

Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code** : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables** : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

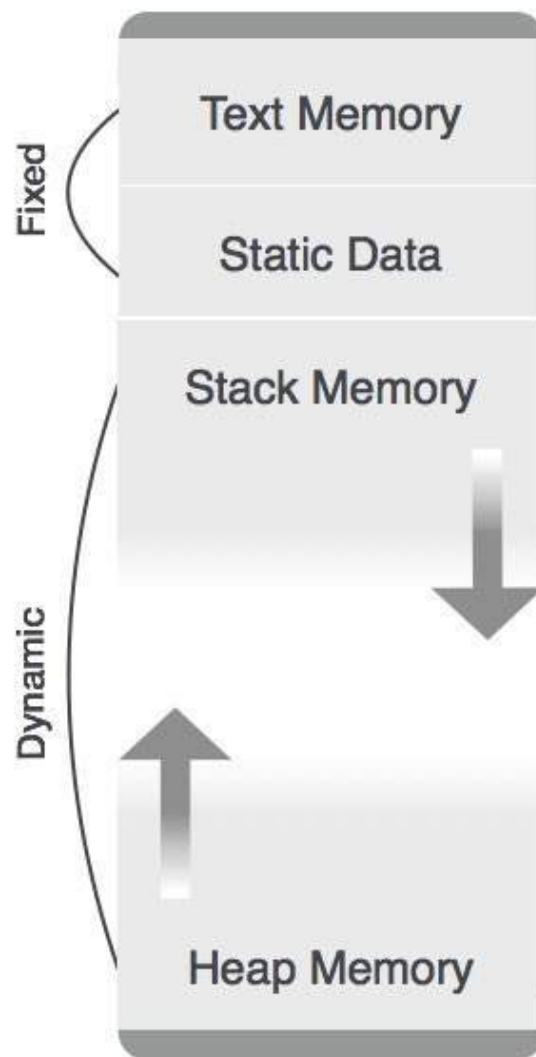
Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



As shown in the image above, the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Parameter Passing

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. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

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

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.

For example:

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

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

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

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

Formal Parameters

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

Actual Parameters

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
    int actual_parameter = 10;
    call fun_two(int actual_parameter);
}

fun_two(int formal_parameter)
{
    print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value

In pass by value mechanism, 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.

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, 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.

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon 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.

Example:

```
int y;
calling_procedure()
{
    y = 10;
    copy_restore(y); //l-value of y is passed
    printf y; //prints 99
}
```

```
}  
copy_restore(int x)  
{  
    x = 99; // y still has value 10 (unaffected)  
    y = 0; // y is now 0  
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, 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 that it can now work on actual parameters, much like pass-by-reference.

Compiler Design - Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

```
<symbol name, type, attribute>
```

For example, if a symbol table has to store information about the following variable declaration:

```
static int interest;
```

then it should store the entry such as:

```
<interest, int, static>
```

The attribute clause contains the entries related to the name.

Implementation

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations

A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

```
int a;
```

should be processed by the compiler as:

