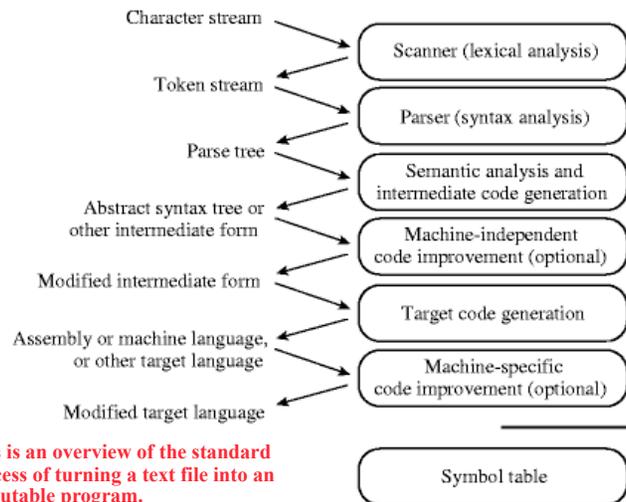


# 4

## Lexical analysis

## Concepts

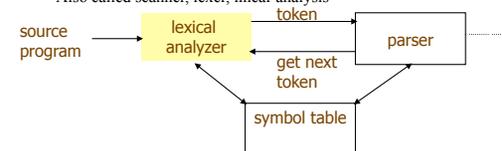
- Lexical scanning
- Regular expressions
- DFAs and FSAs
- Lex



**This is an overview of the standard process of turning a text file into an executable program.**

## Lexical analysis in perspective

- LEXICAL ANALYZER: Transforms character stream to token stream
  - Also called scanner, lexer, linear analysis



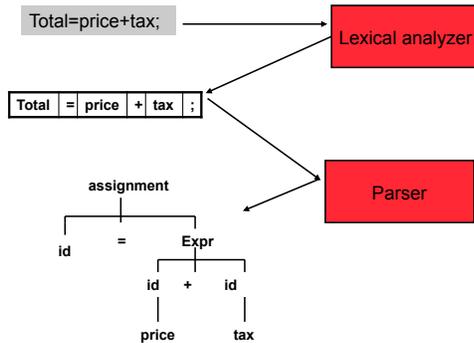
### LEXICAL ANALYZER

- Scans Input
- Removes whitespace, newlines, ...
- Identifies Tokens
- Creates Symbol Table
- Insert Tokens into symbol table
- Generates Errors
- Sends Tokens to Parser

### PARSER

- Performs Syntax Analysis
- Actions Dictated by Token Order
- Updates Symbol Table Entries
- Creates Abstract Rep. of Source
- Generates Errors

## Where we are



## Basic terminologies in lexical analysis

- Token
  - A classification for a common set of strings
  - Examples: <identifier>, <number>, etc.
- Pattern
  - The rules which characterize the set of strings for a token
  - Recall file and OS wildcards (\*.java)
- Lexeme
  - Actual sequence of characters that matches pattern and is classified by a token
  - Identifiers: x, count, name, etc...

## Examples of token, lexeme and pattern

If (price + gst - rebate <= 10.00) gift := false

Token	lexeme	Informal description of pattern
if	if	if
Lparen	(	(
identifier	price	String consists of letters and numbers and starts with a letter
operator	+	+
identifier	gst	String consists of letters and numbers and starts with a letter
operator	-	-
identifier	rebate	String consists of letters and numbers and starts with a letter
Operator	<=	Less than or equal to
constant	10.00	Any numeric constant
rparen	)	)
identifier	gift	String consists of letters and numbers and starts with a letter
Operator	:=	Assignment symbol
identifier	false	String consists of letters and numbers and starts with a letter

## Regular expression

- Scanners are usually based on *regular expressions (REs)*
- These are simpler and less expressive than BNF.
- Examples of a regular expression
  - Letter: a|b|c...|z|A|B|C...|Z
  - Digit: 0|1|2|3|4|5|6|7|8|9
  - Identifier: letter (letter | digit)\*
- Basic operations:
  - Set union
  - Concatenation
  - Kleene closure
- No recursion!

