Compiler Construction

CSE

1 Unit

1st half fbt

Compilers :-

→ Compilers are computer programs that translate one language to another.

→ A Compiler takes as its i/p a program written in its Source language and produces an equivalent program written in its Target language.

→ Usually, the Source language is a high-level language like as C or C++, and the target language is object code / machine code for the target machine, that is, code written in the machine inst of the computer on which it is to be executed.

Source pgm → [Compiler] → Target pgm

Programs Related To Compilers

Interpreters :- Line by line

→ An interpreter is a language translator like a Compiler.

→ It differs from a Compiler in that it executes the source program immediately rather than generating object code that is executed after translation is complete.

Scanned with CamScanner

⇒ Compiler is to be preferred if speed of execution is a primary consideration.

Assemblers

⇒ An assembler is a translator for the assembly language of a particular computer.

⇒ Assembly language is a symbolic form of the machine language of the computer and is particularly easy to translate.

Linkers

⇒ Both compilers and assemblers rely on a program called a linker

⇒ linker collects code separately compiled or assembled in diff object files into a file that is directly executable.

⇒ A linker also connects an object program to the code for standard library fns and to resources supplied by the O.S of the computer. i.e memory allocators, I/O devices.

Loaders

⇒ After a compiler, assembler, or linker will put the code that is not yet completely fixed & ready to execute, but whose principal memory

Scanned with CamScanner

References are all made relative to an undetermined starting location that can be anywhere in memory.

⇒ Such code is said to be Relocatable code.

⇒ Loader will resolve all relocatable addresses relative to a given base, or starting address.

Preprocessors

⇒ A preprocessor is a separate program that is called by the compiler before actual translation begins.

⇒ Such a preprocessor can delete comments, include other files and perform Macro substitutions (shorthand description of a repeated seq of text).

Editors

⇒ Compilers usually accepts source program written using any editor that will produce standard file, such as ASCII file.

⇒ Compilers have been bundled together with editors and other programs into an interactive development Envi, or IDE

⇒ Such Editors are called Structure Based ; the programmer may be informed of errors as the pgm is written rather than when it is Compiled.

Debuggers

⇒ A debugger is a program that can be used to determine Execution errors in a compiled program.

⇒ A debugger keeps track of most or all of the Source code inf. such as line no. & names of var & procedures.

⇒ It can also halt the Execution at prespecified location called Breakpts as well as provide inf on what fns have been called & what the current values of var are.

Profilers

⇒ A profiler is a program that called statistics on the behavior of an object pgm during Execution.

Scanned with CamScanner

Project Managers

⇒ Modern s/w projects r usually so large that they are undertaken by groups of programmers rather than a single programmer.

⇒ A project manager can be written in a lang independent way; but when it is bundled together with a compiler, it can maintain info on the specific compiler & linker operat needed to build a complete executable progr

