can apply optimization techniques on loops also. These are

1. Code Motion / Code Movement
2. Loop Invariant Computations
3. Loop Unrolling
4. Loop Fusion

1. Code Motion / Code Movement

✓ It is a technique which moves the code outside the loop if it won't have any difference, if it executes inside or outside the loop

unoptimized code

```
for(i=0; i<n; i++) {
        x=y+z;
        a[i] = 6*i;     }
```

optimized code

```
x = y + z;
for(i=0; i<n; i++) {
        a[i] = 6*i;    }
```

2. Loop Invariant Computations

➢ The statements in the loop whose result of the computations do not change over the iterations

Ex: a=10; b=20; c=30;

for(i=0;i<10;i++) { **c=a+b; d=a-b; e=a*b;** s=s+i; }

Here 1st 3 statements in for loop is loop invariant computations i.e., these 3 statements are not dependent on for loop so we can eliminate these statements

Loop Optimization (Cont…):

3. Loop Unrolling

- ➢ Loop overhead can be reduced by reducing the no. of iterations & replacing body of the loop

Ex:
```
for(i=0;i<100;i++) {   add();   }
```

reduced to

```
for(i=0;i<50;i++) {   add();    add();   }
```

4. Loop Fusion

- ➢ Adjacent loops can be merged into one loop to reduce loop overhead & improve performance

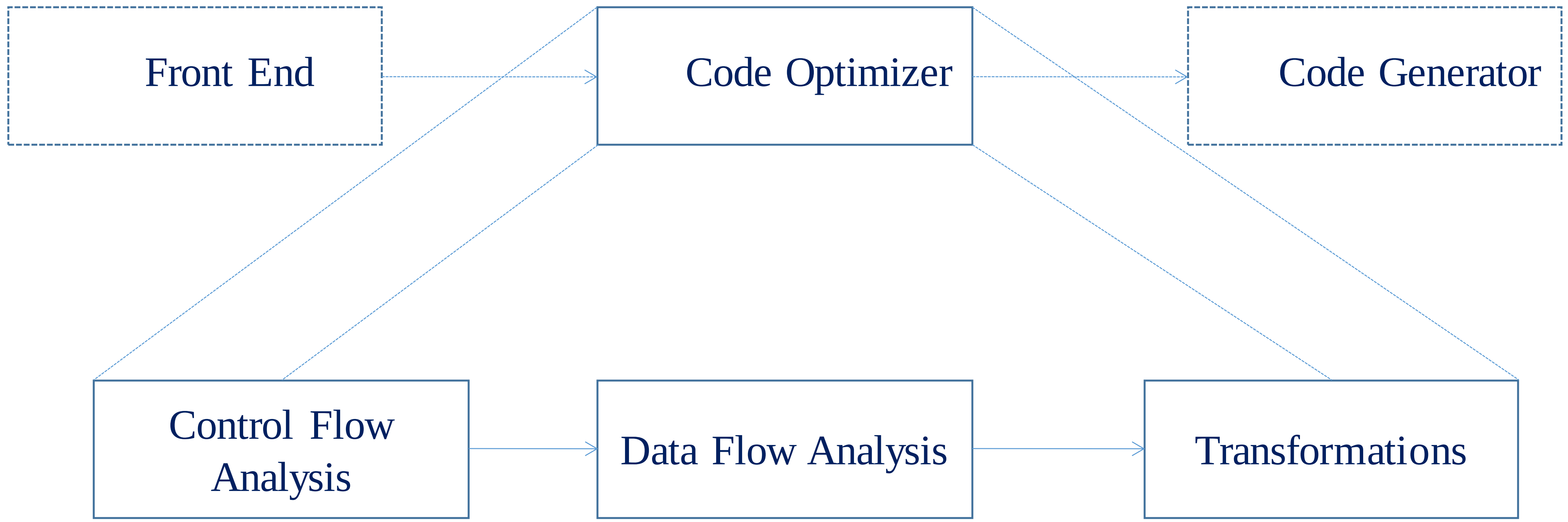
Ex:
```
for(i=0;i<10;i++) {   a[i]=a[i]+10;   }
for(i=0;i<10;i++) {   b[i]=b[i]+10;    }
```

reduced to

```
for(i=0;i<10;i++) {   a[i]=a[i]+10;    b[i]=b[i]+10;   }
```

Code Optimizer:

1.Position of code optimizer



2.Purpose of code optimizer

- ✓ To get better efficiency
  - ➢ Run faster
  - ➢ Take less

Code Optimizer (Cont…):

3. Places for potential improvements by the user and the compiler

✓**Source code**

- ➢User can profile program, change algorithm or transform loops.

✓**Intermediate code**

- ➢Compiler can improve loops, procedure calls or address calculations

✓**Target code**

- ➢Compiler can use registers, select instructions or do peephole transformations

code optimization Techniques

1. Compile Time Evaluation
2. Variable Propagation
3. Dead Code Elimination
4. Code Motion
5. Induction Variable & Strength Reduction

Issues in the design of code optimization:

- ✓Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed
- ✓In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.
- ✓A code optimizing process must follow the three rules given below:
    - ➢The output code must not, in any way, change the meaning of the program
    - ➢Optimization should increase the speed of the program and if possible, the program should demand less number of resources
    - ➢Optimization should itself be fast and should not delay the overall compiling process