Compiler Design :-

Date (1)

Text Books

- Compilers "Principles, techniques and tools" - Aho, Lam, Sethi and Ullman - Pearson Publication
- Compiler Design - Dr O.G Kakde (University Science Press)
- Compiler Design - Rajesh K. Maurya - Wiley Press, New Delhi
- Compiler Design - Jay Kant P.S. Yadav, Rajeev Ranjan (Vayu Education of India).

Unit I - Introduction to compiler

Introduction :-

High level language play a vital role in design of software. But these languages in which we want to express our computational needs are significantly different from the language recognized by real machine. It is here that language translator plays an important role. Let us discuss what is translator ?

Language translator or Language processor :-

A Language translator or Language processor (or simply translator) is a program that translates an input program written in a programming language (called source program) into a functionally equivalent program in another language (called target program). Apart from translation, a translator should have error detection capability. Any violation of source language specification would be detected and reported to the programmer. The working of translator shown in fig 1.1.

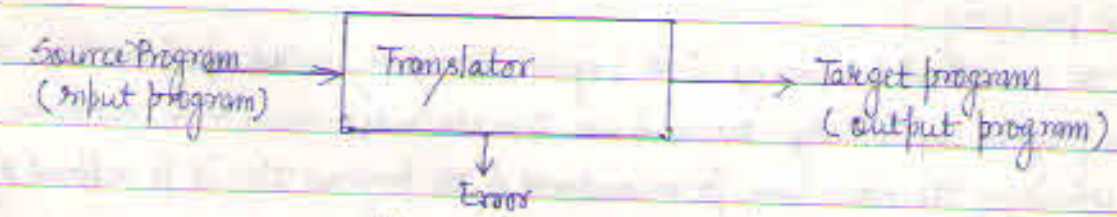


Fig 1.1 : Language translator or processor

Language processing system (causing of compiler)

As we know that any computer system is made of hardware and software. The hardware understands a language of 0,1 which human cannot understand so we write program in high level language, which is easier for human to understand and remember. These prog.

Date

ram are then fed into a series of utilities (programs) and operating system components to get the desired code that can be used by the machine. This is known as language processing system. Fig 1.2 illustrates language processing system

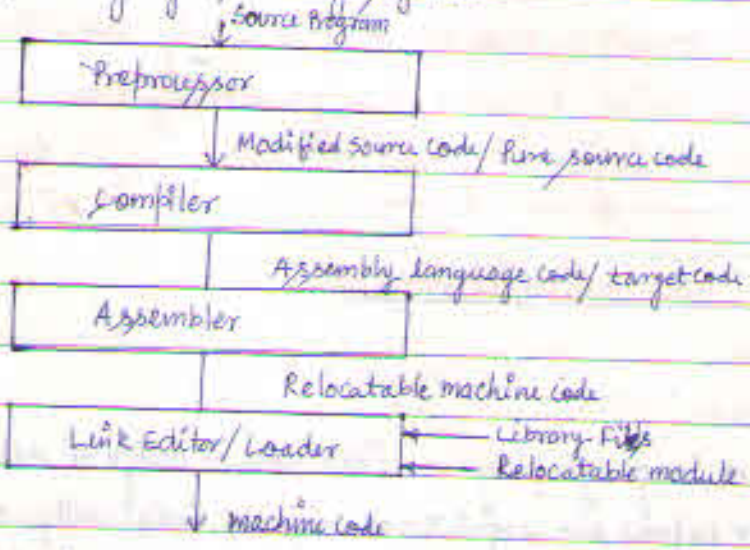


Fig 1.2 : Language Processing System

(i) Preprocessor :- It takes the source code as input and produces modified source code as output.

Before compilation, the source program is preprocessed by preprocessor to prepare it for the compilation. the preprocessor program creates modified source program from the original source program by replacing preprocessor directives with the suitable content. The new source program acts as an input to the compiler. The preprocessor performs various tasks as given here :-

(a) Macro processing :-

It permits the user to include macros in the program. Macros are smallest of instructions that are used in a program repetitively. Macro have two attributes macro name and macro difinition whenever the macro name is encountered in the program then it is replaced by macro difinition (set of statements corresponding to macros.

