03-06-20

CD - Moodle - Assignment-1

Jovial Joe
IES17CS016

Q1. Differentiate between front end & back end of the compiler.

Q2. Define Lexeme, Token & Pattern

Q3. Discuss the input buffering scheme in lexical analyzer.

Q4. State the uses of symbol table.

Q5. Explain any four comiler writing tools.

Q6. Discuss the importance of bootstrapping to develop a compiler.

Q7. Define Cross-compilers.

Q8. Construct a regular expression to denote a language L over $\Sigma = \{a, b\}$ all strings not ending in ab.

Q9. Construct a regular expression to denote a language L over $\Sigma = \{a, b\}$ accepting the string has ab occuring equally.

Q10. Construct a lexical analyzer for the tokens of relational operators.

Q 11. Construct a lexical analyzer for the token conditional statements (if, else, for).

Q 12. Explain the different phases in the design of a compiler.

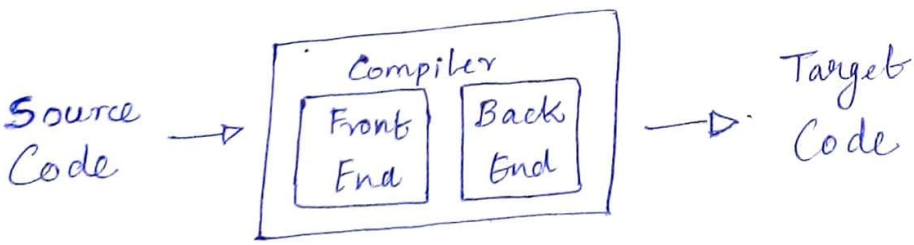
Q 13. Define compiler compiler.

Answers

A1.) Compiler – Frontend & Backend

~ It is trivial that a compiler - compiles and translates the source code to the target language.

~ Therfore the compiler can be depicted as follows.



~ As shown ~~Therefore~~ the compiler can be split into both front-end and back end.

- The front end

~ consists of ~~4~~ sub-components

The back end

~ ~~also~~ consists of ~~2~~ sub components

~ These are
- Lexical Analyser
- Syntax Analyser
- Sematic Analyser
- Intermediate Code-generator

~ And these are
- Code Optimizer
- Code Generator

~ Frontend works with the source code.

~ Backend works with the ~~back end~~ system instructions

~ which is why frontend is system architecture agnostic

~ Backend is usually dependent on the system.

~ The front end parses the code, checks for syntax & sematic erros, and finally generates an intermediate code.

~ The backend obtains the intermediate code & optimizes it finally generate the target code.

~ Both frontend and backend require symbol table to perform their task.

A2.> Lexemes

It is a sequence of characters (in the source language) which is matched by some pattern for a token (eg:- +, -, a, >)

Tokens eg (identifiers, constants, operators etc.)

It is a sequence of characters which are treated as a single meaningful (logical) entity.

Pattern (eg. an identifier in a string with alphanumericals)

It is a set of strings decribed by a rule which produces same token as output.

~ These entities are used in the Lexical analyser phase.

A3.) Input Buffering

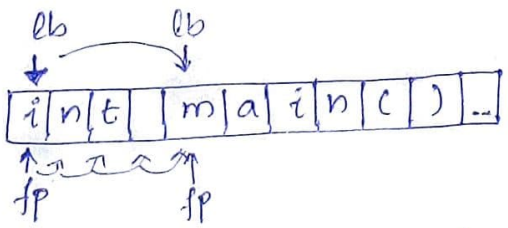
~ The Lexical Analyser scans the source language line by line, character by character from left to right - direction.

~ It scan these character ~~into~~ using two pointers namely:

(i) Lexeme begin (lb)

(ii) Forward pointer (fp)

~ The process is shown below:



eg:-

Source

```
int main() {
   ...
}
```

~ When fb reaches a whitespace then lb moves to the next set of characters.

~ At the same time the scanned token is moved into the memory.

~ This is where buffering is required.

~ Each system call to bring in a character is expensive in terms of CPU cycles.

~ To avoid this an intermediate buffer is kept to which source code is brought in character by character.

~ Then from there it is moved to the main memory block by block thus requiring less number of system calls.

