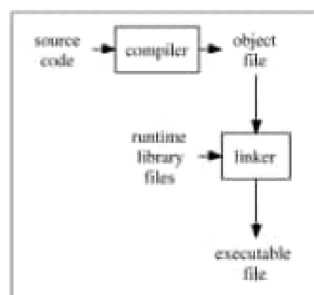
of op codes. Therefore, programming languages were invented to make it easier for humans to write computer programs.

Programming languages are for humans to read and understand. The program (source code) must be translated into machine language so that the computer can execute the program (as the computer only understands machine language). The way that this translation occurs depends on whether the programming language is a compiled language or an interpreted language.

Compiled languages (e.g. C, C++)

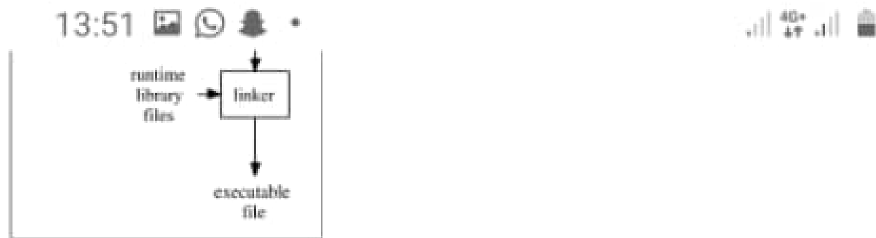
The following illustrates the programming process for a compiled programming language.



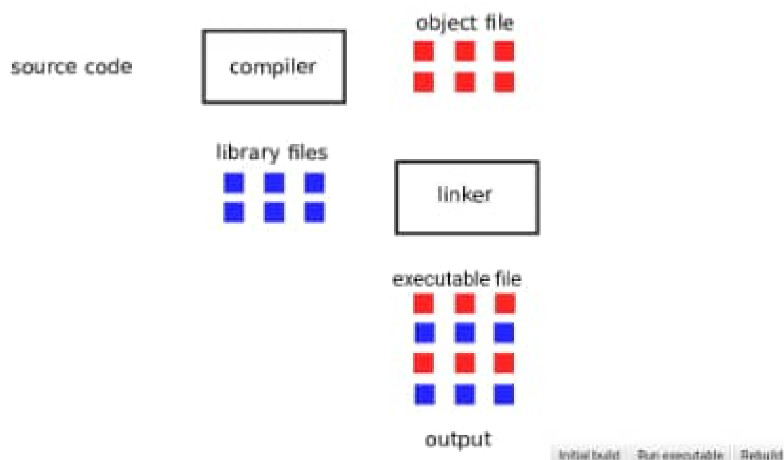
A compiler takes the program code (source code) and converts the source code to a machine language module (called an object file). Another specialized program, called a linker, combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file. This process is diagrammed below. Click **Initial build** to see an

Show simplified view

13:51



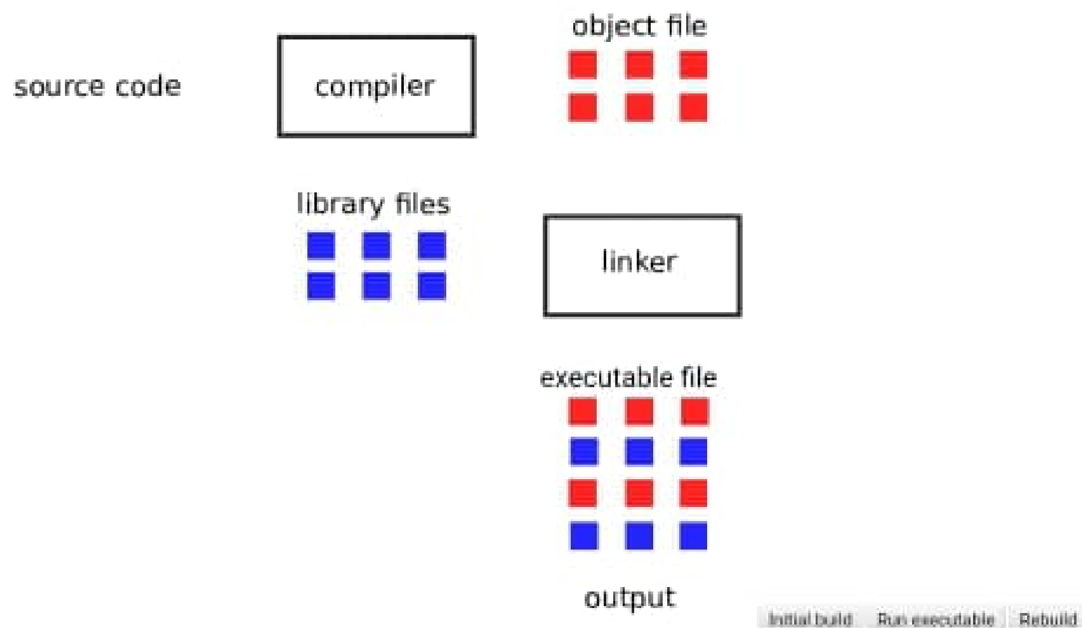
A compiler takes the program code (source code) and converts the source code to a machine language module (called an object file). Another specialized program, called a linker, combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file. This process is diagrammed below. Click **Initial build** to see an animation of how the executable is created. Click **Run executable** to simulate the running of an already created executable file. Click **Rebuild** to simulate rebuilding of the executable file.