```
insert(a, int);
```

lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

```
lookup(symbol)
```

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;

void pro_one()
{
    int one_1;
    int one_2;

    {
        \
        int one_3;    |_ inner scope 1
```

```

    int one_4;      |
    }              /

void pro_one()
{
    int one_1;
    int one_2;

    {
        \
        int one_3;    |_ inner scope 1
        int one_4;    |
        int one_5;    |
    }                /

    {
        \
        int one_6;    |_ inner scope 2
        int one_7;    |
    }                /

void pro_two()
{
    int two_1;
    int two_2;

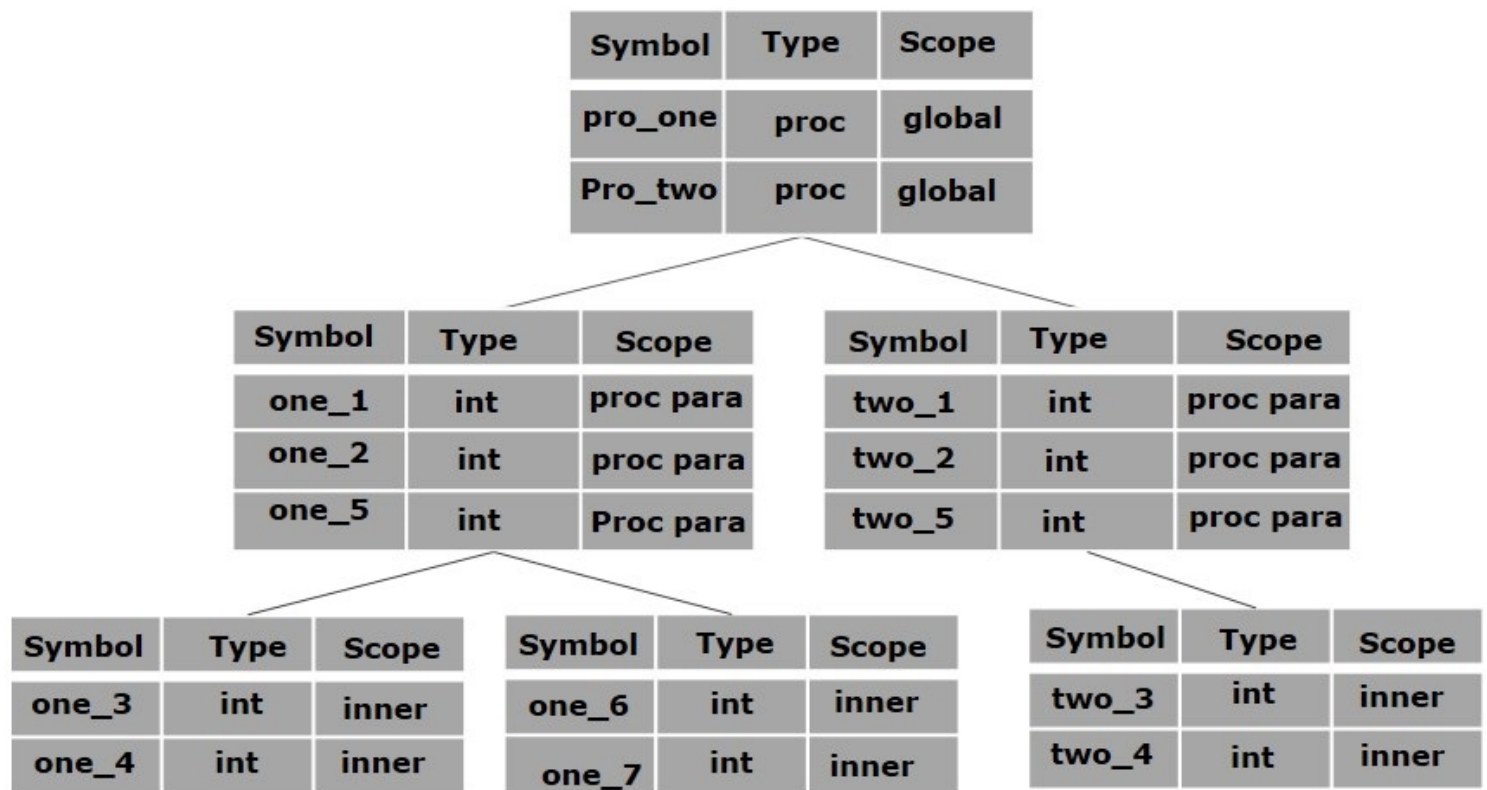
    {
        \
        int two_3;    |_ inner scope 3
        int two_4;    |
    }                /

    int two_5;
}

...