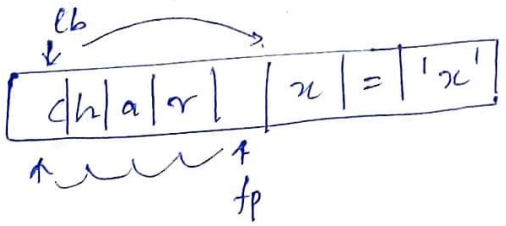
~ Now there are two types of input buffering scheme:

(i). One Buffer Scheme
(ii) Two Buffer Scheme

One Buffer Scheme

~ Single buffer of fixed size exist.

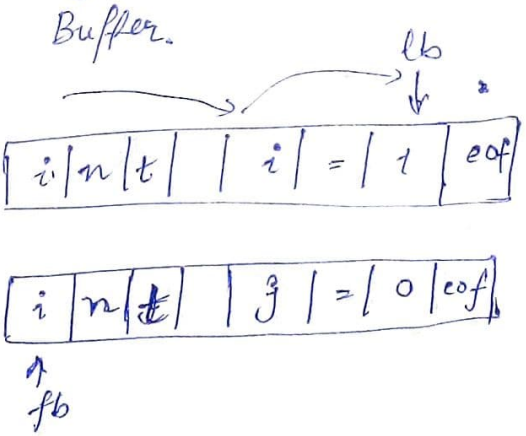
~ Buffer over flow may occur when bring in tokens from the sourced



Two Buffer Scheme

~ As name suggests, two buffers are used here.

~ The overflow is accomodated by the next (ie. Buffer 2) Buffer.



A4.) Symbol Table

- It is a data structure containing a record for each variable name. (value)
- It contains the fields of the attributes of those names.
- Symbol table is used in nearly all the stages of compiler - from Lexical Analyser to final code generation.
- The attributes that are stored provide information about the memory allocated for a 'name', its type, its scope etc.
- As for the procedures (function) names, things such as the number & type of arguments, method of passing (value/reference), return type etc. are stored.
- which is why the compiler's symbol table is extremly significant.

A5.) Compiler Writing Tools

- Even though initial/~~stee~~ pioneer compilers were written from scratch, later ones were generated ~~based~~ with

some help of human input and pre-written compiles.

~ Eventually tools were built up to create the process of ~~com~~ building a compiler, easier, faster and ~~with~~ more reliable.

~ Some of these tools are mentioned below:

(1.) Parser Generator

~ Takes input as grammatical rules or correctness

~ ~~Prints~~ It give the syntax of the languges to (defines) be designed.

(2.) Scanner Generator

~ Accepts input as regular expressions

~ Spits out a lexical analyser.

(3.) Syntax Directed Translation Engine

~ Receives a parse tree as input

~ The output is an intermediate code generator.

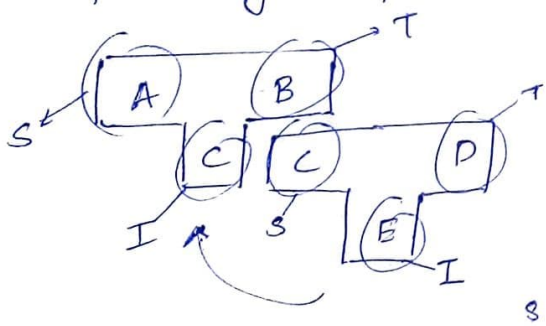
(4.) Data Flow Analysis Engine

~ This is an advanced tools which is being improved daily. ~~es~~

~ Reason is that it helps in the complex process of code optimization.

A 6.) Bootstrapping

- Bootstrapping is used to design a compiler.
- It helps to translate a complex source program to an even more complex program and so on.
- For example the most widespread implementation of Python language is CPython - which is written in C, again in turn C is written in assembly and so on.
- Other implemention of Bootstrapped Python include: Cython, PyPy, Jython, Iron Python etc.
- The process of bootstrapping is show in the following diagram:



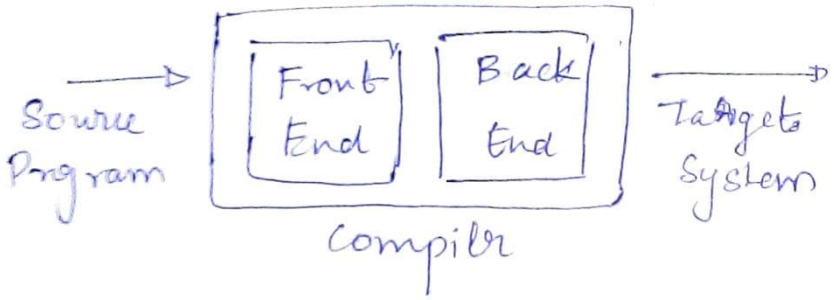
$C_I^{S\,T}$

- A compiler is associated with
  - ① → Source language (S)
  - ② → Target language (T)
  - ③ → Implementation language (I)
- Here "C" is written in "E". & "A" is written in "C" or better termed as 'compiled in'.
- Thus bootstrapping process is also known as compiler compiler.