So, for a compiled language the conversion from source code to machine executable code takes place before the program is run. This is a very different process from what takes place for an interpreted programming language.

This is somewhat simplified as many modern programs that are created using compiled languages makes use of dynamic linked libraries

diagrammed below. Click **Initial build** to see an animation of how the executable is created. Click **Run executable** to simulate the running of an already created executable file. Click **Rebuild** to simulate rebuilding of the executable file.

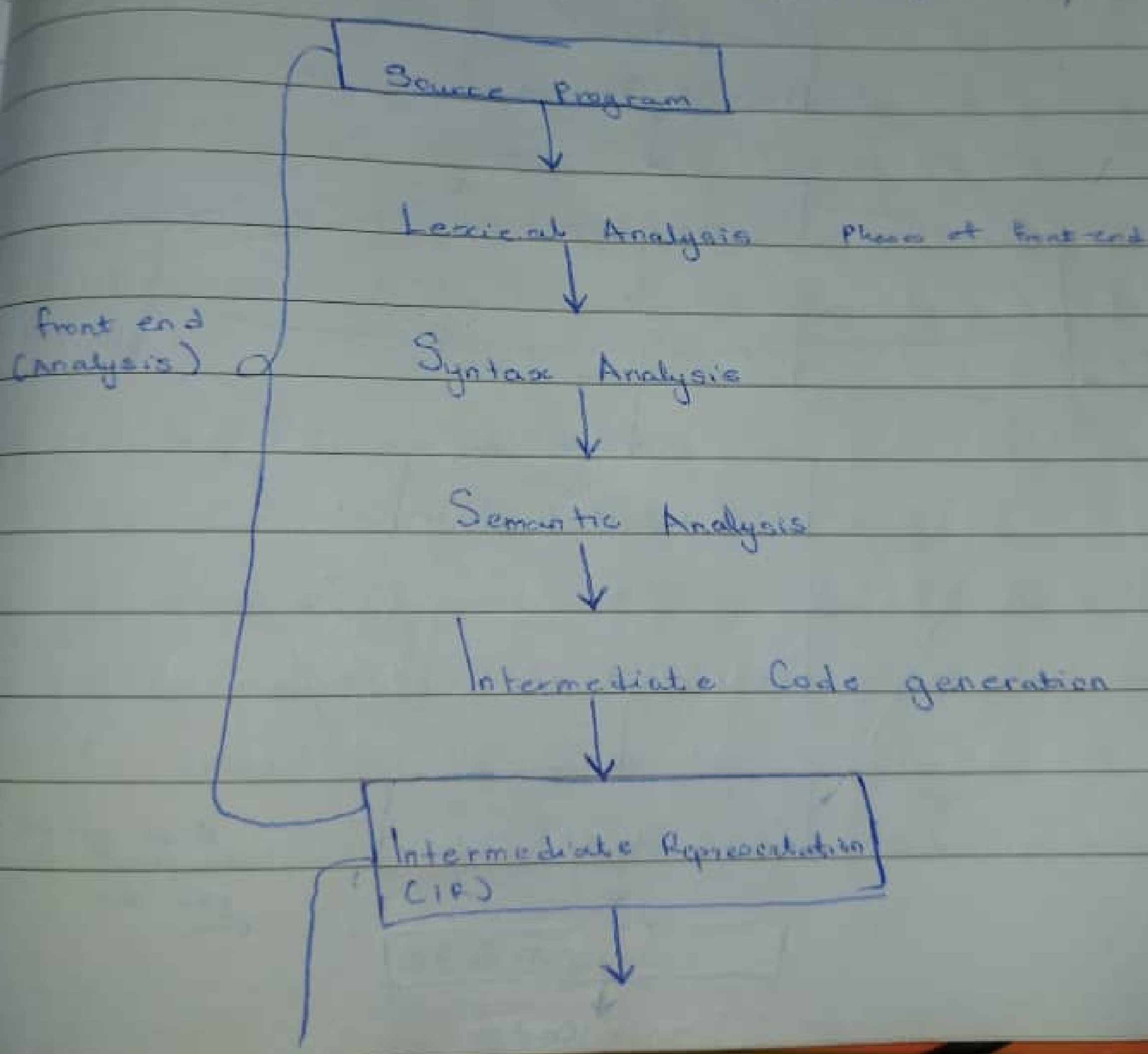


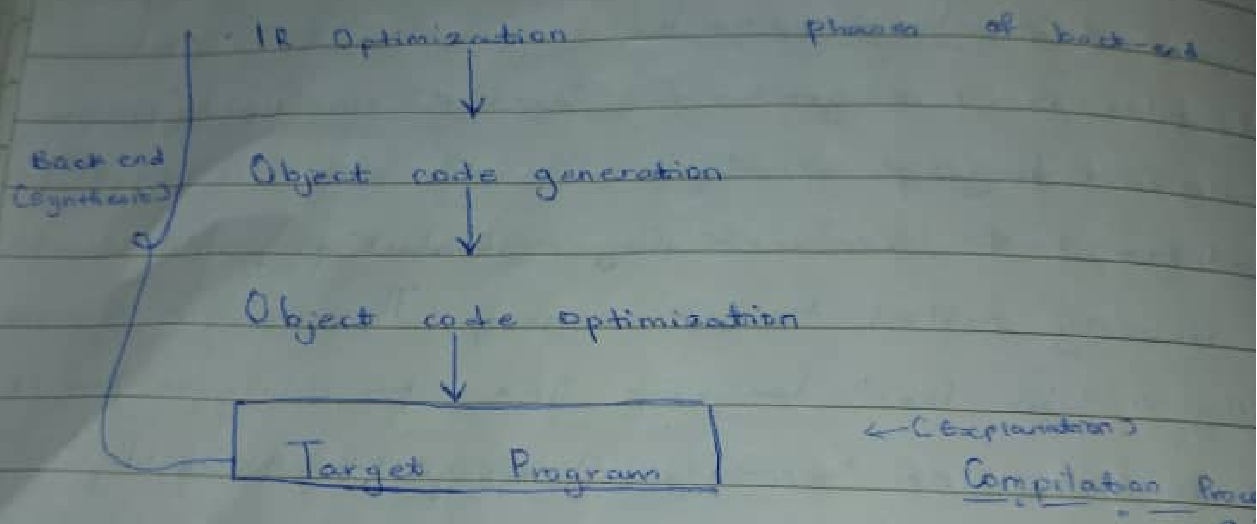
So, for a compiled language the conversion from source code to machine executable code takes place before the program is run. This is a very different process from what takes place for an interpreted programming language.

This is somewhat simplified as many modern programs that are created using compiled languages makes use of dynamic linked libraries or shared libraries. Therefore, the executable file may require these dynamic linked libraries (Windows) or shared libraries (Linux, Unix) to run.