```

The above program can be represented in a hierarchical structure of symbol tables:



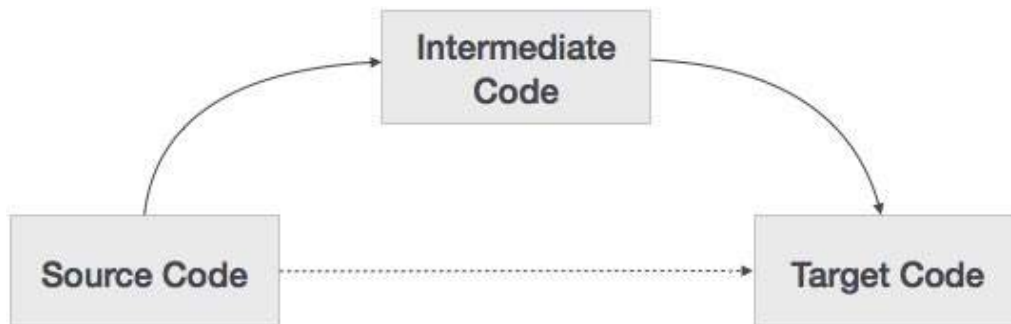
The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

Compiler - Intermediate Code Generation

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg ₁	arg ₂	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg ₁	arg ₂
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;
float b;
Allocation process:
{offset = 0}
int a;
id.type = int
id.width = 2
offset = offset + id.width
{offset = 2}
float b;
    id.type = float
    id.width = 4
    offset = offset + id.width
{offset = 6}
```

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.

Compiler Design - Code Generation

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

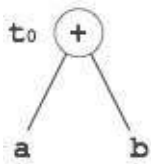
Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

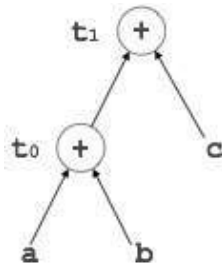
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

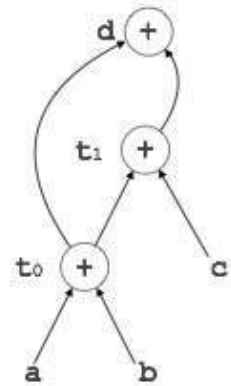
```
t0 = a + b
t1 = t0 + c
d = t0 + t1
```



$[t_0 = a + b]$



$[t_1 = t_0 + c]$



$[d = t_0 + t_1]$

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
{
    int y, z;
    y = 10;
    z = x + y;
    return z;
}
```

```
int add_ten(int x)
{
    int y;
    y = 10;
    y = x + y;
    return y;
}
```

```
int add_ten(int x)
{
    int y = 10;
    return x + y;
}
```

```
int add_ten(int x)
{
    return x + 10;
}
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidentally written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
    return x + 10;
    printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...  
MOV R1, R2  
GOTO L1  
  
...  
L1 :   GOTO L2  
L2 :   INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...  
MOV R1, R2  
GOTO L2  
  
...  
L2 :   INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression $a = a + 0$ can be replaced by a itself and the expression $a = a + 1$ can simply be replaced by $INC\ a$.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, $x * 2$ can be replaced by $x \ll 1$, which involves only one left shift. Though the output of $a * a$ and a^2 is same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

