SYSTEM SOFTWARE

Module - V

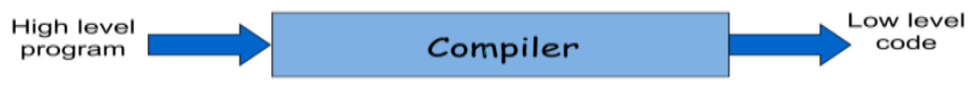
Compilers: Basic compiler functions, machine-dependent compiler features, machine independent compiler features, compiler design options the YACC compiler.

Basic Compiler Functions compiler

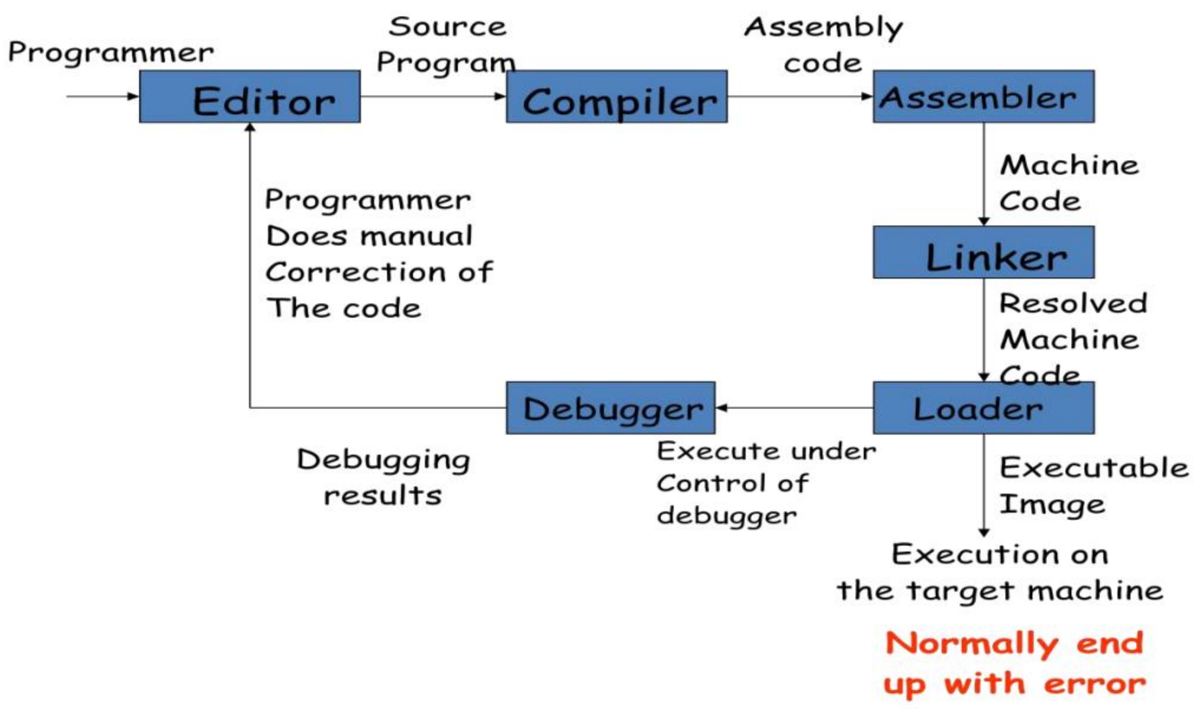
• Translators from one representation of the program to another
• Typically from high level source code to low level machine code or object code
• Source code is normally optimized for human readability
  – Expressive: matches our notion of languages (and application?!)
  – Redundant to help avoid programming errors
• Machine code is optimized for hardware
  – Redundancy is reduced
  – Information about the intent is lost
  This code should execute faster and use less resources.

How to translate
  • Source code and machine code mismatch in level of abstraction

We have to take some steps to go from the source code to machine code.
  • Some languages are farther from machine code than others
  • Goals of translation
    – High level of abstraction
    – Good performance for the generated code
    – Good compile time performance
    – Maintainable code



• Compiler is part of program development environment
The other typical components of this environment are editor, assembler, linker, loader, debugger, profiler etc.
Profiler will tell the parts of the program which are slow or which have never been executed.
• The compiler (and all other tools) must support each other for easy program Development



Machine Dependent Compiler

When control passes from one block to another, all values currently held in registers are saved in temporary variables. For example 3, the quadruples can be divided into five blocks.

They are

Block -- A Quadruples 1 - 3
Block -- B Quadruples 4
Block -- C Quadruples 5 - 14
Block --D Quadruples 15 - 20
Block -- E Quadruples 21 - 23

Fig. shows the basic blocks of the flow group for the quadruples. An arrow from one block to another indicates that control can pass directly from one quadruple to another.

This kind of representation is called a flow group.

Rearranging quadruples before machine code generation

Example

1) DIV SUMSQ 100 i1
2) MEAN MEAN i2
3) - i1 i2 i3
4) i3 VARIANCE

LDA SUMSQ DIV 100 STA i1 LDA MEAN
MUL MEAN STA i2 LDA i1 SUB i2 STA i3 STA

Variance shows a typical generation of machine code from the quadruples using only a single register ie Accumulator

Machine-Independent Compiler Features

There are several different possibilities for performing machine-dependent code optimization . Assignment and use of registers, Registers is used as instruction operand. The number of registers available is limited.