For example :- # difine identifier replacement strings
(macro name) (macro difinition)

Eg :- # difine Array_Size 50 is the macro : and the program statements

```
int array1 [Array_Size];
int array2 [Array_Size];
```

The preprocessor would replace Array_Size in body of source code and result of micro processing is

```
int array1[50];
int array2[50];
```

(b) File inclusion :- It permits the user to include the header file in the program and user can make use of function defined in these header file.

when preprocesser encounters # include directive in the program text, it would immidiately replace it by the entire content of specified file.

(c) Rational preprocessor :- These preprocessor enhances the capabilities of older languages by providing the programmer with built in macros for constructs that support flow of control and data structuring facilities. Some older languages did not have constructs such as while statements and if statement in them; these can be easily augmented through Rational preprocessors.

(d) Language extension :-

Language extensions are preprocessing facilities that enhance the capabilities of the language through concepts such as built in macros. The language extension statements are translated into procedure calls or routines to perform desired operations such as data base access or similar activities.

For example :- the language Equel is database query language embedded in C. Statements beginning with "## are taken by preprocessor to be database statements and statement related to C are translated into procedure calls on routines that perform database access.

Note :-

(i) If program is too large, it may be stored in different files then the preprocessor will takes of all these files

(ii) preprocessor stripped out all comments in a program.

(iii) preprocessor is optional eg the language that donot support # symbol are not used preprocessor.

(ii) Compiler :- It takes modified source code as input and produces target code as output target code may or may not be assembly language code

Note :- compilation is also optional eg. HTML, Java script, 8085

(iii) Assembler :- Some compiler produces assembly language code that is passed to the assembler for further processing other compiler directly produces relocatable machine code that can directly passed to the linker. Assembly language code is mnemonics version of machine code in which names are used instead of binary codes, for

Date

operations and names are also given for memory addresses.

(iv) Link editor and loaders :-

The large source programs are compiled in small pieces by compiler. To run the target machine code of any source program successfully, there is a need to link the relocated machine language code with library files and other relocatable object files. So loader and linker program programs are used for link editing and loading of relocatable codes. link editor creates a single program from several files of relocatable machine code. Loader read the relocatable code and alter the relocatable addresses. To run the machine language program, the code with altered data and commands is placed at correct location in memory.

Compiler

A compiler is a computer program that translates source code written in high level language like C or C++ into functionally equivalent target language code. the target language is usually assembly language or machine language. During the translation process, compiler also informs the programmer about the error present in the source program (see Fig 1.3)

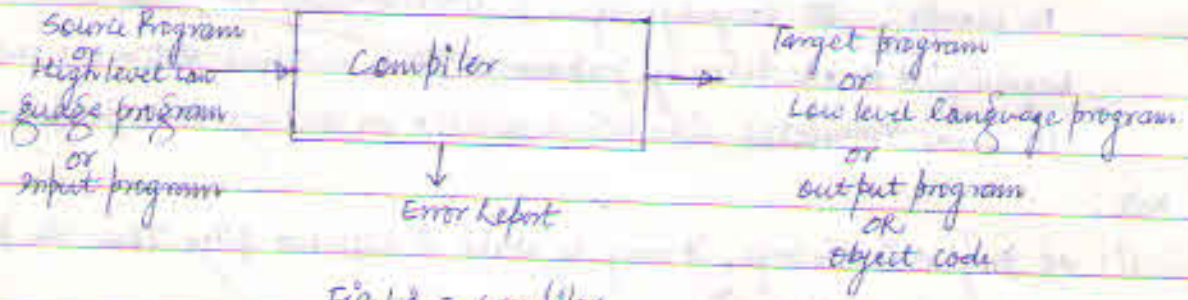


Fig 1.3 - compiler

Ex: what is compiler? what is its primary function? what is its secondary function?

Compiler converts source program written in high level language into functionally equivalent machine or object or target code. Then target code will called by the user to process the input to produce the output.

the compiler executes the entire code at a time, if any error occurs in the code, all of them will be reported at a time.