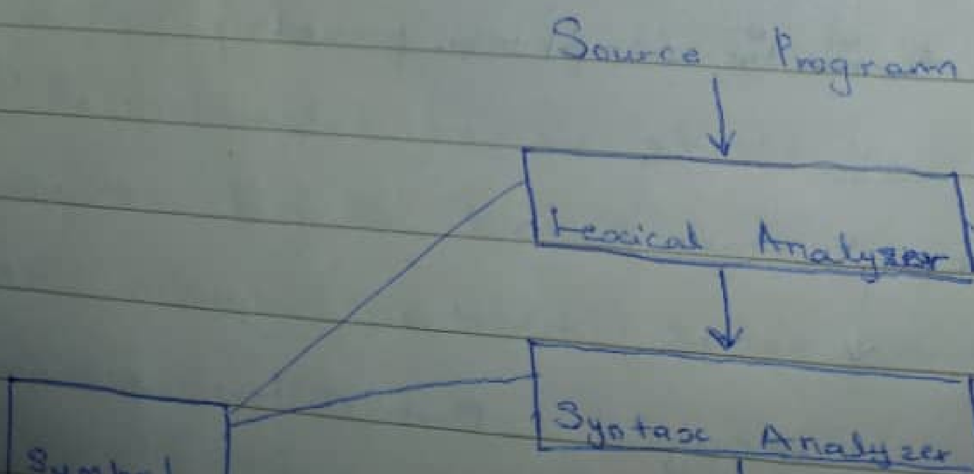
compiler

A compiler is a program that takes as an input a program
...ten in one language usually called the source language
... translates it to a functionally equivalent program in anothe
...guage usually called target language. The source program
... usually a high-level language like C++ and its target
...guage is usually a low-level language like assembly
...guage or machine language. As its translation, the compiler
...n report errors and warnings to help the program make
...rrections to the source so the translation can be
...mpleted. The diagram of the compilation process;





Phases of a Compiler



meaning e.g identifiers, reserved words, integers, doubles or floats, operators and special symbol

Example of lexical analysis;

Consider the C++ code statement given below;

```
int a, b;
a = a+b;
```

A lexical analyzer scanning the code might return

int T- INT (Reserved Word)

a T- IDENTIFIER (variable name)

, T- SPECIAL (Special symbol with value of ",")

b T - IDENTIFER (variable name)

; T - SPECIAL (Special symbol with value of ";")

a T - IDENTIFIER (variable name)

= T- OPERATOR (Operator with value of "=")

a T- IDENTIFIER (variable name)

+ T - OPERATOR (Operator with value of "+")

b T - IDENTIFIER (variable name)

; T - SPECIAL (Special Symbol with value of ";")

2) Syntax Analysis ~~The~~ ~~S~~/Parsing:- The tokens found during the lexical analysis or Scanning are grouped together using a context - free grammar

A grammer is a set of rule governing a languag

[illegible]

[illegible]

[illegible]

In the example shown above, the optimiser was able to eliminate an addition to 'D' and carries out a [illegible] of the same expression, allowing the original five(5) statement to be written in just three(3) statements and use two fewer temporary variables.

7) Object Code Generation :- This is where the target program is generated. The output of this phase is usually machine code or assembly code. Memory location is selected for each variable. Instructions are chosen from each operation. The Three-Address code is translated into a sequence of assembly or machine language instructions that perform the same task.

4/5/2020

8) Object Code Optimisation - There may also be another optimisation pass that follows code generation. This time, transforming the object code into tighter, more efficient object code. This is where we consider features of the hardware itself to make efficient usage of the processors and the registers. The compiler can take advantage of machine specific idioms in reorganising and streamlining the object code itself as with IR optimisation, this phase of

[illegible] of the code which either never [illegible]
[illegible] unreachable or their output is never used are eliminated

i) Strength Reduction:- In this technique,
As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive arithmetic operations with the simple and cheaper ones.

26/2/2020

1) Intermediate Code Optimization:- The optimizer accept input in the intermediate representation e.g TAC and output a streamlined version still in the intermediate representation.
In this phase, the compiler attempt to produce the smallest, fastest and most efficient running ~~[illegible]~~ result by applying various techniques such as, ~~[illegible]~~ (a) (i.e dead/stop) inhibiting code generation of unreachable code segment. b) Getting rid of unused variables c) Eliminating multiplication by one (1) d) Eliminating addition by 0 (zero) e.g 3+0+a = 3+a e) Eliminate division by 1 (one)

f) ~~[illegible]~~ Loop Optimization:- This is removing statement that are

not modified in a loop so that it doesn't take recomputing [illegible]

g) Common - sub expression :- An expression that can still be [illegible] time so they are processed faster

h) Strength reduction :- Breaking down complex expressions [illegible] [illegible] being really

The optimization phase can slow down the compiler, typically it is an optional phase. The compiler may even have fine-grained control that allows the developer to make trade offs between time spent compiling versus optimization quality

Example

$t_1 = b * c$

$t_2 = t_1 + 0$

$t_3 = b * c$

$t_4 = t_2 + t_3$

$a = t_4$

How do you optimise it using a compiler?

