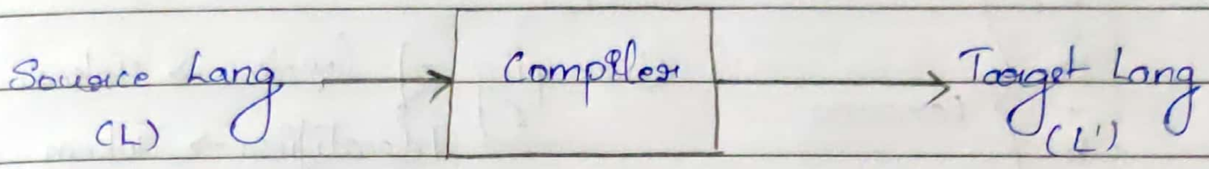
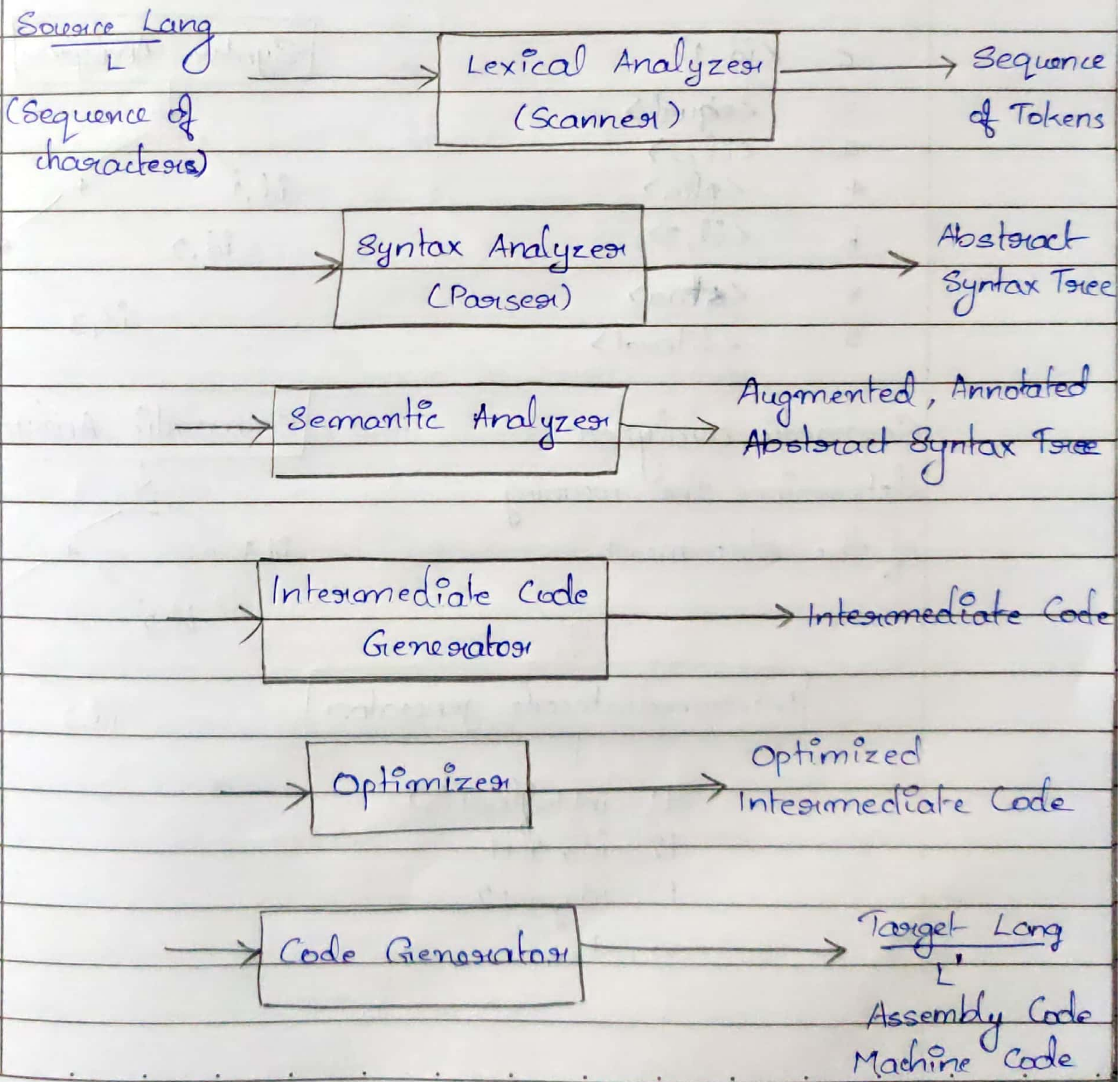
URBAN EDGE 14 / 1 / 20

