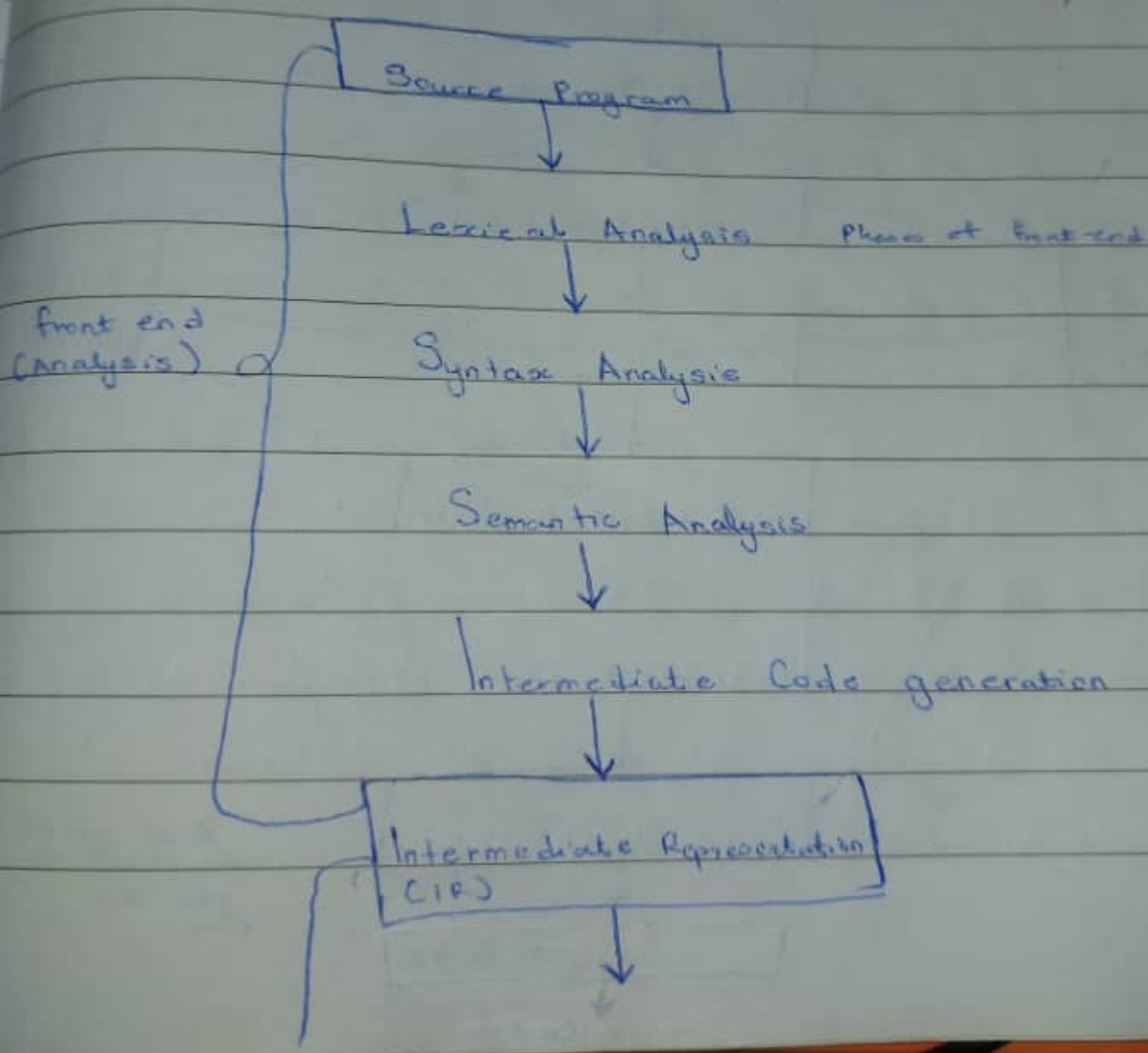


A compiler is a program that takes as an input, program written in one language usually called the source language and translates it to a functionally equivalent program in another language usually called target language. The source program language is usually a high-level language like C++ and its target language is usually a low-level language like assembly or machine language. As its translator, the compiler reports errors and warnings to help the programmer make corrections to the source so the translation can be completed. The diagram of the compilation process;