Fig 1.4 : Compilation model

Interpreter :- In interpreter, data and source program are input to interpreter. Instead of producing any object code or executable code as in compiler, the interpreter produces the result by performing operations of the source program on its data. This model is represented as shown in fig 1.5

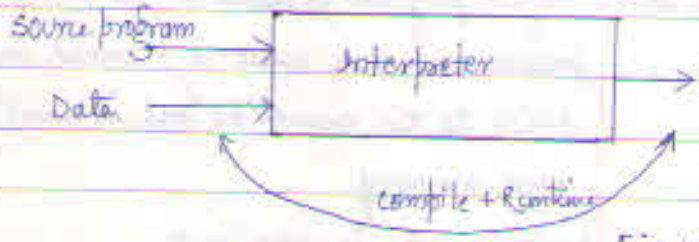


Fig 1.5 interpretation model

Interpreter directly executes the source program line by line according to the given input. The translation and execution of each statement are carried out side by side. There seperate execution of the source program is not required. The line by line execution of the source program provides better debugging environment (error finding) than a compiler because it can check for errors like out of bound array indexing at run time.

The main drawback of interpreter is that the execution time of interpreter program is generally slower than that of a compiled program because the program needs to be translated every time it is executed.

The interpreter has certain advantages over compiler as it can handle certain language features which can not be compiled eg language like APL are normally interpreted as it involves features about the data such as size and shape of arrays which cannot be deduced at compile time. Another advantage is that interpreters can made portable as they donot produce machine code programs they also save the time required for assembling and linking a real program.

Difference between compiler and interpreter

S.NO	Compiler	Interpreter
1.	It is translator	1- It is not a translator it is a simulator.
2.	In compilation process runtime and compile time are different	2- In interpretation process, runtime and compile time are same
3.	The compiler reads input program and translate it only once	3- The interpretation read the input program over and over to compute results.

Date _____

4- It produces object code (or .exe file)	4- It donot produces object code file, it simply runs the program.
5- The execution time of the compiler is less than that of interpreter because compiler reads the input program only once and occupies less space in memory.	5- The execution time of interpreter is more than that of the corresponding object program because in the interpretation process each line must be scanned and parsed prior to the execution and it occupies more memory.
6- Debugging is very slow	6- Debugging is very fast.
7- Size and complexity are more	7- Size and complexity are less.
8- It is used to designing stand alone applications	8- It is not used to design stand alone applications.
9- The compiler reads the sequence of source lines as printed not as executed	9- The interpreter reads that the statements in order of execution rather than as printed.
10- Compiler reads the whole source program code at once and report all error at last Example:- C/C++	10- Interpreter reads a statement from input execute it on data, if any error occurs an interpreter stop execution and report it. Example: Command language, script language, batch language (LISP, PYTHON)

Part of Compiler

As we know and illustrated in fig 1.2, the compiler takes the preprocessed file as the input and translates it into an equivalent assembly language file or machine code file. Here we will get an overview of how a compiler translates a preprocessed input file into a assembly code file or machine language file.

The translation of input source program file (preprocessed) into target assembly language file can be divided into two stages called as:

- Analysis of a source program (front end)
- Synthesis of a source program (back end).

The analysis part or front end of compiler transforms the input source program into intermediate code. The intermidiate code (some times called intermidiate representation IR) is a machine independent representation of input source program.

The synthesis part (back end) of the compiler takes machine independent intermidiate code and generates the target assembly language code. the synthesis part or back end deals with machine specific details like registers, number of allowable operators and so on.

Fig 1.6 illustrates the two stage design approach of a compiler

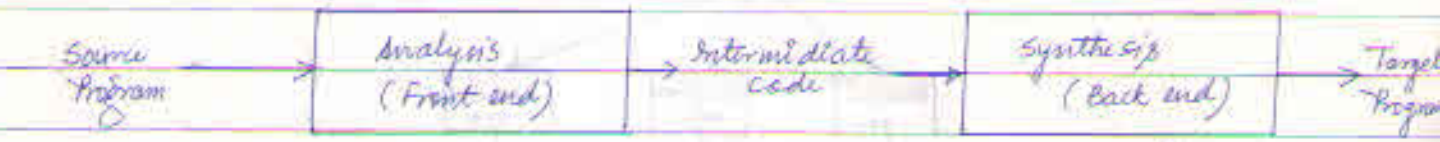


Fig 1.6 - The analysis - synthesis model of compiler

The Main advantage of having two stage approach are:

