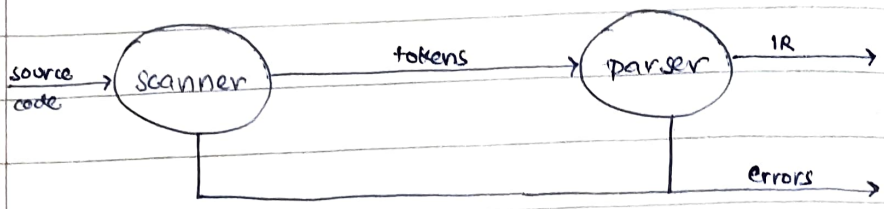
LEXICAL ANALYSIS



SCANNER

maps character stream to words - the basic unit of syntax

produces pairs → word + part-of-speech.
(token) (lexeme) (token type)

x += x + y; ⟨id, x⟩ = ⟨id, x⟩ + ⟨id, y⟩

PARSER

checks if the input stream of tokens constitute a syntactically valid program of the language.

if the input program is syntactically correct
↳ output a concrete representation of program (AST).

formally grammar G = (S, N, T, P)

- S: start symbol
- N: set of non-terminal symbols
- T: set of terminal symbols or words
- P: set of production rules or rewrite rules.

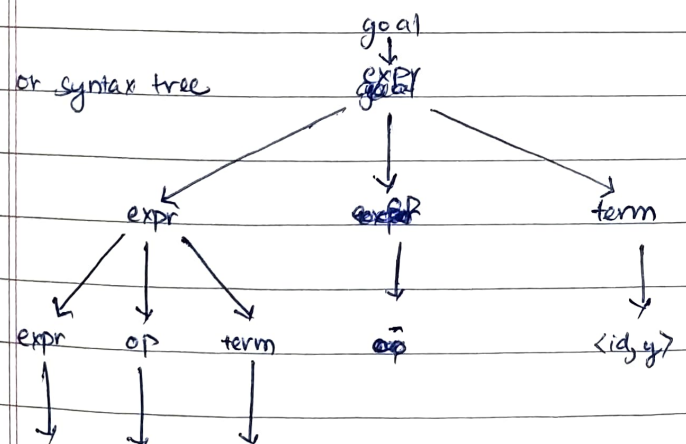
P
- Goal → Expr
- Expr → Expr Op Term | Term
- Term → Number | Id
- Op → + | -

S = Goal (start symbol)
N = { Goal, Expr, Term, op, ... }
T = { number, id, +, - }

ex: given a stream of tokens (read terminals) and the syntax specification in the form of a CFG, how can the parser check the syntactic correctness of the source code?

PARSE TREES

or syntax tree



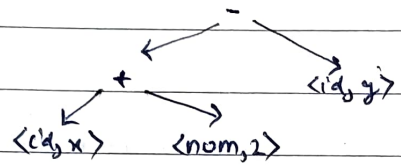
a parse tree contains lot of derivation information not needed by the compiler passes down the line.

ABSTRACT SYNTAX TREES (AST)

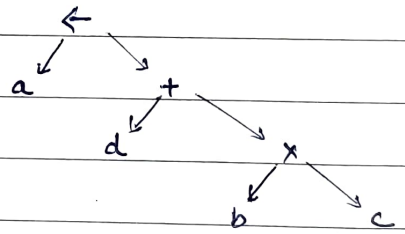
Compilers often use an abstract syntax tree instead of a parse tree. It summarizes grammatical structure, without including detail about the derivation.

$x + 2 - y$

- much more concise
- one kind of IR



$a \leftarrow b \times c + d$



LEXICAL TOKENS

goal → expr

expr → expr op term | term

term → number | id

op → + | -

id → alpha id | alpha

alpha → a | b | c | ... z

number → number digit | digit

digit → 0 | 1 | 2 ... 9

goal → expr
expr → expr op term | term
term → number | id
(+, - are possible ops)

goal → expr
expr → expr + term
| expr - term | term
term → number | id

a token is a fondamental (atomic) syntactic unit in a language. "it depends upon how the grammar for the language is defined though.".

MICRO-SYNTAX AND MACRO SYNTAX

micro syntax:
the rules that govern the lexical structure of a language.
- specified using regular expressions
- implemented using finite state machines

macro syntax:
the grammar of the language.
- specified using CFGs
- implemented using push down automata

q why can't we encode the micro syntax of the language into the grammar for the macro syntax?

dont lift a needle with a crane?