Module 1

Compiler





Scanned by CamScanner

14/1/20

eg; c=a+b*5;

Source prog contains a set of lexems.

c
=
a
+
b
*
5

} Lexems

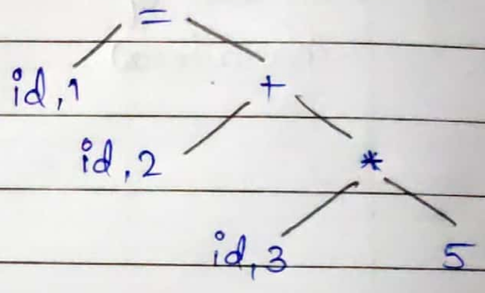
Grouping of lexems → Token

Identifier → Token

Each lexem can also be considered as a token

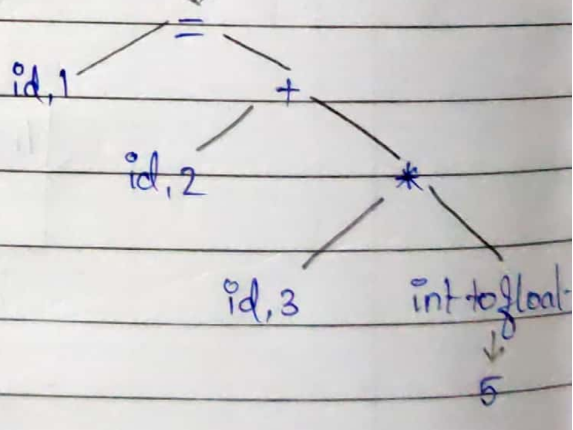
c	<id, 1>
=	<equal>
a	<id, 2>
+	<plus>
b	<id, 3>
*	<star>
5	<literal>

Syntax Analyzer



Semantic analyzer determines the meaning of the statement

Semantic Analyzer



↓

Intermediatecode generator

↓

$t1 = inttofloat(5)$

$t2 = id_x * t1$

$t3 = id_y + t2$

$id_1 = t_3$

↓

Code Optimizer

↓

```
t1 = id5 * 5
id1 = id2 * t1
```

Code Generator

↓

```
LDF R2, id3
MULF R2, #5,0
LDF R1, id2
ADDF R1, R2
STF id1, R1
```

- The Scanner

→ Reads characters from the source prog

→ Groups characters into lexems

- The Parser

→ Groups tokens into "grammatical phrases"

- Intermediate Code Generator
- Optimizer
- Code Generator
- Symbol Tables

→ Keep track of names declared in the program.

Compiler Modularity

- Front End → Lexical analyzer, Syntax analyzer, Semantic analyzer, Intermediate code generator.

- Back End → Optimizer, Code generator
- To add a new language, we modify front end
- " " " " machine, " " back "

16/1/20

Lexical Analysis

The i/p

- Read string i/p

The o/p

- A series of tokens

Free Form vs Fixed Form

- Free form langs (C, C++)

→ White space doesn't matter. Ignore tabs, spaces, new lines, carriage returns

→ Only ordering of tokens is important

- Fixed form langs (Pascal, Cobol)

→ Layout is critical. Fortran, label in cols 1-6, Lexical analyzer must know about layout to find tokens.