⇒ two popular project manager programs on Unix Systems are

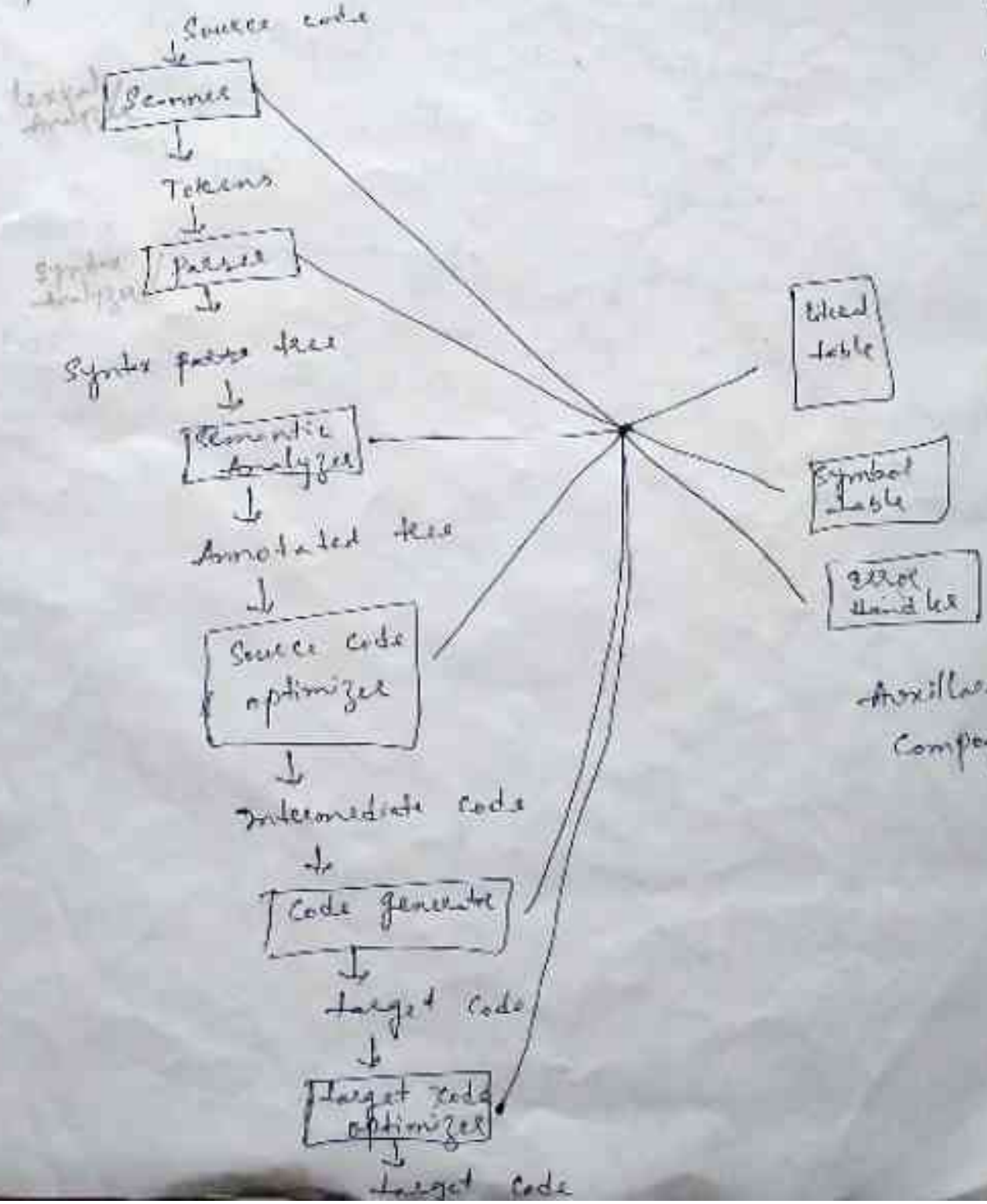
sccs (Source code ctrl sys) &

rcs (revision ctrl sys).

TRANSLATION PROCESS

/ phases of a Compiler
/ Building blocks of compiler

⇒ A Compiler consists internally a no. of steps, or phases, that perform distinct logical operation



Scanned with CamScanner

SCANNER

⇒ This phase of the Compiler does the actual reading of the Source pgm, which is usually in the form of a Stream of char.

⇒ The Scanner performs what is called lexical analysis it collects seq of char into meaningful units called Tokens

Eg:-

a [index] = 4 + 2

a	identifier
[	left bracket
index	id
]	right bracket
=	operator
4	no.
+	operator
2	no.

⇒ A Scanner may perform other operations along with recognization of tokens.

For Eg:- it may Enter identifiers into Symbol table, it may Enter literals into literal table

literal numeric Const e.g. 4
quoted string "hello"