- Take the front end of a (source) compiler and attach different back end to produce a compiler for the same language on different machine (refer Fig 1.7 (a))
- Take the common back end for different front ends to compile several different language on the same machine (see Fig 1.7 (b))

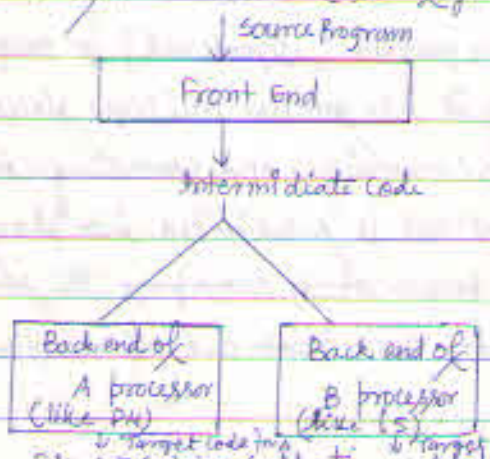


Fig 1.7 (a) - Supporting additional machine by adding back end.

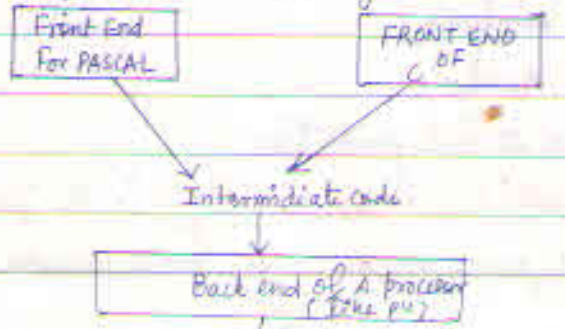


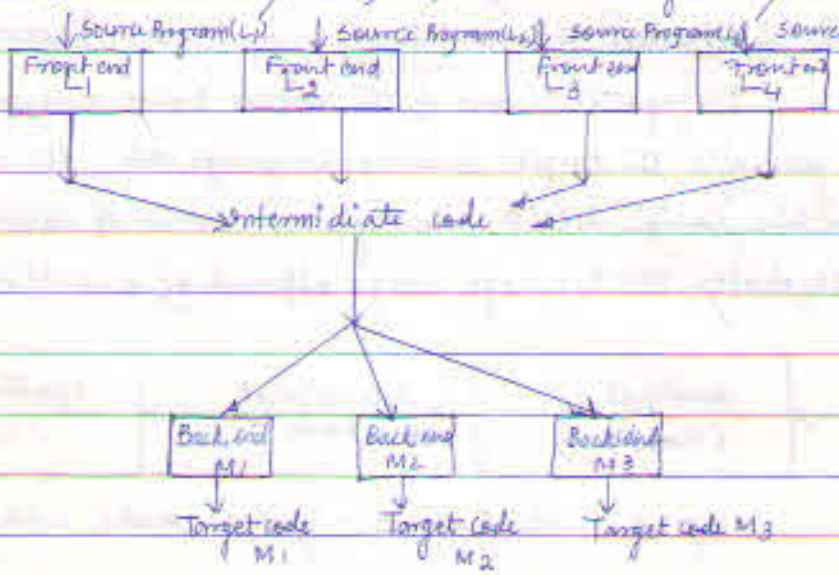
Fig 1.7 (b) - Supporting an additional language by adding front end.

Date _____

This organization of compiler is called contamporary compiler.

Example :- Suppose we want to create compilers for four languages L_1, L_2, L_3, L_4 over three different machines M_1, M_2, M_3. How many codes we have to write down?

Sol^n -



So instead of producing $4 \times 3 = 12$ compiler code we will simply write down four code for front end of every language and three Back end for each of machine. So total code required to construct the compiler for each language over every machine will be $4 + 3 = 7$.

(1) Analysis of source program :-

The analysis of the source program (front end) is responsible for analysing the input source by breaking it into smaller entities (pieces) and checking syntax and verifying the semantics (meaning) and generating intermidiate code.

the front end is subdivided into phases. A phase is an independent task in the compilation process which transforms the source program from one representation to another. The front end of compiler consists of the following phases

- ✓ Lexical Analysis
- ✓ Syntax Analysis
- ✓ Semantic Analysis
- ✓ Intermidiate code generation

(i) Lexical analysis (also known as scanning) is the first phase of compiler. Lexical analyzer or scanner reads the source program in the form of character stream and groups the logically

related characters together that are known as lexemes. For each lexeme, a token is generated by the lexical analyzer.