Approaches to Implement a Lexical Analyzer

(i) Simple Approach

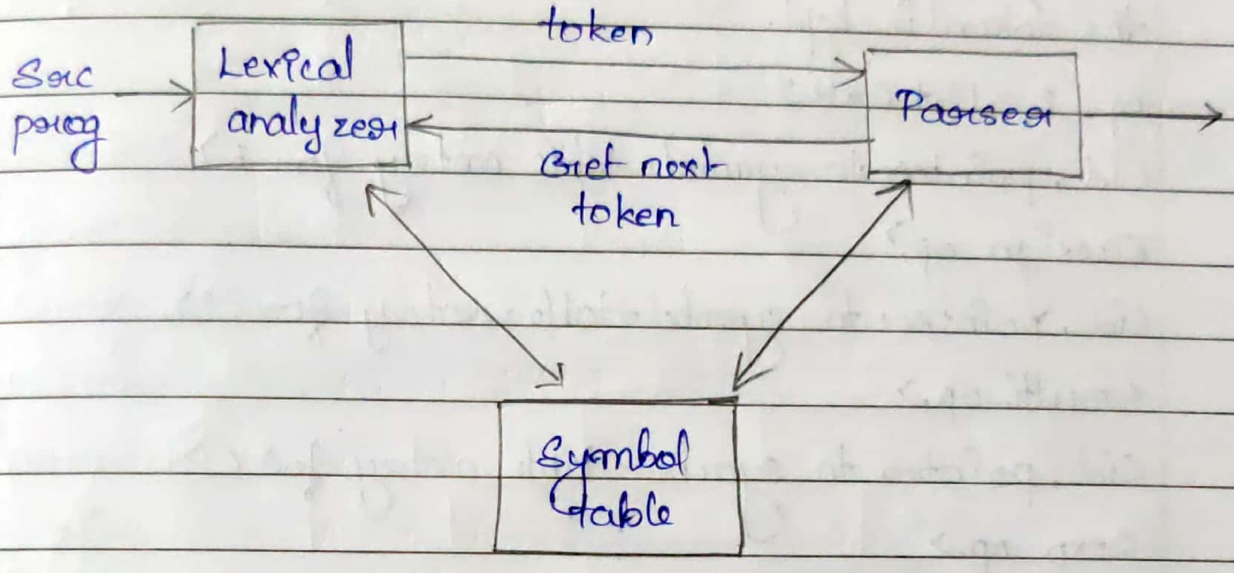
Construct a diagram (FA) that illustrates the structure of the tokens of the source lang.

(ii) Pattern-Directed Programming Approach

- Pattern matching technique
- Specify & design program that execute act^ns triggered by patterns in strings
- We use lex tool to construct FA

Role of Lexical Analyzer

Interactn of Lexical Analyzer with Parser



Processes in Lexical Analyzers

- Scanning
- correlating error messages
- Lexical analysis

Terms

- Token

→ Types of words in src prog

→ Keywords, operators, identifiers, etc

- Lexeme

→ Actual words in src prog

- Pattern

→ A rule describing the set of lexemes that can represent a particular token in src prog (Rule used to define tokens)

→ Relatn {<, <=, >, >=, ==, <>}

16 / 1 / 20

Attributes for Tokens

- A pointer to the symbol-table entry in which the info about the token is kept.

eg: E = M * C ** 2

<id, pointer to symbol-table entry for E>

<assign_op,>

<id, pointer to symbol-table entry for M>

<multi_op,>

<id, pointer to symbol-table entry for C>

<exp_op,>

<num, integer value 2>

Lexical Errors

- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character.
- Transposing 2 adjacent characters (such as, fi => if)
- Pre-scanning

Input Buffering

- 2-buffer i/p scheme to look ahead on the i/p & identify tokens:

(i) Buffer pairs

(ii) Sentinels (Guards)

- Buffer pairs

→ Entire buffer area divided into 2 areas

→ 2 pointers: lexeme beginning & lexeme pointer

16/1/20

- Sentinels (Guards) :-

→ sentinel → special character added at the end of each token.

Specificat^n of Tokens

① Write reg. exp. for the following tokens in C language

(i) identifier

(ii) funct^n name

(iii) arithmetic operators

(iv) numeric literals

(v) character "

(vi) string "

ans (i) [_ + a-z + A-Z] [a-z + A-Z + 0-9 + _]*

(ii) Same as identifiers. No keywords.