Code Generator

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation** : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code

generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

- **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor** : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
- **Address descriptor** : Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor (x) to show that one instance of x is in R1.

Code Generation

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

Note : If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg : Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction $x = y \text{ OP } z$, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of $y \text{ OP } z$ is to be saved:

- Call function *getReg*, to decide the location of L.
- Determine the present location (register or memory) of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**:

MOV **y'**, L

where **y'** represents the copied value of **y**.

- Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

OP **z'**, L

where **z'** represents the copied value of **z**.

- Now L contains the value of $y \text{ OP } z$, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.

- If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.

Compiler Design - 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.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
    item = 10;
    value = value + item;
}while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
    value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

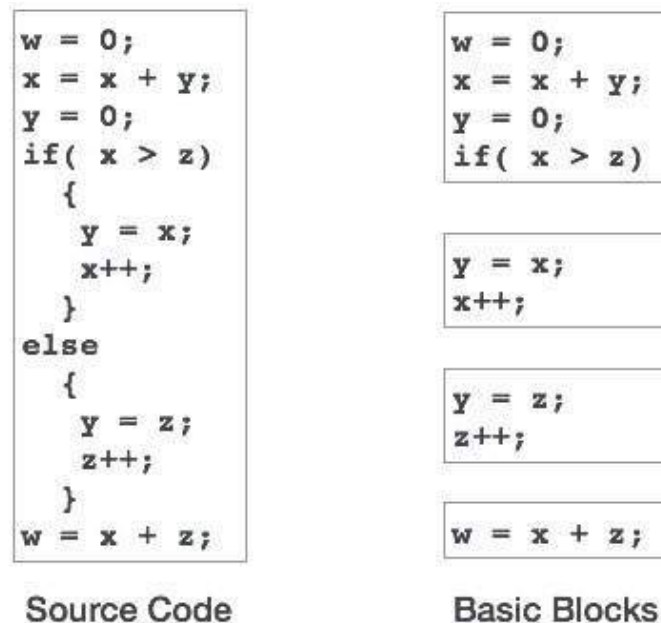
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
 - First statement of a program.
 - Statements that are target of any branch (conditional/unconditional).
 - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.

B1

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

B2

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

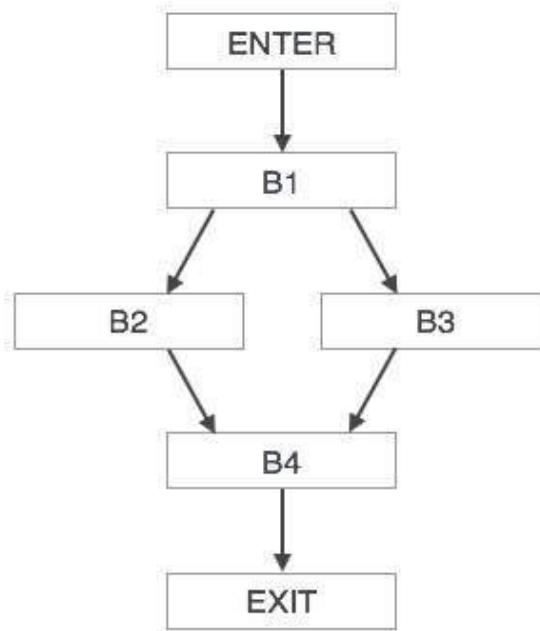
B3