not needed in a loop so that it doesn't take unnecessary

g) Common-sub-expression: An expression that can still be
calculated once so they are precalculated for later

h) Strength reduction: breaking down complex expressions into simpler ones
The optimization phase ^{usually} ~~usually~~ comes later in the
compiler, typically it is an optional phase. The compiler
^{may} ~~can~~ ^{might} even have fine-grained control that allows the
developer to make trade offs between time spent ~~computing~~
versus optimization ~~problem~~ 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

$$t_1 = b * c$$

$$t_2 = t_1 + 0 = t_1$$

$$t_3 = t_1$$

$$t_4 = t_2 + t_3$$

$$= t_1 + t_1$$

Strength Reduction - In this technique

As the name suggests, it involves reducing the strength of expressions

This technique replaces the expressions with simpler and cheaper ones

26/2/2020

d) Intermediate Code Optimization - The optimizer accepts input in the intermediate representation eg TAC and output a streamlined version still in the intermediate representation

In this phase, the compiler attempts to produce the smallest fastest and most efficient running ^{result} ~~expression~~ by applying various techniques such as; ~~parallel~~ 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. $B + 0 + a = B + a$
- e) Eliminate division by 1

f) Loop Optimization - This is removing statements that are

In the example above, the optimizer can write instructions in addition to 'i' and 'j' and a new instruction of the same expression, using the original 'i' and 'j' statement to be written in just three 'i' statements and use two extra temporally variables.

2) 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 instruction and 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.

#1/3/2020

3) 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

meaning e.g. identifiers, reserved words, constants, operators and special symbols

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 - IDENTIFIER (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)

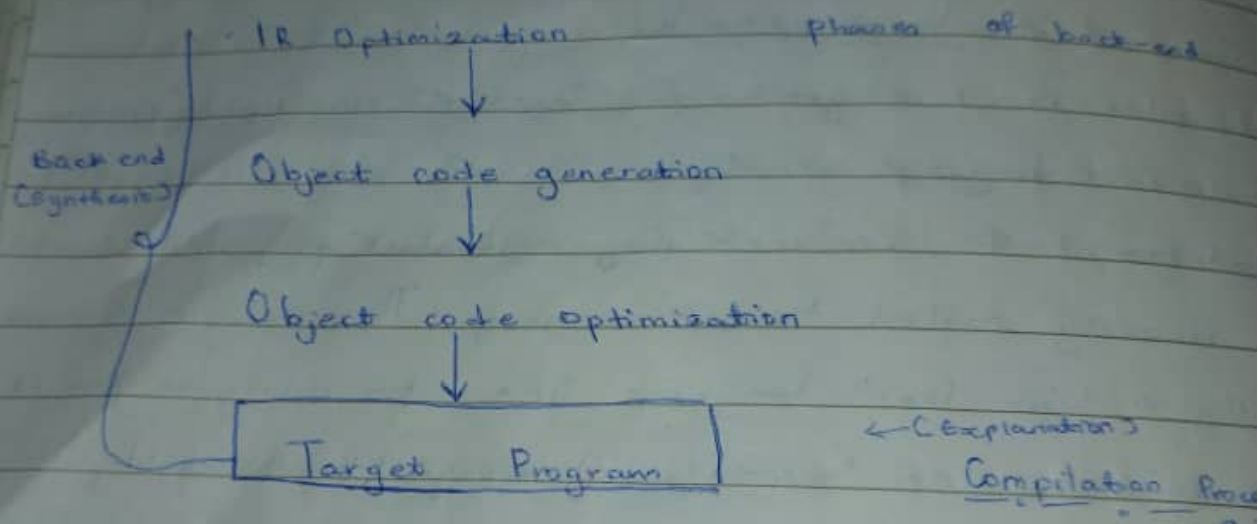
; T - SPECIAL (Special Symbol with value of ";")