## Formal language operations

Operation	Notation	Definition	Example $L=\{a, b\}$ $M=\{0,1\}$
union of L and M	$L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$	$\{a, b, 0, 1\}$
concatenation of L and M	$LM$	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$	$\{a0, a1, b0, b1\}$
Kleene closure of L	$L^*$	$L^*$ denotes zero or more concatenations of L	All the strings consists of "a" and "b", plus the empty string. $\{\epsilon, a, b, aa, bb, ab, ba, aaa, \dots\}$
positive closure	$L^+$	$L^+$ denotes "one or more concatenations of "L"	All the strings consists of "a" and "b".

## Regular expression

- Let  $\Sigma$  be an alphabet,  $r$  a regular expression then  $L(r)$  is the language that is characterized by the rules of  $r$
- Definition of regular expression
  - $\epsilon$  is a regular expression that denotes the language  $\{\epsilon\}$
  - If  $a$  is in  $\Sigma$ ,  $a$  is a regular expression that denotes  $\{a\}$
  - Let  $r$  and  $s$  be regular expressions with languages  $L(r)$  and  $L(s)$ . Then
    - $(r) \mid (s)$  is a regular expression  $\rightarrow L(r) \cup L(s)$
    - $(r)(s)$  is a regular expression  $\rightarrow L(r) L(s)$
    - $(r)^*$  is a regular expression  $\rightarrow (L(r))^*$
- It is an inductive definition!
- A regular language is a language that can be defined by a regular expression

## Regular expression example revisited

- Examples of regular expression
  - Letter:  $a \mid b \mid c \mid \dots \mid z \mid A \mid B \mid C \dots \mid Z$
  - Digit:  $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
  - Identifier:  $\text{letter} (\text{letter} \mid \text{digit})^*$
- Q: why it is an regular expression?
  - Because it only uses union, concatenation and Kleene closure

## Precedence of operators

- $*$  is of the highest precedence;
- Concatenation comes next;
- $\mid$  lowest.
- All the operators are left associative.
- Example
  - $(a) \mid ((b)^*(c))$  is equivalent to  $a \mid b^*c$

## Properties of regular expressions

We can easily determine some basic properties of the operators involved in building regular expressions.

Property	Description
$r s = s r$	$ $ is commutative
$r (s t) = (r s) t$	$ $ is associative
$(rs)t = r(st)$	Concatenation is associative
$r (s t) = rs   rt$ $(s t)r = sr   tr$	Concatenation distributes over $ $
... ..	

## Notational shorthand of regular expression

- One or more instance
  - $L^+ = L L^*$
  - $L^* = L^+ | \epsilon$
  - Example
    - »  $\text{digits} \rightarrow \text{digit digit}^*$
    - »  $\text{digits} \rightarrow \text{digit}^+$
- Zero or one instance
  - $L? = L|\epsilon$
  - Example:
    - »  $\text{Optional\_fraction} \rightarrow .\text{digits}|\epsilon$
    - »  $\text{optional\_fraction} \rightarrow (.digits)?$
- Character classes
  - $[abc] = a|b|c$
  - $[a-z] = a|b|c \dots |z$

More syntactic sugar

## Regular grammar and regular expression

- They are equivalent
  - Every regular expression can be expressed by regular grammar
  - Every regular grammar can be expressed by regular expression
- Example
  - An identifier must begin with a letter and can be followed by arbitrary number of letters and digits.