Scanned with CamScanner

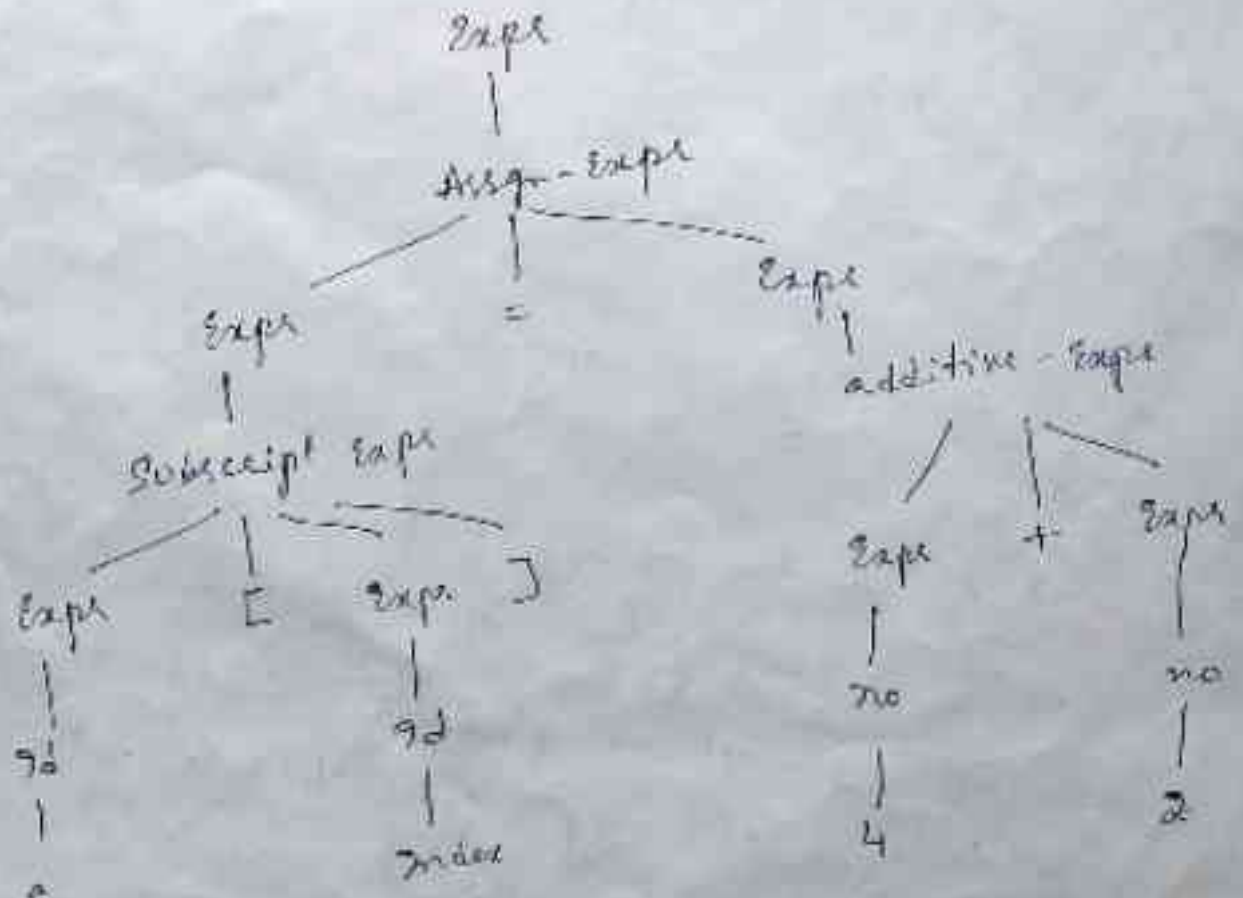
PARSER

⇒ the parser receives the source code in the form of tokens from the scanner & performs Syntax Analysis, which determines the structure of the program.

⇒ this is similar to performing grammatical analysis on a sentence in a natural lang.

⇒ the results of syntax analysis are usually represented as a parse tree or syntax tree

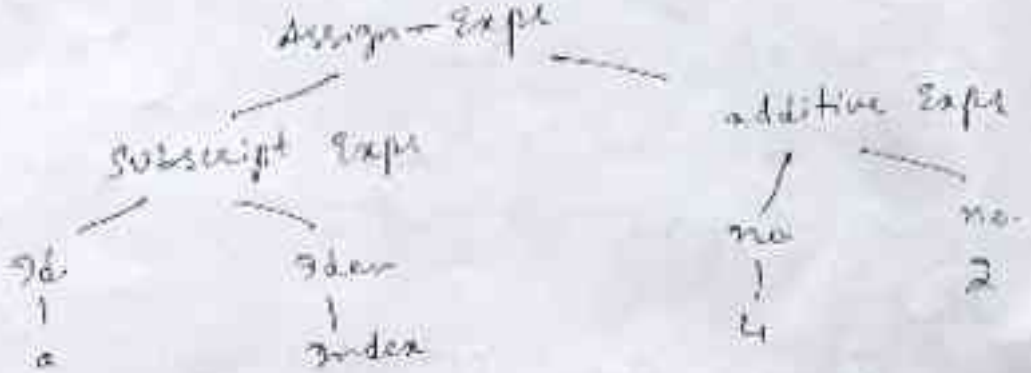
Eg:- a[index] = 4+2



Syntax tree / parse tree

Scanned with CamScanner

⇒ Syntax tree are called sometimes Abstract Syntax Tree, since they represent a further abstraction from parse trees.



Abstract Syntax Tree

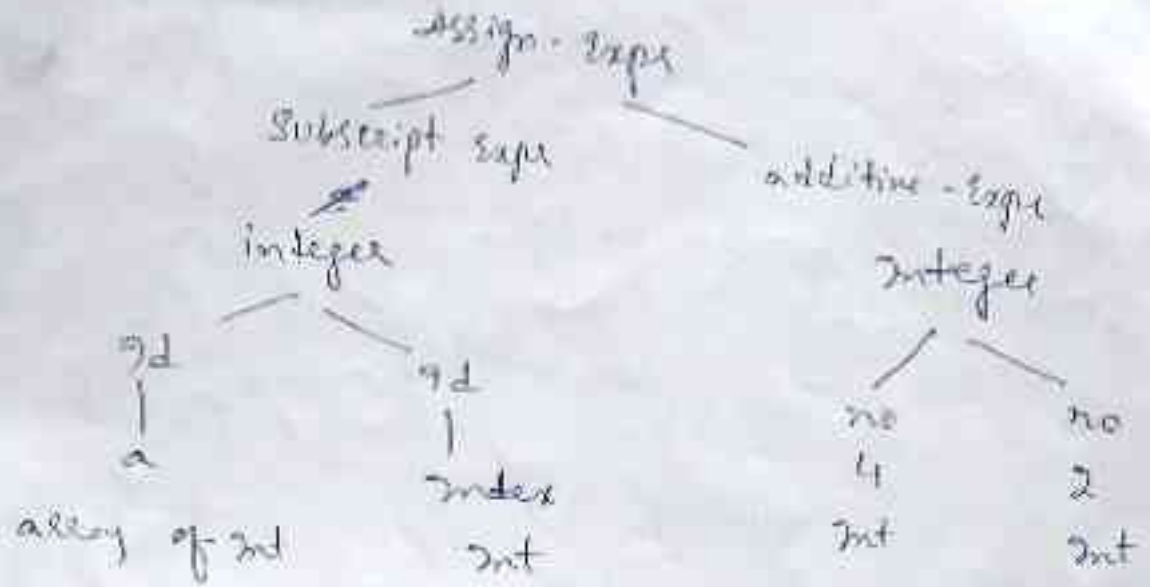
SEMANTIC Analyzer

⇒ the Semantics of a prgm are its "meaning"

⇒ the Semantics of a prgm determine its Runtime behavior, but most PL have features that can be determined prior to execution and yet can be conveniently Expressed as Syntax & analyzed by the parser.