A.7.) Cross Compilers

- Compilers that are capable of translating the source program into multiple system with different architecture are known as cross-compiles.

- In otherwords cross-compilers are platform agnostic.



- The back end of an ordinary compiler is hard-bound to the system & instruction set of the architecture it is serving.

- In contrast the backend of the cross-compilers can generate code for multiple platforms.

- Today a single language cannot be used to cover architetus like x86, x86_64, arm, and64 etc. Build systems are used.
  (e.g. Yocto framework has a cross compiler within)

A 8.) Given:

alphabet $\Sigma = \{a, b\}$

required : string not ending in 'ab'

∴ The language $L = \{a, b, aa, bb, ba, aaa, bbb, \ldots\}$

All combination of a & b $\Rightarrow (a+b)^*$

now it should not end with 'ab'.

∴ The other possible endings are a, bb, aba

∴ The require regex : $r = (a+b)^*(a|bb|ba)$

A 9.)

Given:

alphabet $\Sigma = \{a, b\}$

required : sub string 'ab' occurs equally

The language $L = \{\epsilon, ab, abab, ababab, \ldots\}$

∴ The require regulare expression is $r = (a \cdot b)^*$

A 10.) The following is a list of relational operators

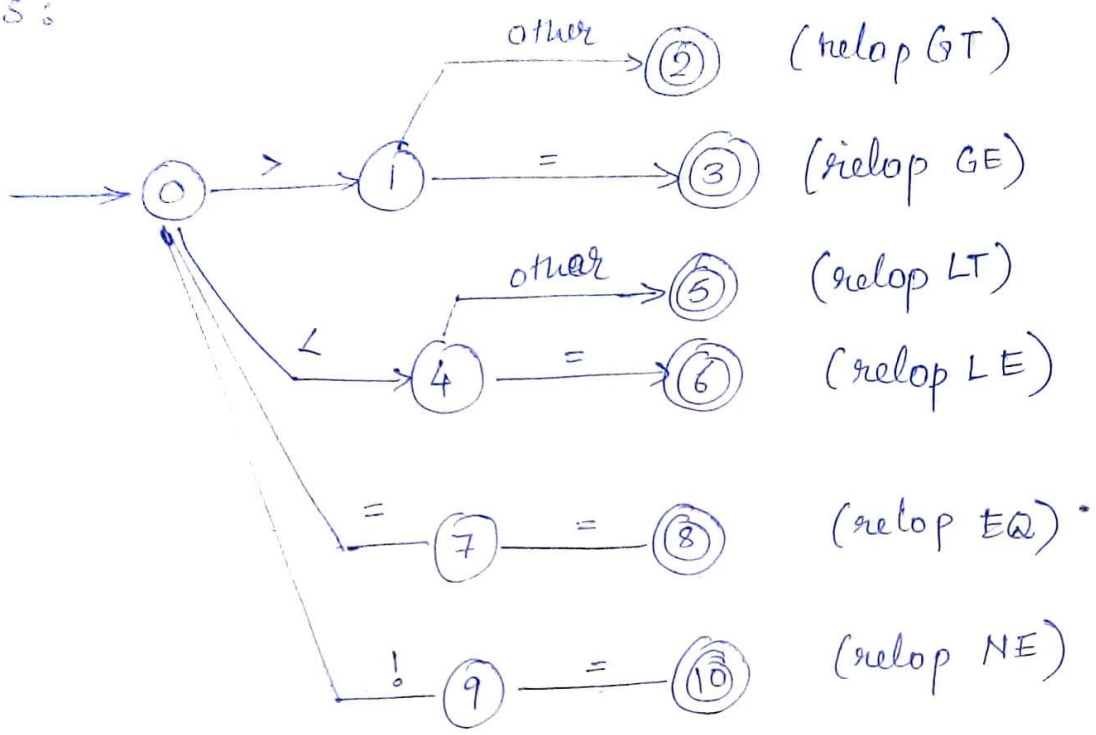
< - less than (LT) <= - less that or equal to (LE)