A stream of tokens is generated as the output of the lexical analysis phase which acts as input for the syntax analysis phase. Tokens can be of different types, namely keywords, identifiers, constants, punctuation symbols, operator etc. the syntax of any token is < token_name, value >

where token name is the name or symbol which is used during syntax analysis phase and a value is the location of that token in the symbol table.

2. Syntax analysis phase (parsing.)

Syntax analysis phase also known as parsing. parser uses token name taken from from the token stream to generate the output in the form of tree like structure known as syntax or parse tree.

syntax analysis checked, if they form a valid sequence as defined in programming language. so parse tree illustrate the grammatical structure of the token stream.

Note :- Due to parse tree arrangment of stream of tokens. this phase is also known as hierarchical analysis.

(3) Semantic analysis :- In semantic analysis, we check if the syntatically correct statements make a meaningful reading. For example, a statement in the input source program "X = Y + Z" would not make a meaningful reading. if say, x is the name of function or array, and y is a float type of identifier. This statement might by syntatically acceptable by the productions of the context free grammar in syntax analysis but would not hold out during semantic analysis because the data types of x and y are not compatible.

In natural language parlance, this is very similar to having a grammatically correct sentence, but devoid of meaning. For example, the syntax rule would also accept a sentence, "The bicycle rides the boy". This sentence does not make sense so we would reject it during semantic analysis.

In similar way a c statement "myfun = y" where "myfun" is the name of function and y, a float type identifier might be acceptable in syntax analysis but would be rejected in semantic analysis, since the data types of left hand

Date_____

side and right hand side donot match.

Most of semantic analysis resolves around such type checking other task of semantic analysis involves detection of undeclared variable, access violations (classes accessed correctly or not) and so on.

The result of semantic analysis is annotating the parse tree with more information on the data types.

4- Intermidiate code generation :-

In intermidiate code generation phase, the parse tree representation of the source code is converted into low level machine like representation (easy to generate from source program and easy to convert (translate) into machine language ✓

In intermidiate code generation phase, we walk through the annotated parse tree and generate intermidiate code. The three address (or postfix) is one of the common forms of intermidiate code. Three address code is a sequence of instructions each of which can have atmost three operands.

(ii) Synthesis of source Program (Back end) :→

The synthesis of source program (Back end) of compiler is responsible for translating the machine independent intermidiate code into target code (either machine code or assembly code). The back end of the compiler depends on target processor, where final binary would be executed

The back end or synthesis of part of the compiler consists of the following phases

- ✓ code optimization
- ✓ code generation

(5) Code optimization :-

Code optimization phase, which is an optional phase, performs the optimization of the intermidiate code, optimization means making the code shorter and less complex, so that it can execute faster and takes lesser space, so that it can execute faster and takes less space. The output of code optimization phase is also an intermidiate code, which performs the same tasks as the input code but requires lesser space and time.

(6) Code Generation :- Code generation phase translates the intermidiate code representation of the source program into the target program (machine or assembly code). If the target

program is machine language code, the code generator produces the target code by assigning registers, and memory locations to store variables defined in the program and to hold the intermidiate computation results. The machine code produced by code generator can be directly executed on the machine.

In case of assembly code, mnemonics code is generated which translated by assembler and other utility in the machine language code further.

Besides three, there are two other phases which interact with all the other phases of compiler. These are:

- Symbol table Management
- Error detection and handling

(a) Symbol table Management :-

A symbol table is a data structure that is used by compiler to record and collect information about source program constructs like identifiers (variables) and subroutines (procedure) in the source program.

For identifiers (variables), in addition to its name, table contains information about the type, scope and storage space of identifiers.

For subroutines (procedures) it records the number and types of its parameters, the way each parameter passed and return type of the sub routines (if there is any).

Symbol table management is an important function of each phase of the compiler. During lexical analysis phase, lexical analyzer identifies tokens and put them into symbol table. About type and other information or any other information is entered by syntax and semantic analysis phase.