(iii) [+ | - | / | * | %]

(iv) [0-9]* [0-9]* | [0-9][0-9]* [.] [0-9][0-9]*

(v) ['] [a-z | A-Z | 0-9 | symbol] [']

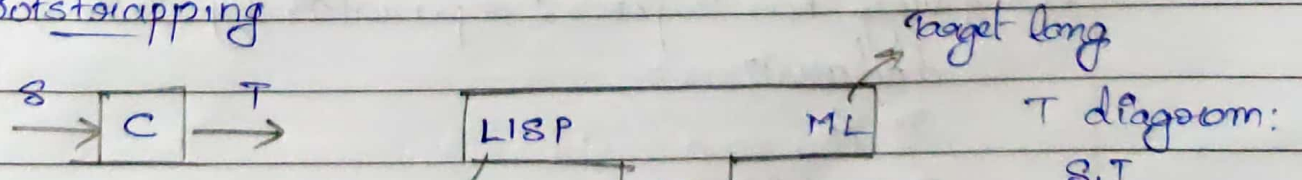
(vi) ["] [a-z | A-Z | 0-9 | symbol]* ["]

Recognit^n of Tokens

identifier → [a-z | A-Z][a-z | A-Z | 0-9]*

keywords → [int | float | char | for | while | ..]

17/1/20 Bootstrapping



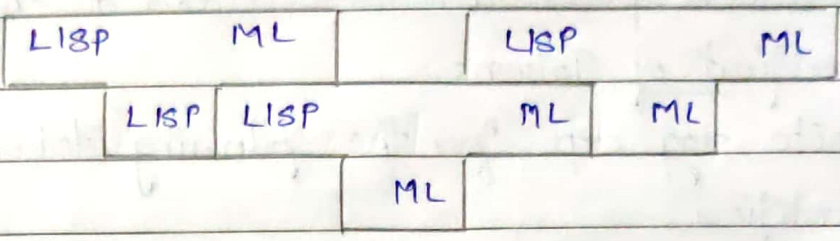
Source lang ↓ (lisp fus)

lang in which compiler is designed

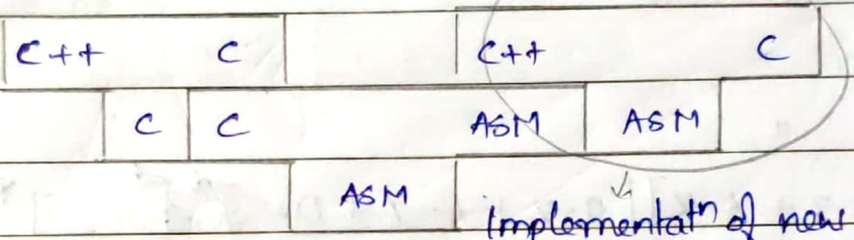
Process of making a new/more complex compiler by using existing compilers

17/1/20

For more complex langs,

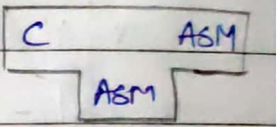


When target lang is another lang having its own compiler.
Design a compiler for C++ using C in which target lang is C.



Implementatⁿ of new compiler

Existing C compiler.



Compiler Writing Tools

Separate/Diff tools are used for designing each phase of a compiler

(i) Lexical Analysis - LEX
(ii) Syntax - YACC
(iii) Intermediate Code Generatⁿ
(iv) Code Optimizatⁿ

No seperate tool for semantic analysis. It is done by the parser itself

No separate tool for code generatⁿ: it is machine dependent

LEX prog has 3 parts; separated by %%

declaratⁿ
%%
Rule definitⁿ → has reg. exp
%%
functⁿs → yylex() → creates FA based on the reg exp given in rule definitⁿ.
user defined functⁿs

YACC - Yet Another Compiler Compiler

YACC prog:- declaratn

%%

Rule definitn

%%

functns → yyparse() → construct a parse tree based on the CFG given in rule definitn.

(i) Construct a reg exp

(ii) Design an ε-NFA

(iii) Convert ε-NFA to NFA

(iv) " NFA to DFA

(v) Minimize the states in the DFA

① Design a compiler which accepts the lang. L & produces L'. Lang L is used for processing arithmetic exp.s only. Recognize keywords if used & generate error messages.

1st stage - construct a lexical analyzer.

→ uses regexp.