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

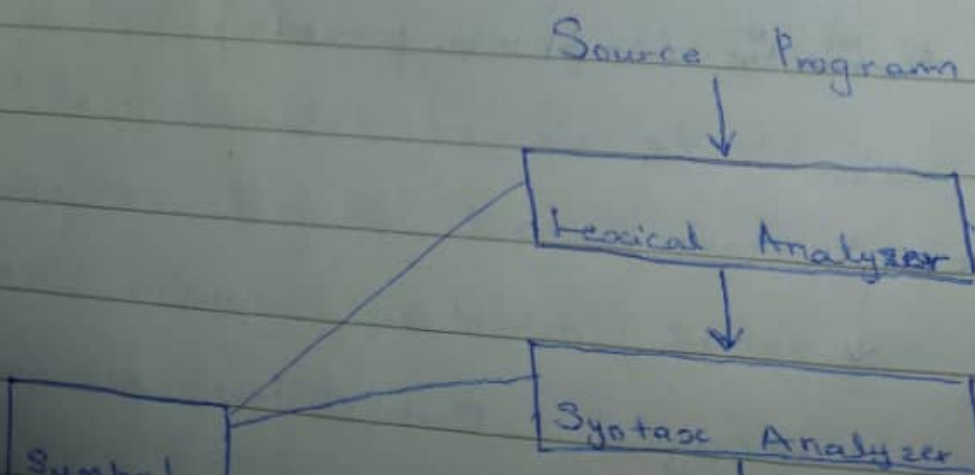
A grammar is a set of rules governing a language

Elements in the symbol table depends on your code i.e. 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 used to handle such situation.

of the compiler is usually configurable or can be skipped entirely. i.e. It is not compulsory.*



Phases of a Compiler



Compile program - what you get after translation

The C++ on the left is translated into a sequence of four(4) instructions 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) ~~subsequent~~ phases of a

1) The intermediate code optimization processing in a compiler.

1) The intermediate code optimization (using ^{speed, overall efficiency} ~~reduce~~ the minimum amount of space 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 be finished by scanning what ideas we are used to parse, we can determine if the structure present in the source program

2) Semantic Analysis

The parse tree or derivation is checked for semantic error is a statement that is syntactically correct (associated with a grammar rule correctly) but disagrees to 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} ~~Through~~ 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 ~~are~~ cannot use an array variable in a multiplication

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

Analysers Stage (Front end)

There are four (4) phases in this stage.

1) Lexical Analysis / Scanning :- A stream of characters (input) is read from left-to-right and making up a source program is read from left-to-right and group into tokens.
Tokens are sequence of characters that has a collective

is a set of rules that define valid structure in a programming language. Each token is associated with a specific rule and put together accordingly, this process is called parsing. The output of this phase is called a parse tree or a derivation. A parse tree is a diagram that shows the hierarchical structure of a program. Example: Part of a grammar for a simpler arithmetic operation of a C++ can look like this,

- Expression \rightarrow Expression + Expression
- Expression \rightarrow Expression (variable/constant)
- Variable \rightarrow \bar{T} - IDENTIFIER
- Constants \rightarrow \bar{T} - IDENTIFIER
- \bar{T} - DOUBLE constant / Integer constant

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

- expression \rightarrow Expression + Expression
- \rightarrow Variable^(a) + Expression⁽²⁾
- \rightarrow \bar{T} - IDENTIFIER + Expression
- \rightarrow \bar{T} - IDENTIFIER + Constant
- \rightarrow \bar{T} - IDENTIFIER + \bar{T} - Integer constant

Use double (2.3)

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

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. (It is called three-address code because the maximum number of operands (3) 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.

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 is a;

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

$$t_1 = b * c$$

$$t_2 = b * d$$

$$t_3 = t_1 + t_2$$

$$a = t_3$$

(each of the line must not be three operands)