**Experiment No:-10**

**Aim**: To study & implement Code Generation Algorithm.

**Theory:**

In [computing](https://en.wikipedia.org/wiki/Computing), code generation is the process by which a [compiler](https://en.wikipedia.org/wiki/Compiler)'s code generator converts some [intermediate representation](https://en.wikipedia.org/wiki/Intermediate_representation) of [source code](https://en.wikipedia.org/wiki/Source_code) into a form (e.g., [machine code](https://en.wikipedia.org/wiki/Machine_code)) that can be readily executed by a machine. Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many [algorithms](https://en.wikipedia.org/wiki/Algorithm) for [code optimization](https://en.wikipedia.org/wiki/Code_optimization) are easier to apply one at a time, or because the input to one optimization relies on the completed processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the *backend*) needs to change from target to target. (For more information on compiler design, see [Compiler](https://en.wikipedia.org/wiki/Compiler).) The input to the code generator typically consists of a [parse tree](https://en.wikipedia.org/wiki/Parse_tree) or an [abstract syntax tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree). The tree is converted into a linear sequence of instructions, usually in an [intermediate language](https://en.wikipedia.org/wiki/Intermediate_language) such as [three-address code](https://en.wikipedia.org/wiki/Three-address_code). Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program. (For example, a [peephole optimization](https://en.wikipedia.org/wiki/Peephole_optimization) pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.)

**Major task in code generation:**

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way.

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

* Instruction selection: which instructions to use.
* Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.
* [Register allocation](https://en.wikipedia.org/wiki/Register_allocation): the allocation of variables to processor registers
* [Debug data](https://en.wikipedia.org/wiki/Debugging_data_format) generation if required so the code can be [debugged](https://en.wikipedia.org/wiki/Debugging).

Instruction selection is typically carried out by doing a [recursive](https://en.wikipedia.org/wiki/Recursion) [postorder traversal](https://en.wikipedia.org/wiki/Postorder_traversal) on the abstract syntax tree, matching particular tree configurations against templates; for example, the tree W := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.

In a compiler that uses an intermediate language, there may be two instruction selection stages — one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the [instruction set](https://en.wikipedia.org/wiki/Instruction_set) of the target machine. This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding [opcodes](https://en.wikipedia.org/wiki/Opcode). However, if the compiler is actually a [language translator](https://en.wikipedia.org/wiki/Transcompiler) (for example, one that converts [Eiffel](https://en.wikipedia.org/wiki/Eiffel_(programming_language)) to [C](https://en.wikipedia.org/wiki/C_(programming_language))), then the second code-generation phase may involve *building* a tree from the linear intermediate code.

**Runtime code generation:**

When code generation occurs at [runtime](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)), as in [just-in-time compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation) (JIT), it is important that the entire process be [efficient](https://en.wikipedia.org/wiki/Algorithmic_efficiency) with respect to space and time. For example, when [regular expressions](https://en.wikipedia.org/wiki/Regular_expression) are interpreted and used to generate code at runtime, a non-deterministic [finite state machine](https://en.wikipedia.org/wiki/Finite_state_machine) is often generated instead of a deterministic one, because usually the former can be created more quickly and occupies less memory space than the latter. Despite its generally generating less efficient code, JIT code generation can take advantage of [profiling](https://en.wikipedia.org/wiki/Profiling_(computer_programming)) information that is available only at runtime. The fundamental task of taking input in one language and producing output in a non-trivially different language can be understood in terms of the core [transformational](https://en.wikipedia.org/wiki/Transformational_grammar) operations of [formal language theory](https://en.wikipedia.org/wiki/Formal_language_theory). Consequently, some techniques that were originally developed for use in compilers have come to be employed in other ways as well. For example, [YACC](https://en.wikipedia.org/wiki/YACC) (Yet Another [Compiler Compiler](https://en.wikipedia.org/wiki/Compiler_compiler)) takes input in [Backus-Naur form](https://en.wikipedia.org/wiki/Backus-Naur_form) and converts it to a parser in [C](https://en.wikipedia.org/wiki/C_(programming_language)). Though it was originally created for automatic generation of a parser for a compiler, yacc is also often used to automate writing code that needs to be modified each time specifications are changed. (For example, see.[[2]](https://en.wikipedia.org/wiki/Code_generation_(compiler)#cite_note-2))

Many [integrated development environments](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDEs) support some form of automatic [source code generation](https://en.wikipedia.org/wiki/Source_code_generation), often using algorithms in common with compiler code generators, although commonly less complicated. (See also: [Program transformation](https://en.wikipedia.org/wiki/Program_transformation), [Data transformation](https://en.wikipedia.org/wiki/Data_transformation).)

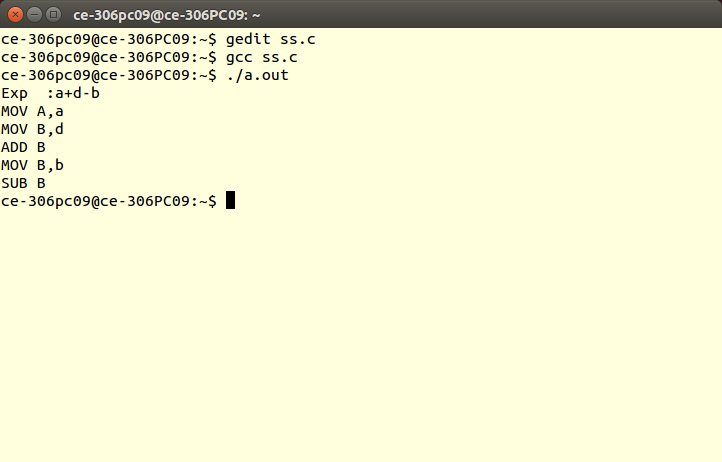
Reflection

In general, a syntax and semantic analyzer tries to retrieve the structure of the program from the source code, while a code generator uses this structural information (e.g., [data types](https://en.wikipedia.org/wiki/Data_type)) to produce code. In other words, the former *adds* information while the latter *loses some* of the information. One consequence of this information loss is that [reflection](https://en.wikipedia.org/wiki/Reflection_(computer_science)) becomes difficult or even impossible. To counter this problem, code generators often embed syntactic an

***Program: -***

#include<stdio.h>  
#include<ctype.h>  
#include<string.h>  
main()  
{  
char a[20];  
int i=2,n;  
printf("Exp  :");  
scanf("%s", a);  
if(isdigit(a[0]))  
printf("MVI A,%c\n",a[0]);  
else printf("MOV A,%c\n",a[0]);  
n=strlen(a);  
while(i<n)  
{  
switch(a[i])  
{  
case '+':printf("ADD B\n");i+=3;break;  
case '-':printf("SUB B\n");i+=3;break;  
default:if(isdigit(a[i]))  
printf("MVI B,%c\n",a[i]);  
else  
printf("MOV B,%c\n",a[i]);  
i--;  
}}}

***Output: -***



**Conclusion**: Hence program for code generation is studied and implemented successfully.