⇒ such features r refferred to as Static Semantics and the analysis of such Semantics is the task of the Semantic Analyzer.

⇒ the Extra pieces of info (dt) computed by the Semantic analyzer is called Attributes



Source Code optimizer

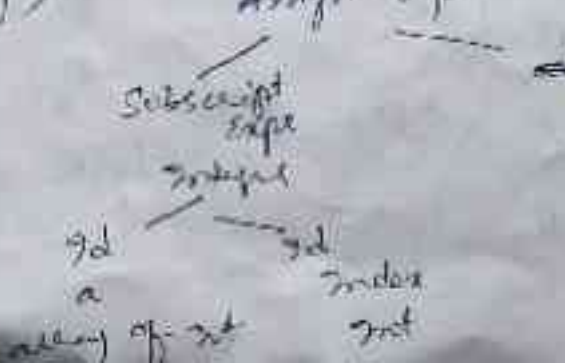
⇒ Compilers often include a no. of code improvement or optimizations steps.

⇒ Individual compilers exhibit a wide variation not only in the kinds of optimizations performed but also in the placement of the optimization phases.

Eg:- Source-level optimization:

expr 4 + 2 can be precomputed by the Compiler to the result 6.

this particular optimizatn is known as (Constant folding).



Scanned with CamScanner

⇒ It is easier to optimize a linearized form of the tree that is closer to assembly code.

⇒ Standard choice is three-address code

```
t = u + 2      ; 1st addr
a[index] = t
```

t - temp var to store intermediate result.

```
t = 6          ; 2nd addr
a[index] = t
```

```
a[index] = 6   ; 3rd addr
```

⇒ Source code optimizer may use 3-addr code by referring to its o/p as Intermediate code.

⇒ Intermediate code refer as Intermediate Repr (IR).

Code Generator

⇒ The code generator takes the intermediate code or IR and generates code for the target machine.

Target Code optimizer

⇒ In this phase, the compiler attempts to improve the target code generated by the code generator.

⇒ Such improvements include choosing addressing modes to improve performance, replacing slow inst by

Scanned with CamScanner

faster ones, and Eliminating redundant or unnecessary operations.

Major Data Structures In A Compiler

principal DS that are needed by the phases as part of their operation and that serve to Communicate information among the phases.

Tokens → Scanner collects char & generate tokens.
Syntax Tree → Each Syntax tree node itself may require diff attributes to be stored in Symbol table
Symbol table
literal table
Intermediate code
temporary files

Symbol table

⇒ This DS keeps info associated with identifiers, functions, Variables, Constants, and data types.

⇒ The Symbol table interacts with almost every phase of the Compiler

⇒ The Scanner, parser, or Semantic analyzer may enter identifiers into the table

⇒ The Semantic analyzer will add data type and other info provided by the Symbol table to make appropriate object code choices.

Scanned with CamScanner

literal table

⇒ It stores Constants & strings used in a program.

⇒ literal table is imp in reducing the size of a program in memory by allowing the reuse of Constants and strings.

⇒ It is also needed by the code generator to Construct Symbolic addresses for literals and for Entering data definations in the target code file.

Intermediate Code

⇒ depending on the kind of intermediate code (three-address code) and the kinds of optimizations performed, this code may be kept as an array of text strings, a temp text file, or as a linked list of structs

Temporary files

⇒ keeping only enough info from earlier parts of the source prgm to enable translation to proceed.

Scanned with CamScanner

Other Views in Compiler Structure

⇒ Compiler writer should be familiar with as many views of Compiler structure as possible, since the structure of the Compiler will have a major impact on its reliability, efficiency, usefulness, and maintainability.

⇒ other Aspects of Compiler structure & indicate how each view applies.

Analysis & Synthesis

⇒ Compiler operations that analyze the source pgm to Compute its properties are classified as the Analysis part of the Compiler,

while the operations involved in producing translated code are called Synthesis part of the Compiler

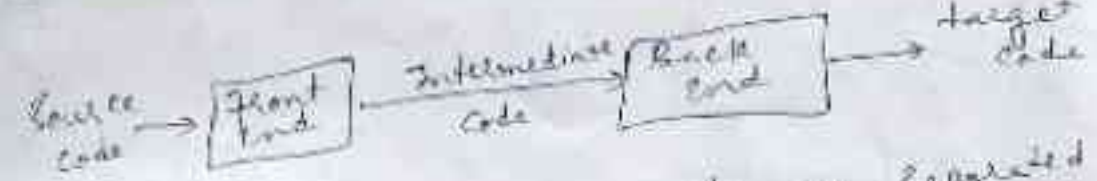
⇒ lexical, syntax, Semantic analysis belong to the Analysis part

while the Code generation is Synthesis.

⇒ optimization steps may involve both analysis and Synthesis.

Scanned with CamScanner

Front End & Back End



⇒ this view regards the compiler as separated into those operations that depend only on source lang (front end) and those operations that depend only on the target lang (back end).

⇒ this model is especially imp for compiler portability

Passes

⇒ A compiler often finds it convenient to process the entire source prgm several times before generating code. These repetitions are referred to as passes.

⇒ depending on the lang, a compiler may be one PASS, in that all phases occur during a single pass. This result in efficient compilation but also in less efficient target code.