> - greater that (GT) >= - greater than or equal to (GE)

== Equal to (EQ) != - Not equal to (NE)

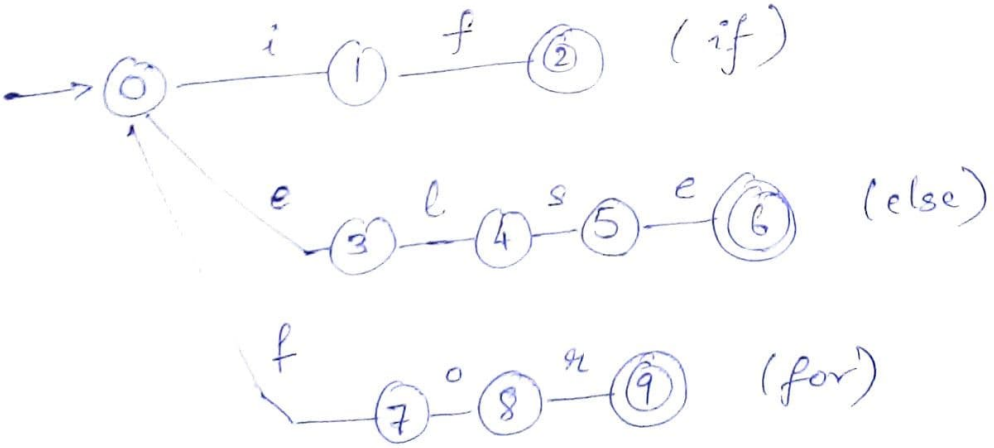
∴ Therefore the transition diagram for the analyser

is:



A 11.) The branching conditionals are given as: if, else, for

- The lexical analyser moves character by character to confirm the lexeme
- The transition diagram would be



- If any other characters apper in between they are then not these conditionals.

A 12.) Different Phase (in design) of a Compiler

~ A compiler is a piece of software that translates source language into target language

~ It has major 6-phases:

@ Lexical Analyser

~ This is the first phase which performs scanning of the source language.

~ This is done to collect tokens (group of characters)

eg:-

position = ~~long~~ x_cap + ycap * 30

~ Here tokens are: 'position', '=', 'x_cap', '+', 'ycap', '*' & 30

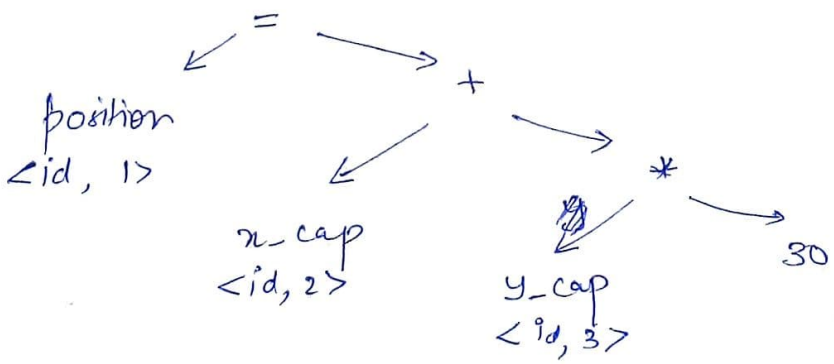
<id, 1> <=> <id, 2> <+> <id, 3> <*> <30>

↳ <token name, attribute value> pair

⑤ Syntax Analyser

~ Parsing is done to ensure that the source code adhers to syntax rules of the source language.

~ It generates a parse tree:

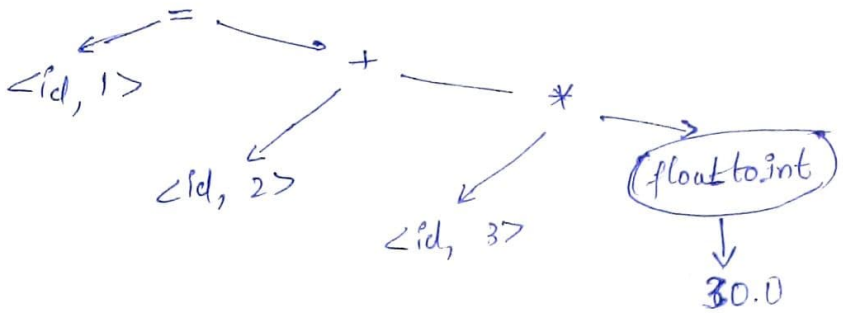


(c) Semantic Analyser

~ This phase focuses upon the data types of subjects in the source code.