Various /Valid tokens in this lang { identifiers,
keywords,
arithmetic operators (+, -, *, /, %)
assign (=)
semicolon (;)
const (0-9)*

Letter → a-z | A-Z

digit → 0-9

identifier → (letter)(letter | digit)*

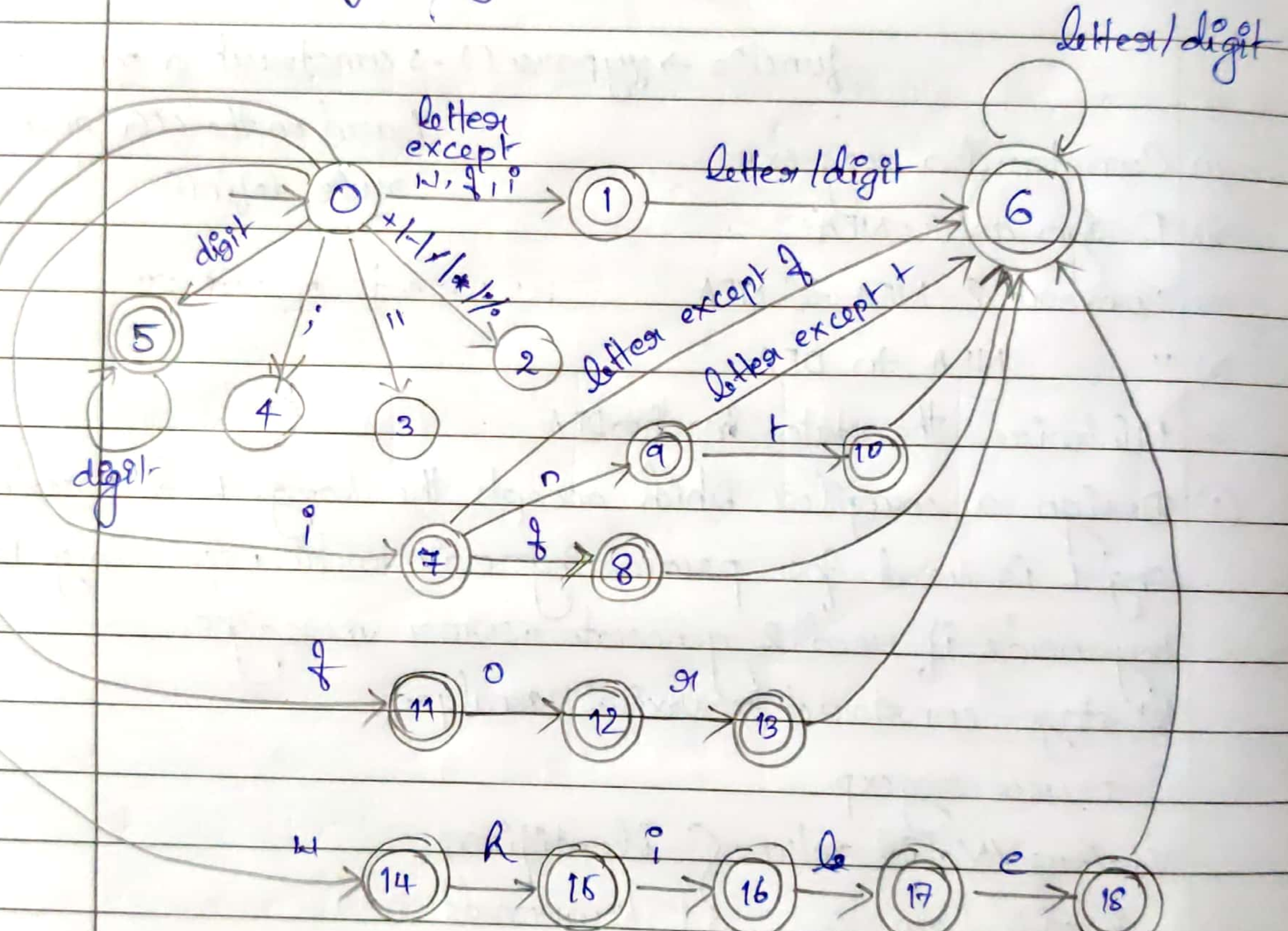
keywords → if | int | for | while

arithmetic operators → + | - | * | / | %

assign → =

semicolon → ;

const → (digit)(digit)* or [(digit)$^+$]



This prog code is generated by LEX in C lang

```
token getnexttoken()
{  while(1)
   {  switch(state)
      {  case 0: c=nextchar();
                 if (c== blank || c== tab || ...)
                 {  state=0;
                    lexeme_beginning ++;
```

17/1/20

```
        }
        else if (c== letter except i,f,w)
            state = 1
        else if(c==w) state = 14;
        else if (c==i) state = 7;
case 1: if it is not end of token, read next char
        else return (id, ptr, value);
    }
case 18: if it is not end of token, c=getnextchar()
        else        → returns to syntax analyzer
          return (keyword, while);
```