Answer

$t_1 = b * c$

$t_2 = t_1 + 0 = t_1$

$t_3 = t_1$

$t_4 = t_2 + t_3$

$= t_1 + t_1$

Compile program - what you get after translation

The C++ on the left is translated into a sequence of four(4) instruction of the three - address code on the right.

Note: the use of temporary variables that are created by the compiler as needed to keep the number of operands down to three(3)

The Synthesis Stage / Back-end

There can be upto three (3) phases of a compilation processing in a compiler.

1) The intermediate code optimization (using the minimal amount of space, speed, overall efficiency in the best way possible)

Assignment

Discuss at least five(5) optimization techniques being used by a compiler.

When we get to a point in a parse where we have only tokens we can [illegible] by scanning which when we are used to parse, we can determine the structure present in the source program.

5) Semantic Analysis

The parse tree or derivation is checked for semantic errors in a statement that is syntactically correct (associate with a grammar rule correctly) but disobeys the semantic rules of the source language. Semantic Analysis is the face where the compiler detects such things as use of an undeclared variable. A function call with the proper argument, access violations, incompatible operands and type mis-matches

Examples of Semantic Analysis

```
Int arr[2], C;
C = arr * 10;
```

Most Semantic analysis partake to checking of types. Although the c++ fragment above will scan into valid tokens and successfully match the rules for valid expression It is not semantically valid. In the semantic Analysis phase, the compiler checks the types and reports that you cannot use an array variable in a multiplication

Elements in the symbol table depends on your code ie it is not static, it is dynamic., the equivalent statement will be matched with the machine language
Error handling:- This is when during compilation phases, there will be error encountered so it is use to handle such situation
of the compiler is usually configurable or can be skipped entirely. ie It is not compulsory *

... stage constructs the desired target program from the intermediate representation.

Typically, the compiler analysis stage is called the front-end and the synthesis stage is the back-end. Each of the stages is broken down into a set of phases that handles different part of the tasks.

Analysis Stage (Front end)

There are four (4) phases in this stage,

1) Lexical Analysis /Scanning :- A stream of characters ([illegible]) making up a source program is read from left-to-right and group into tokens

Tokens are sequence of characters that has a collective

It is a set of rules that defines valid structure in a programming language. Each token is associated with a specific rule and group together accordingly, this process is called Parsing. The output of this phase is called a parse tree or a derivation. A record of which grammar rules were used to create the source program. Examples:- Part of a grammar for a simpler arithmetic operation of a C++ can look like this,

Expression → Expression + Expression

Expression – Expression (variable/constant)

Variable → T-IDENTIFIER

Constant → T-IDENTIFIER

T-DOUBLE CONSTANT / Integer Constant

Note the symbol on the left side of the "→" in each item can be replaced by the symbol on the right. Use the grammar above to parse the statement (a+2)

answer; expression → Expression + Expression

(a) (2)

→ Variable + Expression

→ T-IDENTIFIER + Expression

→ T-IDENTIFIER + Constant

→ T-IDENTIFIER + T-Integer constant

Use double (2.3)

expression and that the type of the right operand of the assign-ment is not compatible with the left.

1) Intermediate Code Generation

This is where the intermediate representation of the source program is created. We want this representation to be easy to generate and translate into the target program. The representation can have a variety of forms but the former one is called Three-Address Code (TAC). (It is called three-address code because the maximum amount of operand is three.)

You reduce operands by using temporary variables

TAC is a lot like generic assembly language.

Three-Address Code is a sequence of simple structures each of which can have at most three (3) operands e.g.

Example of intermediate code generation;

$a = b * c + b * d$

Rule;

1) The variable at the left hand-side must be maintained at the final answer i.e. a;

2) You represent temporary variable using t_1, t_2, t_3

(each of the line must not be three operands)

$t_1 = b * c$

$t_2 = b * d$

$t_3 = t_1 + t_2$

$a = t_3$