```
y = z;
z++;
```

B4

```
w = x + z;
```

Basic Blocks



Flow Graph

Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication ($x * 2$) is expensive in terms of CPU cycles than ($x << 1$) and yields the same result.

Dead-code Elimination

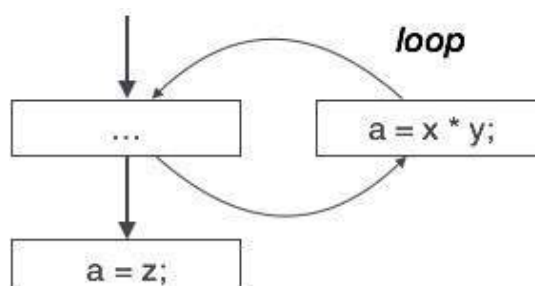
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

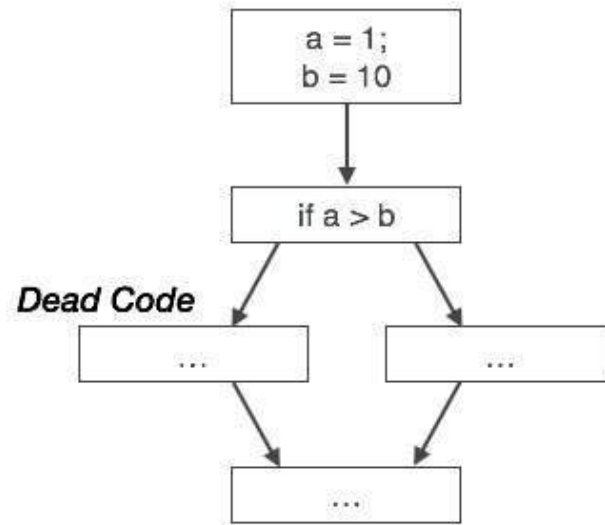
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



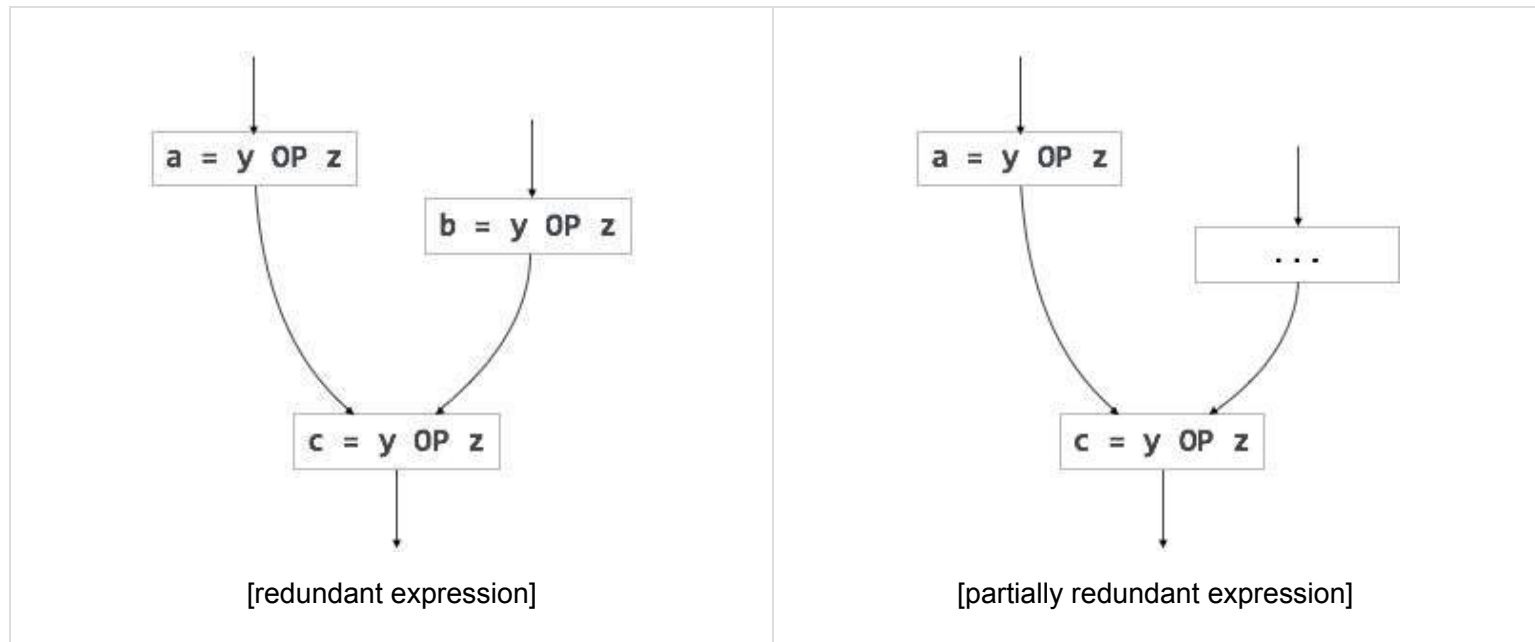
The above control flow graph depicts a chunk of program where variable 'a' is used to assign the output of expression 'x * y'. Let us assume that the value assigned to 'a' is never used inside the loop. Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.



Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands. whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
    a = y OP z;
}
else
{
    ...
}
c = y OP z;
```


We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then **y OP z** is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
    ...
    tmp = y OP z;
    a = tmp;
    ...
}
else
{
    ...
    tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; **y OP z** should be computed only once.