For ex:-

For following code segment in C, the symbol table will be given below

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

// now it will read each identifier & entry will be done in symbol Table.
// If already entered so it will not be reentered

Entry No	Symbol	Type	Length	...	Address
1	a	integer	2		
2	b	Float	4		
3	c	Float	4		
4	a*	Float	4		
5	T_1	Float	4		

compiler type cast variable 'a' to a* in float type from integer

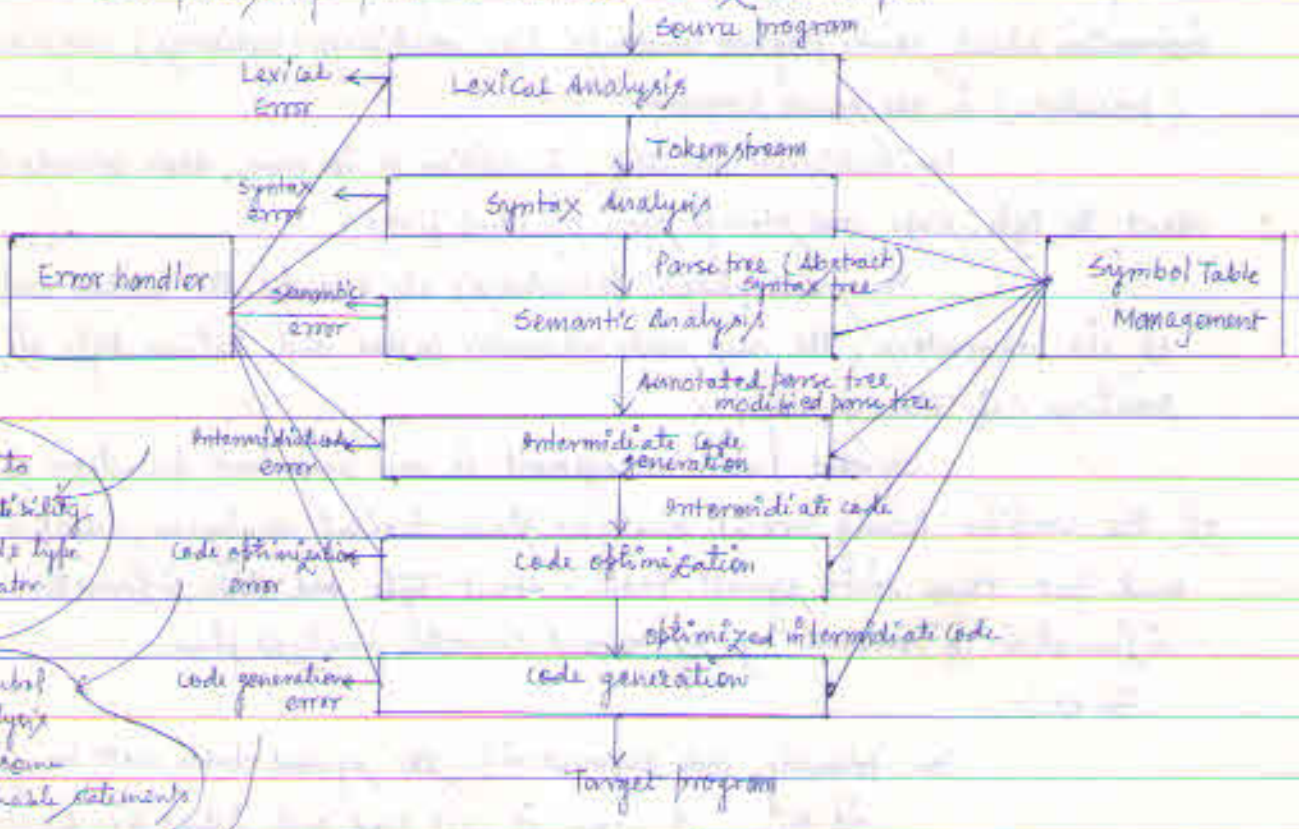
Date_____

(b) Error detection & handling

Program are written by programmer (human) and hence cannot be free from errors. Each phase encounters errors. The lexical analyzer detects all errors when the remaining characters in the input donot form any token. The syntax analyzer usually detect a large number of errors where the token stream violates the structural rules of language. The type mismatch type errors are usually detected by semantic analyzer (eg. addition of character variable to an integer variable)

Whenever, a phase encounters an error, it must report it, in addition, it must somehow deal with that error. Error handler is invoked. Error handler generates a suitable message allow programmer to find out exact location of error.