Scanned with CamScanner

Language Definitions & Compilers

⇒ PL are usually specified in formal terms and use regular Expr & CFG.

⇒ these descriptions are usually collected into a Language Reference Manual or language Definition.

⇒ the way in which a lang is defined will have a major impact on the techniques that are needed to construct the compiler.

Compiler Options & Interfaces

⇒ Compiler Constructor is the inclusion of mechanisms for interfacing with the OS and for providing options to the user for various purposes.

⇒ Interface mechanisms are the provision of I/p & o/p facilities as well as to the file system of the target machine.

⇒ Both interfacing & options are collectively referred to as Compiler PRAGMATICS.

Error Handling

⇒ Errors can be detected during almost every phase of compilation.

Scanned with CamScanner

⇒ These static (compile-time) errors must be reported by a Compiler.

⇒ Error Handler must contain diff operations, each appropriate for a specific phase and situation.

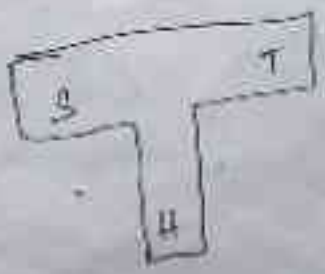
Bootstrapping & Porting

Compiler for lang A written in lang B → [Existing Compiler for lang B] → Running compiler for lang A

⇒ To write the compiler in another lang for which a Compiler already exists.

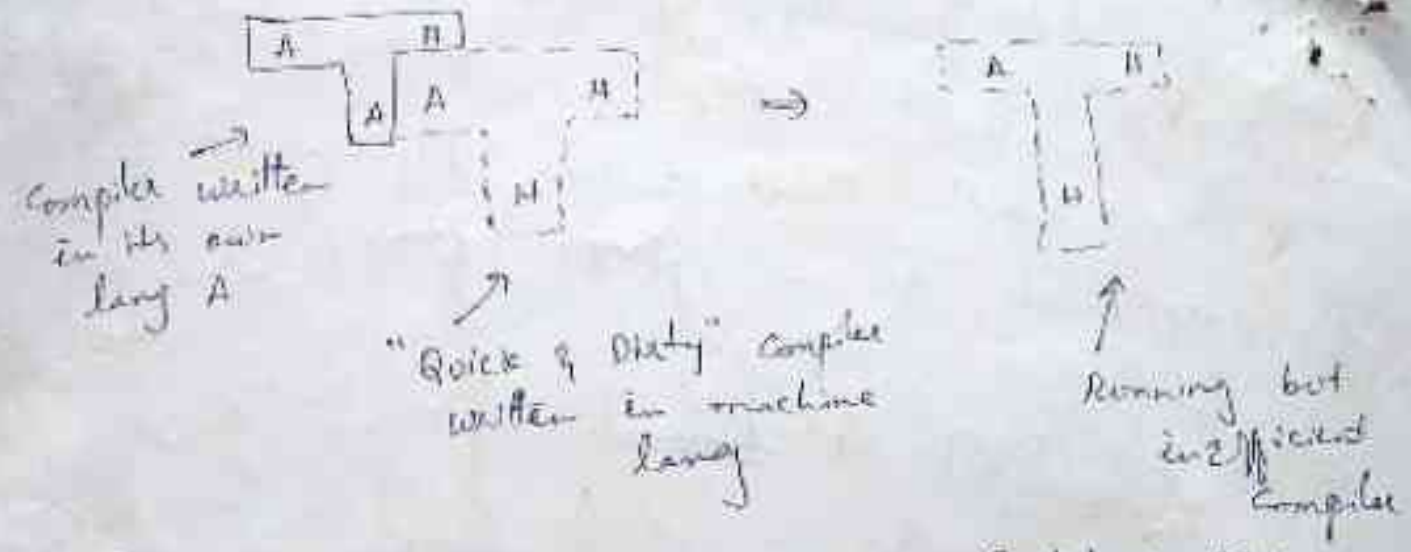
⇒ If the existing compiler already runs on the target machine, then we need only compile the new compiler using the existing compiler to get a running program.

⇒ Cross Compiler is a compiler that generates target code for a diff machine from the one on which it runs.