- Required to find the least used register to replace with new values when needed.
- Usually the existence of jump instructions creates difficulty in keeping track of registers contents.
- Divide the problem into basic blocks to tackle such problems.
- A basic block is a sequence of quadruples with one entry point, which is at the beginning of the block, one exit point, which is at the end of the block, and no jumps within the blocks.

CALL operation is usually considered to begin a new basic block.

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.

Methods for handling structured variables such as array.

Code optimization.

Storage allocation for the compiled program.

Compiling a block-structured language.

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);
```

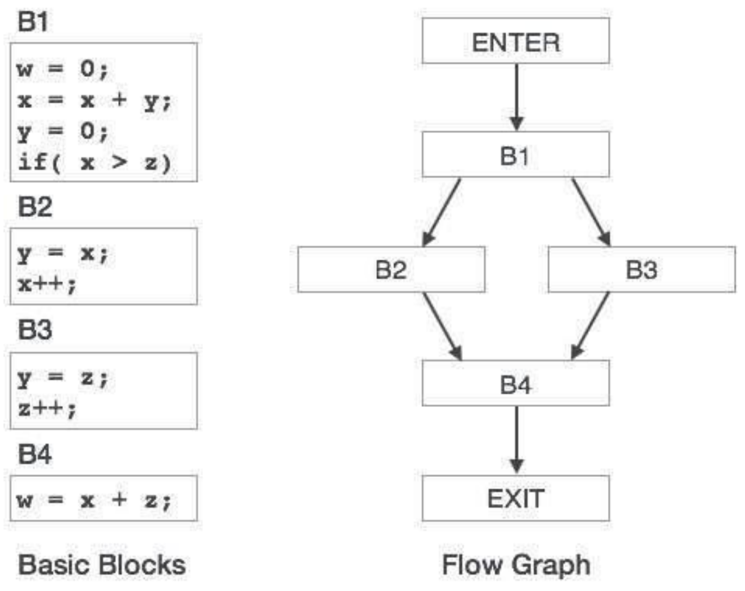
This code involves repeated assignment of the identifier item, which if we put this way:

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

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

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.



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

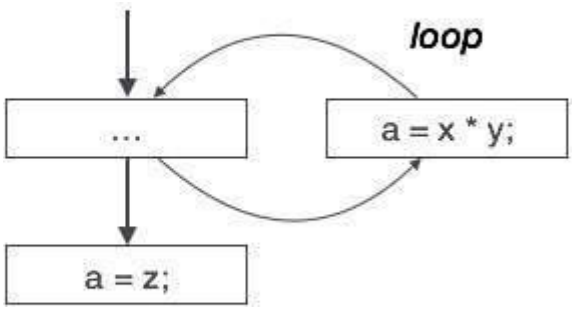
Dead code is one or more than one code statements, which are:

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

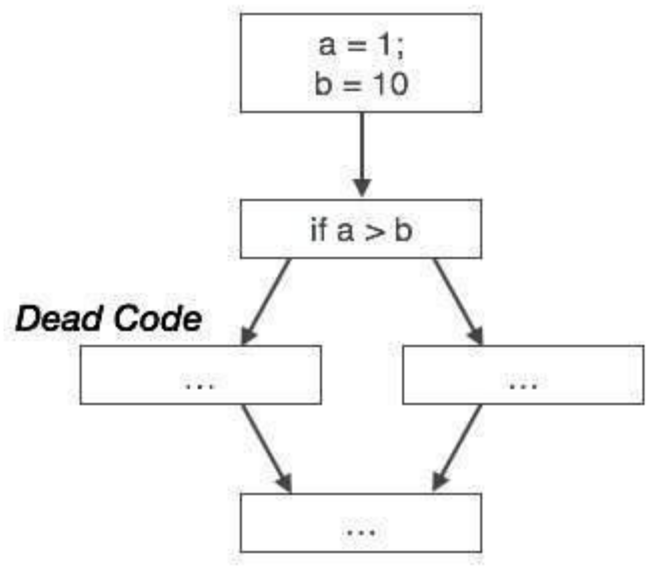
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

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.

YACC

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

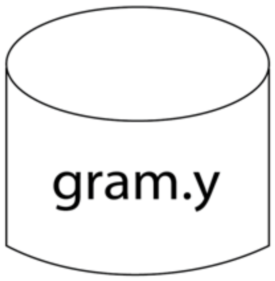
These are some points about YACC:

Input: A CFG- file.y

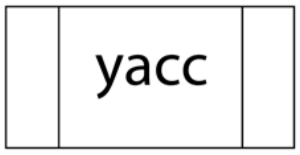
Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

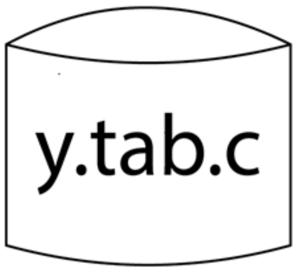
The basic operational sequence is as follows:



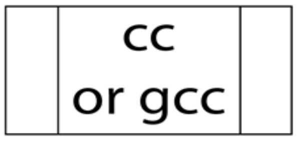
This file contains the desired grammar in YACC format.



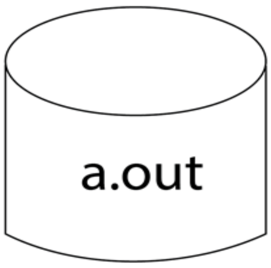
It shows the YACC program.



It is the c source program created by YACC.



C Compiler



Executable file that will parse grammar given in gram.Y