The analysis and synthesis model of compiler in fig 1.8, consolidating the phases of front end and back end of the compiler



Error due to incompatibility of operands type for operator

during control flow analysis due to some unreachable statements

Incompatibility with computer architecture

Fig 1.8 phases of compiler (analysis & synthesis model)

Example of compilation :-

Let us consider the translation of the following statement

$$x = y + z * 10 \quad \text{---} \quad (1.1)$$

The internal representation of source program changes with each phase of the compiler

(1) Lexical Analysis phase

The lexical analyzer builds uniform descriptors for the elementary constituents of the source string. To do this it must identify lexical units in the source string and categorize them into identifiers, constants, keywords, operators etc. This uniform descriptor is called token. A token has the following format

category	lexical value

For example, when lexical analyzer finds an identifier x, it generates a token, say id and also enters 'x' into the symbol table, if it is not already there. The token for x will be <id, #7>, if x is 7th entry in symbol table. The lexical value associated with this occurance of id contains a pointer (or index) to the symbol table entry for x.

A token is generated / constructed for each identifier / constant as well as each operator in the string. Let it assign id1, id2, and id3 to x, y, z respectively, and assign, addop to '=', multop to '*', addop to '+' and num to 10. Now representation of statement (1) after lexical analysis may be given as

$x = y + z * 10$

↓

Lexical Analysis

↓

id2 assignop id2 addop id3 multop num ---(2)

(2) Syntax analysis phase

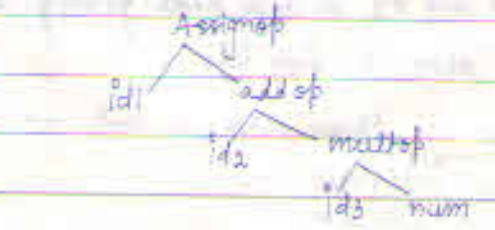
Stream of token acts as input for the syntax analyzer. output of the syntax analyzer is a parse (syntax) tree that acts as input to semantic analyzer. The parse tree for statement (2) is shown in fig

id1 assignop id2 addop id3 multop num

↓

Syntax analyzer

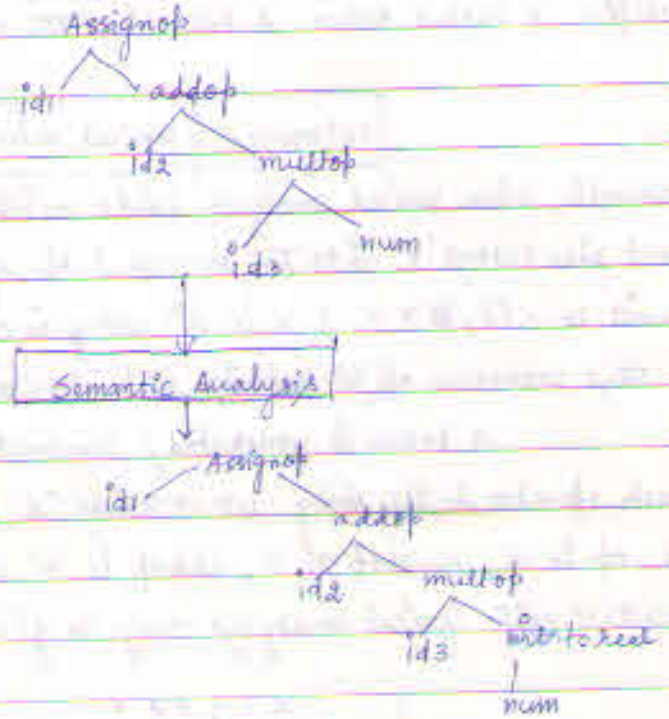
↓



Date_____

(iii) Semantic analysis phase

The semantic analyzer determines the meaning of a source string: it findout that the types of operands in fig are not identical as id3 is real (float) and num is identifier. Therefore, it convert 'num' to type real so that the syntax tree now looks as in figure