S - Source
H - Host
T - Target

Scanned with CamScanner

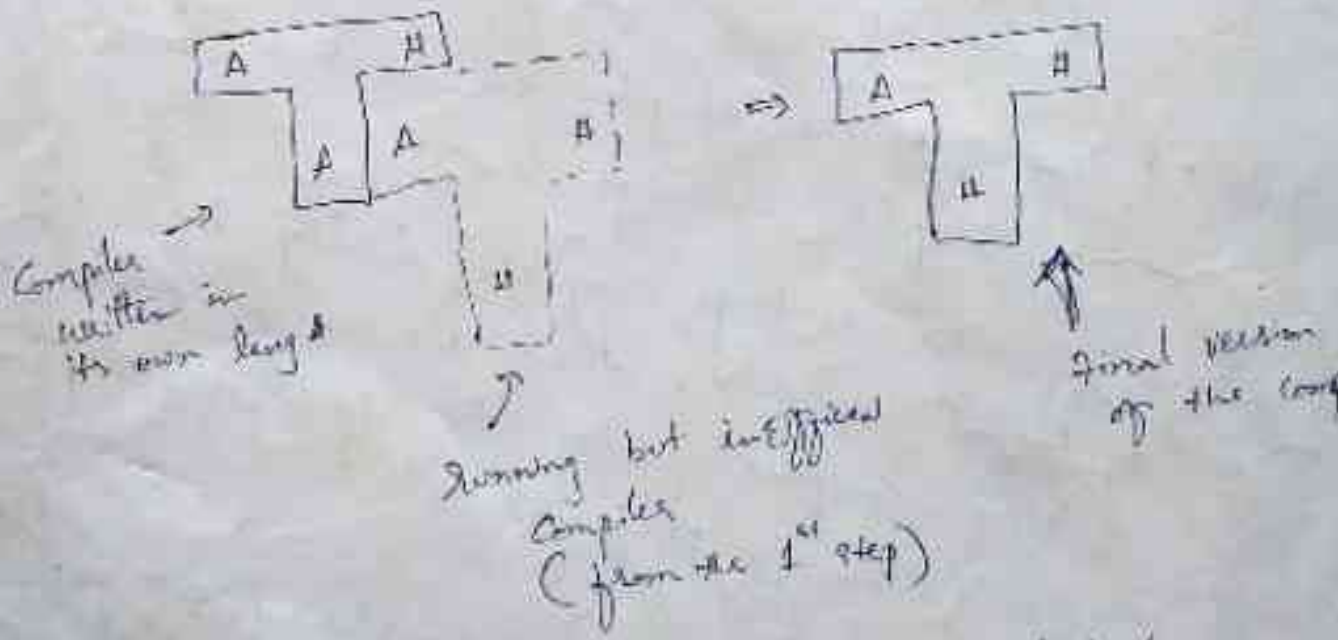


Compiler written in its own lang A

"Quick & Dirty" compiler written in machine lang

Running but inefficient Compiler

1st step in a Bootstrap process



Compiler written in its own lang A

Running but inefficient compiler (from the 1st step)

final version of the compiler

2nd step in Bootstrap process

Porting

porting the Compiler to a new host computer now only requires that the Back End of the Source Code be rewritten to generate Code for the new machine.

Scanned with CamScanner

Translation of an Assignment Stmt

position = initial + rate * 60

↓

[Lexical Analyzer]

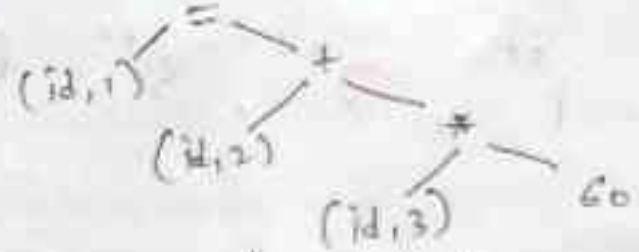
↓

(id,1) = (id,2) (+) (id,3) (*) (60)

↓

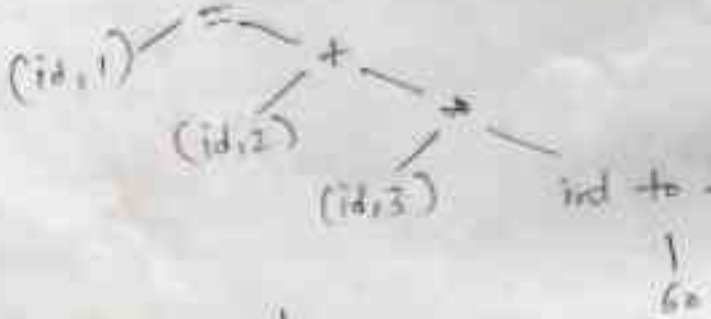
[Syntax Analyzer]

↓



symbol table

[Semantic Analyzer]



↓

[Intermediate Code Generation]

↓

t1 = int to float (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

↓

[Code optimizer]

↓

t1 = id3 * 60.0
id1 = id2 + t1

↓

[Code Generator]

↓

```
LD F   R2, id3
MULT F R2, R2, #60.0
LD F   R1, id2
ADD F  R1, R1, R2
ST F   id1, R1
```

Scanned with CamScanner

1 Unit 2nd half

⇒ The Role of the Lexical Analyzer

→ Input buffering

→ Specification of Tokens

→ Recognization of Tokens

→ The lexical - Analyzer Generator Lex

Scanned with CamScanner

I UNIT

⇒ Input Buffering

- Buffer pairs
- Sentinels

- The task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme
- Single-char operators like -, =, or < could also be the beginning of a two char operator like →, == or <=
- we shall introduce a two-buffer scheme that handles large lookaheads safely.

Buffer pairs

- two buffers that are alternately reloaded.

- Each buffer is of the same size N

Scanned with CamScanner

Two pointers to the i/p are maintained:

1. pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. pointer forward scans ahead until a patter match is found.

⇒ Sentinels

Sentinel is a special char that can't be part of the source pgm, and a natural choice is the char eof.

Scanned with CamScanner

SPECIFICATION OF TOKENS

- Strings & languages
- operations on languages
- Regular Expression
- Regular Definition
- Extensions of R.E

Strings & languages

String over an alphabet is a finite sequence of symbols drawn from that alphabet

-the terms "sentence" and "word" are often used as synonyms for "string".

