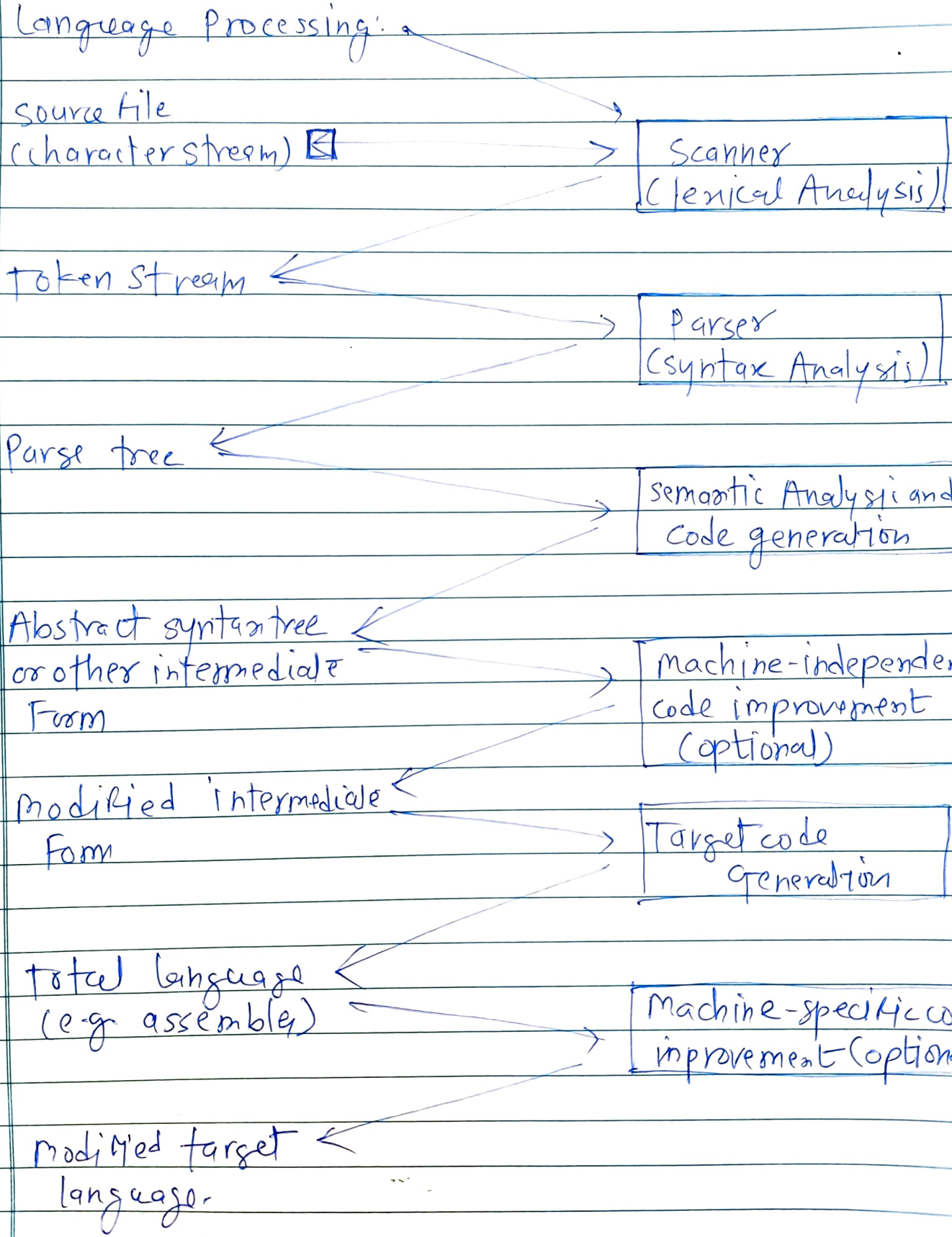
COLLEGE OF ENGINEERING, PUNE

Wellesly Road, Shivajinagar, Pune- 411 005

Anuj Mohite
Comp(SP) TG - Div 2
142103002
Assignment.

Language Processing:



Lexical Analysis :→ Scanner produces a stream of tokens from the input source.

Lex / Flex :→ lex is a scanner generation generator. input is a set of regular expression and associated actions (written in C). output is a table-driven scanner (lex.yy.c)
Flex: an open source implementation of the original UNIX lex utility.

lex input -

```
FIRST PART
%%
pattern    action
...
%%
THIRD PART
```

e.g.

```
filename: ex1.l
%%
"hello world"   printf("Good Bye\n");
...
%%
```

Prints "Good Bye" anytime the string "hello world" is encountered.