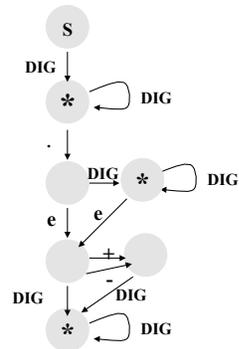
Regular expression	Regular grammar
ID: LETTER (LETTER   DIGIT)*	ID $\rightarrow$ LETTER ID_REST ID_REST $\rightarrow$ LETTER ID_REST   DIGIT ID_REST   EMPTY

## Formal definition of tokens

- A set of tokens is a set of strings over an alphabet
  - $\{\text{read, write, +, -, *, /, :=, 1, 2, \dots, 10, \dots, 3.45e-3, \dots}\}$
- A set of tokens is a *regular set* that can be defined by using a *regular expression*
- For every regular set, there is a *deterministic finite automaton* (DFA) that can recognize it
  - Aka deterministic Finite State Machine (FSM)
  - *i.e.* determine whether a string belongs to the set or not
  - Scanners extract tokens from source code in the same way DFAs determine membership

## Token Definition Example

- Numeric literals in Pascal, e.g.  
1, 123, 3.1415, 10e-3, 3.14e4
- Definition of token *unsignedNum*  
 $DIG \rightarrow 0|1|2|3|4|5|6|7|8|9$   
 $unsignedInt \rightarrow DIG DIG^*$   
 $unsignedNum \rightarrow$   
 $unsignedInt$   
 $((. unsignedInt) | \epsilon)$   
 $((e (+ | - | \epsilon) unsignedInt) | \epsilon)$



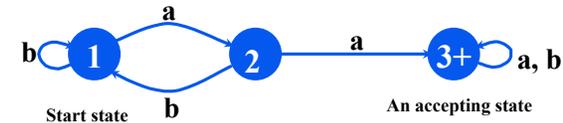
### Notes:

- Recursion is not allowed!
- Parentheses used to avoid ambiguity
- It's always possible to rewrite removing epsilons ( $\epsilon$ )

- FAs with epsilons are **nondeterministic**.
- NFAs are **much harder to implement (use backtracking)**
- Every NFA can be rewritten as a DFA (gets larger, tho)

## Simple Problem

- Write a C program which reads in a character string, consisting of a's and b's, one character at a time. If the string contains a double aa, then print string *accepted* else print string *rejected*.
- An abstract solution to this can be expressed as a DFA



The state transitions of a DFA can be encoded as a table which specifies the new state for a given current state and input

		input	
		a	b
current state	1	2	1
	2	3	1
	3	3	3

```
#include <stdio.h>
main()
{ enum State {S1, S2, S3}; an approach in C
  enum State currentState = S1;
  int c = getchar();
  while (c != EOF) {
    switch(currentState) {
      case S1: if (c == 'a') currentState = S2;
              if (c == 'b') currentState = S1;
              break;
      case S2: if (c == 'a') currentState = S3;
              if (c == 'b') currentState = S1;
              break;
      case S3: break;
    }
    c = getchar();
  }
  if (currentState == S3) printf("string accepted\n");
  else printf("string rejected\n");
}
```

## Using a table simplifies the program

```
#include <stdio.h>
main()
{ enum State {S1, S2, S3};
  enum Label {A, B};
  enum State currentState = S1;
  enum State table[3][2] = {{S2, S1}, {S3, S1}, {S3, S3}};
  int label;
  int c = getchar();
  while (c != EOF) {
    if (c == 'a') label = A;
    if (c == 'b') label = B;
    currentState = table[currentState][label];
    c = getchar();
  }
  if (currentState == S3) printf("string accepted\n");
  else printf("string rejected\n");
}
```

# Lex

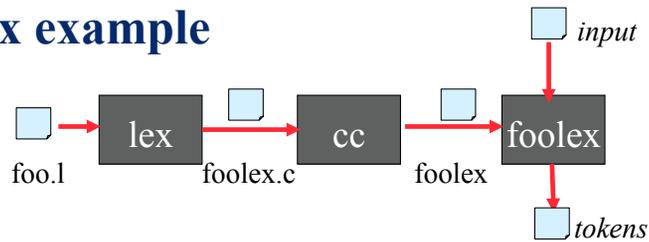
- Lexical analyzer generator
  - It writes a lexical analyzer
- Assumption
  - each token matches a regular expression
- Needs
  - set of regular expressions
  - for each expression an action
- Produces
  - A C program
- Automatically handles many tricky problems
- flex is the gnu version of the venerable unix tool lex.
  - Produces highly optimized code