-the length of a string s, usually written |s|, is the no. of occurences of symbols in s.

Language is any countable set of strings over some fixed alphabet.

Abstract languages like ϕ, the Empty set, or $\{\varepsilon\}$.

Scanned with CamScanner

operations on languages

- operations on languages are Union, Concatenation, and closure.
- Kleene closure of a language L, denoted L^*, is the set of strings u get by concatenating L zero or more times.
- L^0, the "concatenation of L zero times", is defined to be $\{\epsilon\}$
- Finally, the positive closure, denoted L^+, is the same as kleene closure, but w/o the term L^0. one or more times

operation	Defination & Notation
Union of L & M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L & M	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Scanned with CamScanner

Regular Expressions

- R.E often contain unnecessary pairs of parenthesis.
- we may drop certain pairs of parentheses if we adopt certain rules.

(a) Unary operator * has highest precedence & is left Associative.

(b) Concatenation has 2nd highest " " "

(c) | has the lowest precedence and is left associative.

Example. let $\Sigma = \{a, b\}$

1. The R.E a|b denotes the language $\{a, b\}$
2. (a|b)(a|b) denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another R.E for the same lang is aa|ab|ba|bb.
3. a^* denotes the lang consisting of all strings of zero or more a's, that is $\{\epsilon, a, aa, aaa, \ldots\}$

Scanned with CamScanner

4. $(a|b)^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's & b's:

$L = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$

5. $a|a^*b$ denotes the lang $\{a, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

— A lang that can be defined by a R.E is called a Regular Set.

Algebric Laws for R.E

Law	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt$; $(s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is identity for Concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in closure
$r^{**} = r^*$	* is idempotent

Scanned with CamScanner

Regular Definitions

- if Σ is an alphabet of basic symbols, then a Regular Definition is a sequence of definations of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n$$

where

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a R.E over an alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Extensions of R.E

1. One or more Instances:
   the Unary postfix operator + represents the positive closure of a R.E and its language.
   two useful Algebraic laws,

$$r^* = r^+ | \epsilon$$
$$r^+ = r r^* = r^* r$$

relate to kleene closure & positive closure.

Scanned with CamScanner

2. Zero or one instance:

The unary postfix operator ? means "zero or one occurence"

$$\gamma? = \gamma | \epsilon$$
$$L(\gamma?) = L(\gamma) \cup \{\epsilon\}$$

3. character classes

A R.E $a_1 | a_2 | \ldots | a_n$, where a_i's are each symbols of the alphabet, can be replaced by

$[a_1, a_2, a_3 \ldots a_n]$

$[abc] = a|b|c$

$[a-z] = a|b|\ldots|z$

⇒ Recognition Of Tokens

- Transition Diagrams
- Recognition of Reserved words & identifiers

Now we will study how to take patterns for tokens & build a piece of code that examines the i/p string & finds a prefix that is a lexeme matching one of the pattern.

Scanned with CamScanner

Eg:- stmt → if expr then stmt
| if expr then stmt else stmt
| ∈
expr → term relop term
| term
term → id
| number

Grammar for branching statements

- the terminals of the grammar, which are if, then, else, relop, id, and number, are the names of token as far as the lexical analyzer is concerned.

digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?

letter → [A-Za-z]
id → letter (letter | digit)*

if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

Pattern for tokens

Scanned with CamScanner

Transition Diagrams

→ As an intermediate step in the construction of a lexical Analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams".

→ Transition diagrams have a collection of nodes or circles, called states

→ Edges are directed from one state of the transition diagram to another.

→ Each edge is labeled by a symbol or set of symbols.

→ We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Important Conventions about Transition diagrams

(1) Certain states are said to be Accepting, or final states.

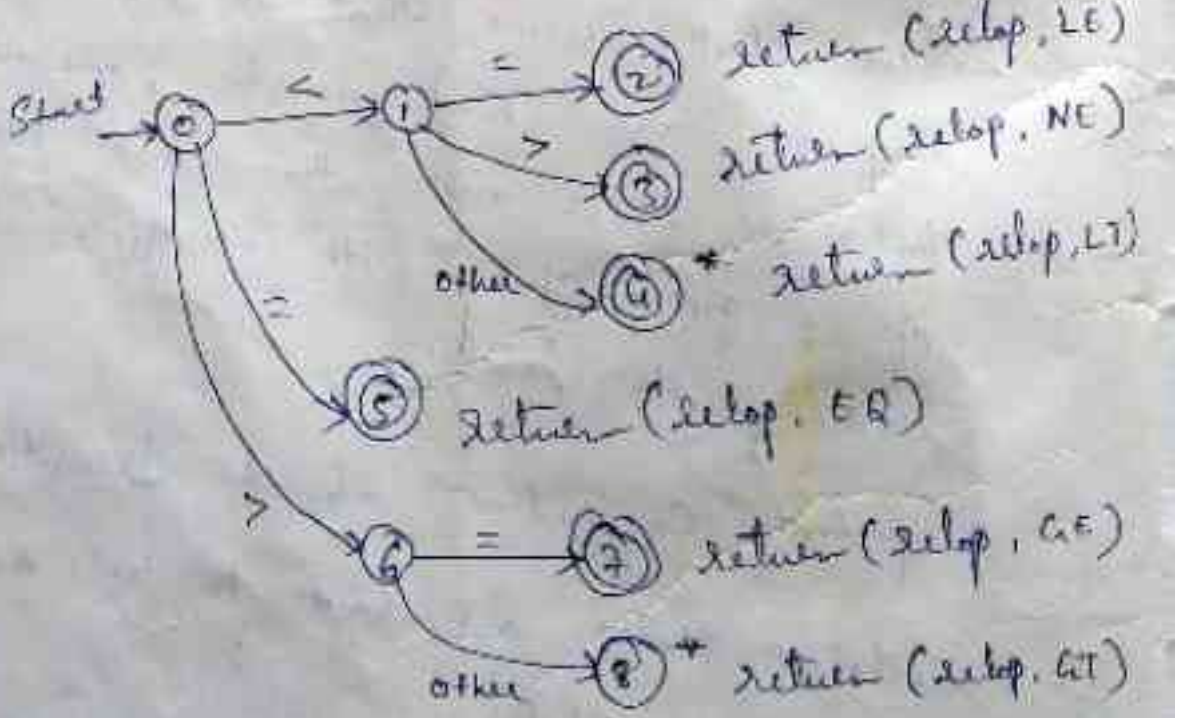
Accepting state is indicated by a double circle.