~ Wherever required it performs implicit type conversion

eg:-



(d) Intermediate Code Gerator

~ Based on address, and temporary variables intermediate code is generated

eg:-
```
t1 = float_to_int (30.0)
t2 = id3 * t1
t3 = id2 + t1
id1 = t3
```

(e) Code Optimizer

~ Here the intermediate code optimized to retain memory / space efficiency and possibly time.

~ The above code can be optimized to:

t1 = id3 * float_to_int(30.0)

id1 = id2 + t1

~ This has reduced the number of lines of code and also the memory used.

(f) Code Generator

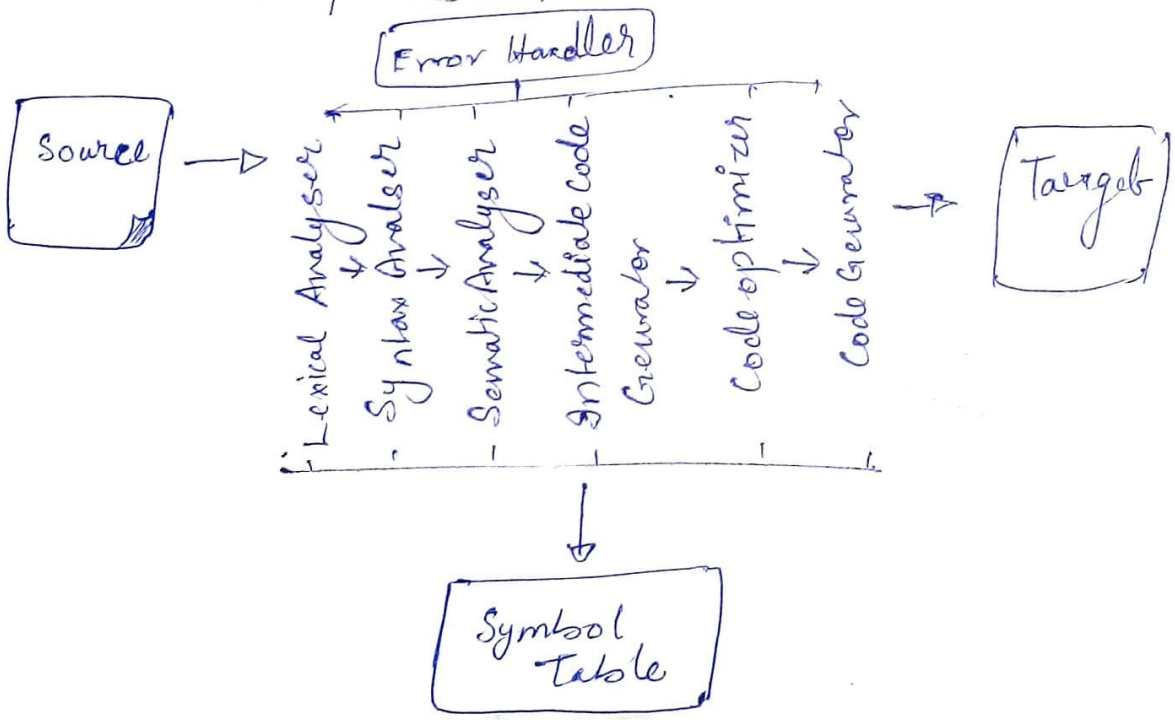
~ This is the final phase which generates the target code.

eg:-

```
LDF   R2, id3
MULF  R2, #60.
LDF   R1, id2
ADDF  R1, R2
STF   id1, R1
```

is a the genrated assembly code.

~ These six phases can shows as:



A 13.) Compiler Compiler

~ It is another term for bootstrapping, which is the process of translation complex languages using

simpler ones.

- This then in turn can be used to ~~create~~ compile even more complex languages.

*g:- C language can be compiled using B (developed at Bell's lab)

Python is compiled using C.

- Thus 'C' is a compiler itself
- But since 'B' compiles C, B is also a compiler
- Therefore 'B' is also called a compiler-compiler.

- Stacking up of compilers to generate high level languages is the process of bootstrapping and it may contain more than one compiler-compiler.