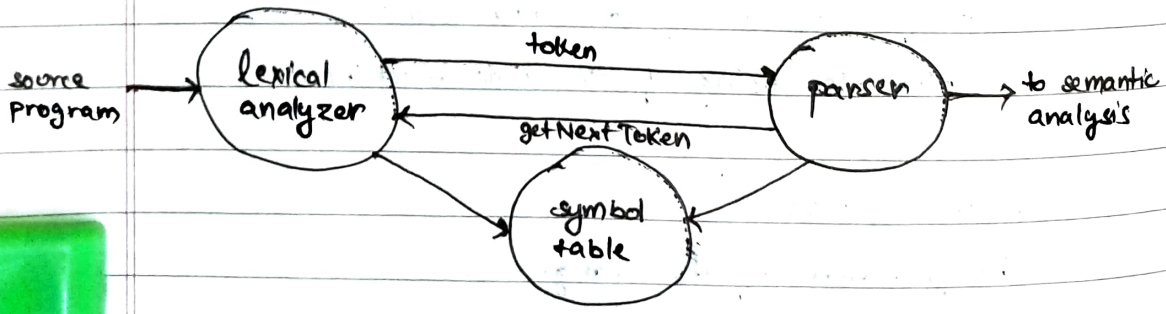
Separation of micro-syntax gives a clean grammar for the core programming language constructs.

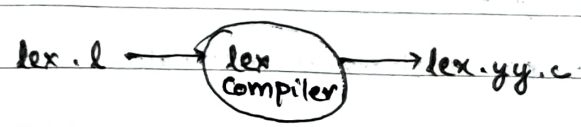
grammar need not keep track of nitty gritty details which are taken care of cleanly and efficiently in lexical analyzer.

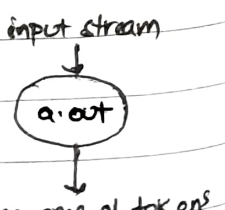
every grammar rule dropped shrinks the size of parser.

parsing is more harder and slower than lexical analysis.

LEXICAL ANALYSIS AND PARSER







STRUCTURE OF A LEX PROGRAM

declarations

%%

translation rules pattern { action }

%%

auxiliary functions

ex /* regular definitions */

delim	[\t\n]
ws	{delim}+
letter	[A-Za-z]
id	{letter} ({letter}|{digit})*
digit	[0-9]
number	{digit}+ (\.{digit}+)? (E [+-]? {digit}+)?

%%

{ws}	{ /* no action and no return */ }
if	{ return IF; }
then	{ return THEN; }
else	{ return ELSE; }
{id}	{ yylval = (int) installID(); return ID; }
{number}	{ yylval = (int) installNum(); return NUMBER; }
"<"	{ yylval = LT; return RELOP; }
"<="	{ yylval = LE; return RELOP; }
"="	{ yylval = EQ; return RELOP; }
"<>"	{ yylval = NEQ; return RELOP; }
">"	{ yylval = GT; return RELOP; }
">="	{ yylval = GE; return RELOP; }

RULES

① return longest matching lexeme (token).
② token types are prioritized according to their occurrence in the lex file.

ⓐ always prefer a longer prefix to a shorter prefix.
ⓑ if the longest possible prefix matches two or more patterns, prefer the pattern listed first in the flex program.

LEXICAL ANALYZER DESIGN

Q. given an RE R, build a C function which checks whether an input string is described by the RE R.

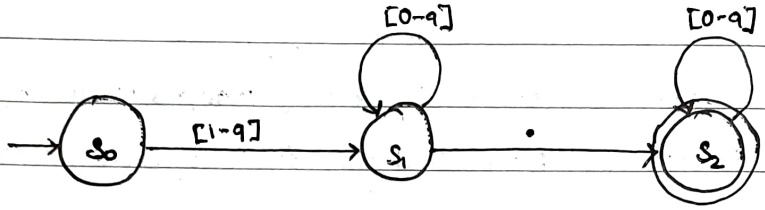
1. convert the RE into an NFA.
2. convert NFA to DFA.
3. minimize the DFA.
4. translate the DFA into an equivalent program function.

time complexity: $O(t)$

t - input string length.

ENCODING A DFA INTO A PROGRAM FUNCTION

eg FLOAT-LITERAL [1-9][0-9]*.[0-9]*



	.	0	1	2	3	4	5	6	7	8	9
S_0	S_e	S_e	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1
S_1	S_2	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1
S_2	S_e	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2
S_e	S_e	S_e	S_e	S_e	S_e	S_e	S_e	S_e	S_e	S_e	S_e

note:
we can compress the table by identifying common columns.