Does nothing for any other character

Using lex:

```
lex ex1.l
cc lex.yy.c -ll
./a.out
→ hello world
  Good Bye!
```

process the lex file to generate a scanner (gets saved as lex.yy.c)

compile the scanner & grab main() from the lex lib (-ll option)

Run the scanner taking input from standard input

lex pattern examples:

i) abc → match the string "abc"
ii) [a-z, A-Z] → Match any lower or upper letter
iii) dog.*cat → match any string starting with dog and ending with cat.
iv) (ab)+ → Match one or more accourences of "ab" concatenated.
v) [^a-z]+ → matches any string of one or more characters that donot include lower case a-z.
vi) [+-]?[0-9]+ → Matches any string of one or more digits with an optional prefix of + or -.

yacc & lex used together:

- lex: Semantic Analysis:
  - splits the input file into tokens.

yacc: yet another compiler compiler.
- parse and does semantics processing on the stream of token produced by lex.

bison: GNU parser. parse upward compatibility with yacc.

- lex/yacc

source code → a = b + c + d

↓

lexical Analyzer ← lex ← patterns

↓

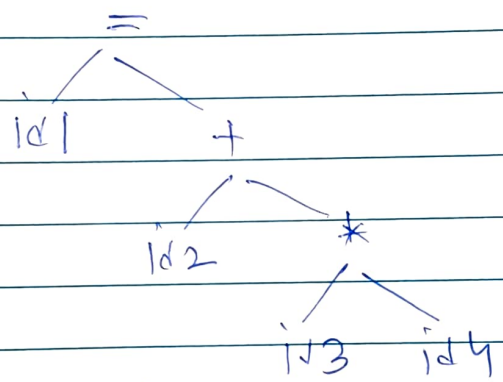
tokens → id1 = id2 + id3 + id4

↓

Syntax Analyzer ← yacc ← grammer

↓

syntax tree



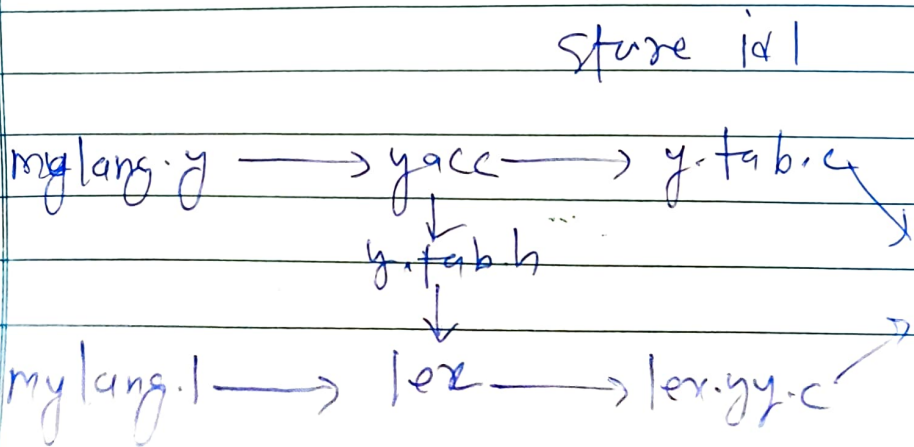
↓

Code Generator

↓

Generated Code

load id3
mov id4
add id2
store id1



Yacc - First part
First part of yacc specification includes:
- c declarations enclosed in %{ %}
- yacc definitions
  - %start
  - %token
  - %union
  - %type.

- The middle section represents a grammar - a set of productions. the left-hand side of a production is followed by a colon. and a right hand side.
- multiple right-hand sides may follow separated by a '|'.
- Actions associated with a value are entered in braces.

Yacc productions.
$1, $2, ... $n can be refer to the values associated with symbols.
$$ refer to the value of the left
every symbol have a value associated with it (including token & non-terminals)
Default action
  $$ = $1

e.g.
  Statement: identifier '+' identifier
    { $$ = $1 + $3; }

contain valid c code
```
% include "myscanner.h"
extern int yylex();
extern int yylineno();
extern char* yytext;
char* name[] = {NULL, "dbtype", "db_name", "db-table-prefix",
                "db-port"};

int main(){
int ntoken, vtoken;
 ntoken = yylex();
while(ntoken){
printf("%d\n", ntoken);
if (yylex() != COLON)
  return 1;
vtoken = yylex();
switch(ntoken){
 case TYPE:
 case NAME:
 case TABLE_PREFIX:
    if (vtoken != IDENTIFIER){
      return 1;}
  printf("%s is set to %s",
    name[ntoken], yytext);
  break;
 case PORT:
   if[vtoken != INTEGER){
     return 1;}
  Printf("\n%s is set to %s");
  name[token], yytext);
   break;
 default: break;
ntoken = yylex(); return 0;
```

myscanner.h
```
% define TYPE 1
% define NAME 2
% define TABLE_PREFIX 3
% define PORT 4
% define COLON 5
% define IDENTIFIER 5
% define INTEGER 6
```

myscanner.l
```
%{ % include "myscanner.h" %}
%%
:return COLON;
"dbtype" return TYPE;
"dbname" return NAME;
"db-table-prefix" return TABLE_PREFIX;
[a-z, A-Z][a-z, A-Z, 0-9] return identifier;
[1-9][0-9]* return INTEGER;
[\t\n] ;
. printf("unexpected character")
%%
int yywrap(void)
{
  return 1;
}
```