(iv) Intermidiate code generation :-

After semantic analysis, the compiler generates an explicit intermidiate code representation of source program this intermidiate code could be three address code or postfix (polish) code but it should be easy to produce and easy to translate into target program

For syntax tree shown in above figure the following steps must be taken in the intermidiate code generation

1- convert num to real giving t_1
2- multiply t_1 and id3 in type real giving to t_2
3- Add id2 and t_2 in type real giving to t_3
4- Finally store t_3 into id1

These steps shown in three address code as follow

t_1 = int to real (num)
t_2 = id3 * t_1
t_3 = id2 + t_2
id1 = t_3

(v) code optimization phase :-

the code optimizer tries to improve the intermidiate code in order to achieve faster running machine code. In our example we can convert num from integer to real once at compile time (no need of conversion at execution time). Also we can substitute id1 for t3 as t3 is used only once to transmit it value to id1. Now intermidiate code (of previous one) can be rewritten as

$$t_2 = id_3 * rnum$$
$$id1 = id2 + t_2$$

(vi) code generation phase

finally, compiler can generate the target code for the intermidiate code. The storage must be allocated and registers must be assigned to the variables and addressing modes to be used for accessing the data must be decided before generating code.

if we use two register R_1 and R_2, the code can be written as

```
MOVF  R2, id3
MULF  R2, rnum
MOVF  R1, id2
ADDF  R1, R2
MOVF  id1, R1
```

the first operand in each instruction specifies destination, 'F' in each instruction indicates distination - real value.

Questions :-

illustrate the compilation process on the following statements

(1) x = y * z + 10

(2) I = (P X R X t)/100

Soln :- Here we illustrate compilation process of each statement

PAGE NO

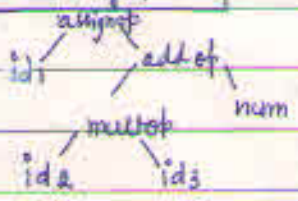
DATE

x = y * z + 10

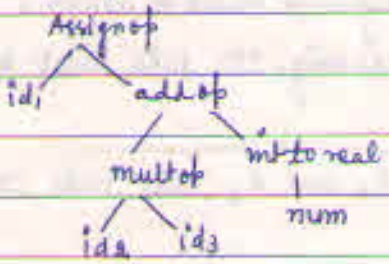
Lexical analysis

id1 Assignop id2 multop id3 addop num

Syntax Analysis



Semantic Analysis



Intermidiate code Generation

$t_1 = id_2 * id_3$

$t_2 = inttoreal(num)$

$t_3 = t_1 + t_2$

$id_1 = t_3$

Code optimization

$t_1 = id_2 * id_3$

$id_1 = t_1 + rnum$

Code generation

MOVF R_2, id_3;

MULTF R_2, id_2;

MOVF R_1, rnum;

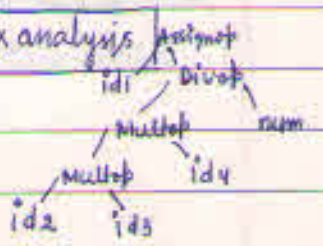
ADDF R_1, R_2; MOVF id_1, R_1

I = (b*r*t)/100

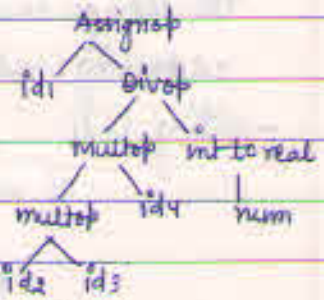
Lexical Analysis

id1 assignop id2 multop id3 multop id4 div num

Syntax analysis



Semantic Analysis



Intermidiate code generation

$t_1 = id_2 * id_3$

$t_2 = t_1 * id_4$

$t_3 = inttoreal(num)$

$t_4 = t_2 / t_3$

$id_1 = t_4$

Code optimization

$t_1 = id_2 * id_3$

$t_2 = t_1 * id_4$

$id_1 = t_2 / rnum$

Code Generation

MOVF R_1, id_2

MULF R_1, id_3

MOVF R_2, id_4

MULF R_1, R_2

MOVF R_2, rnum

DIVF R_1, R_2

MOVF id_1, R_1