# Scanner Generators

- E.g. lex, flex
- These programs take a table as their input and return a program (i.e. a scanner) that can extract tokens from a stream of characters
- A very useful programming utility, especially when coupled with a parser generator (e.g., yacc)
- standard in Unix



# Lex example



```
> flex -ofoolex.c foo.l
> cc -ofoolex foolex.c -lfl
```

```
>more input
begin
if size>10
then size * -3.1415
end
```

```
> foolex < input
Keyword: begin
Keyword: if
Identifier: size
Operator: >
Integer: 10 (10)
Keyword: then
Identifier: size
Operator: *
Operator: -
Float: 3.1415 (3.1415)
Keyword: end
```

# A Lex Program

... definitions ...  
%%  
... rules ...  
%%  
... subroutines ...

```
DIG [0-9]
ID [a-z][a-z0-9]*
%%
{DIG}+      printf("Integer\n");
{DIG}+"."{DIG}* printf("Float\n");
{ID}        printf("Identifier\n");
[ \t\n]+    /* skip whitespace */
.           printf("Huh?\n");
%%
main(){yylex();}
```

## Simplest Example

```
%%  
.\n    ECHO;  
%%  
  
main()  
{  
    yylex();  
}
```

## Strings containing aa

```
%%  
(a|b)*aa(a|b)*    {printf("Accept %s\n", yytext);}  
  
[a|b]+              {printf("Reject %s\n", yytext);}  
  
.\n                    ECHO;  
%%  
main() {yylex();}
```

## Rules

- Each rule has a *pattern* and an *action*.
- Patterns are regular expressions
- Only one action is performed
  - The action corresponding to the pattern matched is performed.
  - If several patterns match the input, the one corresponding to the **longest** sequence is chosen.
  - Among the rules whose patterns match the same number of characters, the rule given first is preferred.

```
/* scanner for a toy Pascal-like language */  
%{  
#include <math.h> /* needed for call to atof() */  
%}  
DIG [0-9]  
ID  [a-z][a-z0-9]*  
%%  
{DIG}+      printf("Integer: %s (%d)\n", yytext, atoi(yytext));  
{DIG}+".{DIG}* printf("Float: %s (%g)\n", yytext, atof(yytext));  
ifthen|begin|end printf("Keyword: %s\n",yytext);  
{ID}        printf("Identifier: %s\n",yytext);  
"+|-|"|"*|"/|" printf("Operator: %s\n",yytext);  
"{"["^"\n]*"}" /* skip one-line comments */  
[\t\n]+      /* skip whitespace */  
.  
printf("Unrecognized: %s\n",yytext);  
%%  
main() {yylex();}
```

## Flex's RE syntax

<b>x</b>	character 'x'
<b>.</b>	any character except newline
<b>[xyz]</b>	<i>character class</i> , in this case, matches either an 'x', a 'y', or a 'z'
<b>[abj-oZ]</b>	<i>character class</i> with a range in it; matches 'a', 'b', any letter from 'j' through 'o', or 'Z'
<b>[^A-Z]</b>	<i>negated character class</i> , i.e., any character but those in the class, e.g. any character except an uppercase letter.
<b>[^A-Z\n]</b>	any character EXCEPT an uppercase letter or a newline
<b>r*</b>	zero or more r's, where r is any regular expression
<b>r+</b>	one or more r's
<b>r?</b>	zero or one r's (i.e., an optional r)
<b>{name}</b>	expansion of the "name" definition (see above)
<b>"[xy]"foo"</b>	the literal string: '[xy]"foo' (note escaped ")
<b>\x</b>	if x is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \x. Otherwise, a literal 'x' (e.g., escape)
<b>rs</b>	RE r followed by RE s (e.g., concatenation)
<b>r s</b>	either an r or an s
<b>&lt;&lt;EOF&gt;&gt;</b>	end-of-file