TABLE DRIVEN APPROACH FOR ENCODING DFAS

```
IsFloatLiteral (String str)
Begin
  State = S0
  for i = 1 to str.length
  do
    state = nextState (state, str[i])
    if (state = Se) then return reject
  end for
  if (state ∈ F)
```

then return accept
else return reject
end

① only the transition function, table changes used by the next state function changes to recognize another token.

② coell the set of final states F also changes.

DIRECT CODING APPROACH

```
IsFloatLiteral (String str)
Begin
  i=1
s0: if (i> str.length) then return reject
    if ('1'≤ str[i] ≤'9') then goto s1
    return reject
s1: i=i+1
    if (i> str.length) then return reject
    if ('0'≤ str[i] ≤'9') then goto s1
    if (str[i] = '.') then goto s2
      else return reject
s2: i=i+1
    if (i> str.length) then return accept
    if ('0'≤ str[i] ≤'9') then goto s2
      else return reject
End
```

TABLE DRIVEN VS DIRECT CODED APPROACH

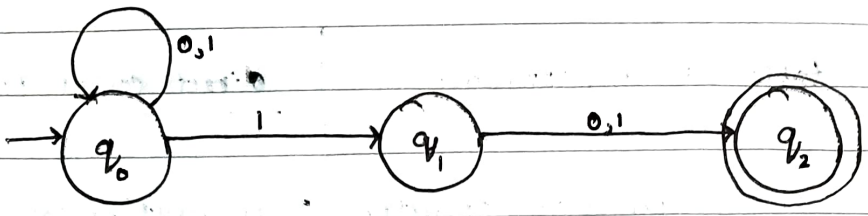
table driven approach	direct coded approach
• one multiplication operation per terminal symbol.	• no multiplication operation.
• One branch instruction per terminal. but these branches are highly predictable (loop back).	• one branch instruction per terminal. but predicting the branch targets could be hard.
• good code locality.	• code locality?
• data locality?	• data locality?

LEXICAL ANALYZER DESIGN

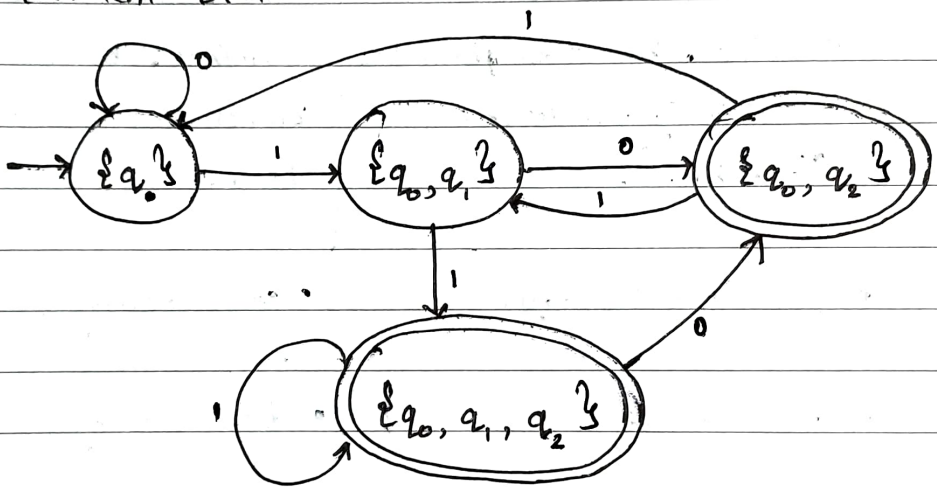
Q given a sequence of REs $R_1, R_2, \ldots R_k$, construct a function which takes a string as input and outputs the first RE R_i describing the input string.

time complexity: $O(kt)$
can we design an $O(t)$ algorithm for the checkstring function?

RECALL NFA TO DFA CONVERSION ALGORITHM



equivalent DFA



1. convert each of the REs to their respective NFAs.

2. unify the NFAs.

3. convert the unified NFA into a DFA.

4. examine each final state in the DFA (2 possible cases)
   (a) accepting states are coming from only one NFA.
   (b) accepting states are coming from different NFAs.
   use priority rule.

5. we can use the DFA to do the search over all REs in one shot!

- KEYWORD VS IDENTIFIER

- unify the NFAs for if and Identifier