Scanned with CamScanner

(2) If it is necessary to move the forward pointer one position, then we shall additionally place a * near that Accepting state.

(3) one state is designated the start state or initial state; it is indicated by an edge labeled "start", entering from nowhere. The transition diagram always begins in the start state before any i/p symbols have been read.

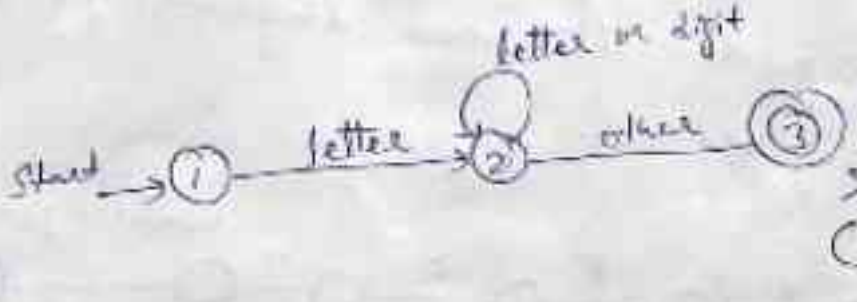
Eg:- Start state 0
Searching for relop <, <>, <=, =, >=



Transition Diagram for relop

Recognition of Reserved words & Identifiers

→ Recognizing keywords & identifiers presents a prob. usually, keywords like if or then are reserved, so they are not identifiers even though they look like identifiers.



Transition Diagram for id's & keywords

There are two ways that we can handle Reserved words that look like identifiers:

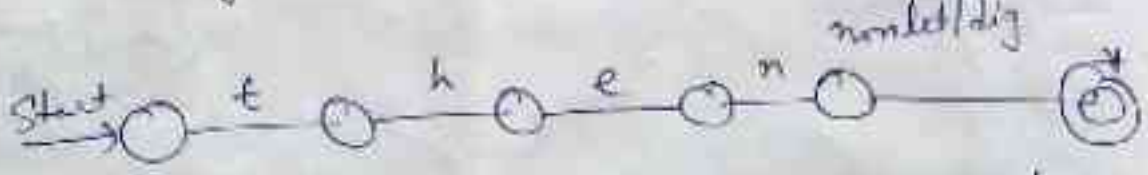
(1). Install reserved words in the symbol table initially.

- A field of the symbol-table entry indicates that these strings are never ordinary id, and tells which token they represent.
- when we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found.

Scanned with CamScanner

- the function getToken examines the Symbol table Entry for the lexeme found, and returns whatever token name the Symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.

(2) create seperate transition diagrams for each keyword



⇒THE LEXICAL-ANALYZER GENERATOR LEX

→ Lex is a tool that allows one to specify a lexical analyzer by specifying R.E to describe patterns for tokens.

→ I/p notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex Compiler

→ lex compiler transforms the I/p patterns into a transition diagram and generate code, in a file called lex.yy.c

Scanned with CamScanner

- Use of lex
- Structure of lex programs
- Conflict Resolution in lex

Use of lex

Creating a lexical Analyzer with lex

Structure of lex programs

→ A lex pgm has the following form:

```
%{ declaration
%}
%%
rules
%%
C pgm / auxiliary pgms
```

Declaration section includes declarations of variables, constant and regular definations.

Scanned with CamScanner

Transition Rules

pattern & actions

- Each pattern is a R.E, which may use the regular definations of the declaration section.
- The actions are fragments of code, typically written in C.

C functions / Auxiliary fns

- this section holds whatever additional fns are used in the actions.
- these fns can be compiled separately and loaded with the lexical analyzer.

Conflict Resolution in lex

two rules that lex uses to decide on the proper lexeme to select, when several prefixes of the i/p match one or more patterns:

(1) Always prefer a longer prefix to a shorter prefix.
   <, <= => prefer <=

(2) If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the lex prgm.

a then b prefer then
↓ ↓ ↓
id KW id

lex prgm