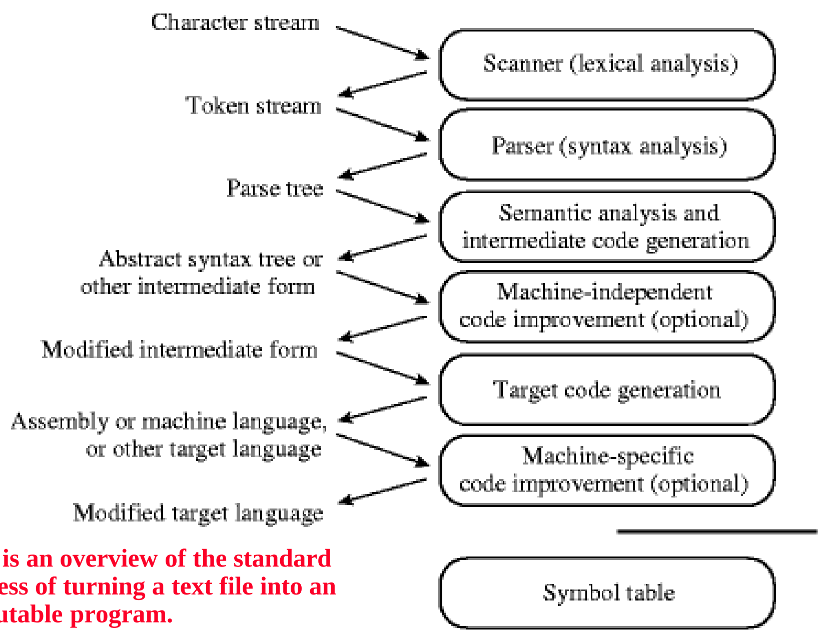
Chapter 4

Lexical analysis

Concepts

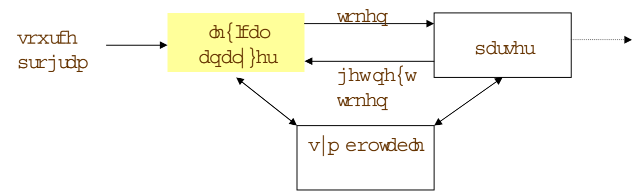
- Lexical scanning
- Regular expressions
- DFAs and FSAs
- Lex



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

Lexical analysis in perspective

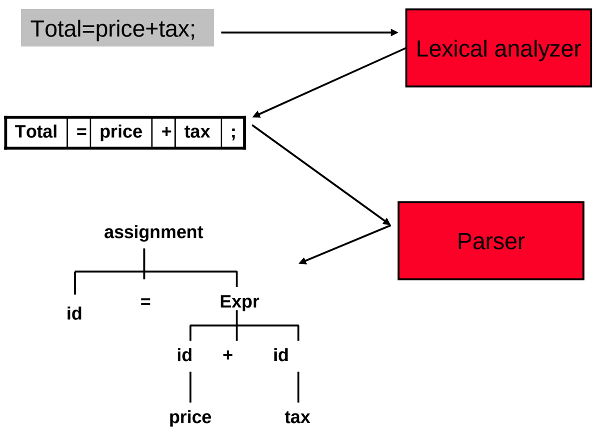
- LEXICAL ANALYZER: Transforms character stream to token stream
  - Also called scanner, lexer, linear analysis



LEXICAL ANALYZER
- Scans Input
- Removes whitespace, newlines, …
- Identifies Tokens
- Creates Symbol Table
- Inserst 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 useually based on *regular expressions*.
- Remember language is a set of strings.
- Examples of 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

Formal language operations

Operation	Notation	Definition	Example L={a, b} M={0,1}
union of L and M	L ∪ M	L ∪ M = {s | s is in L or s is in M}	{a, b, 0, 1}
concatenation of L and M	LM	LM = {st | s is in L and t 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. {ε, a, b, aa, bb, ab, ba, aaa, … }
positive closure	L+	L+ denotes "one or more concatenations of " L	All the strings consists of "a" and "b".

Regular expression

- Regular expression: constructing sequences of symbols (Strings) from an alphabet.
- Let Σ 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
  - ε is a regular expression that denotes the language {ε}
  - If a is in Σ, a is a regular expression that denotes {a}
  - Let r and s be regular expressions with languages L(r) and L(s). Then
    - » (r) | (s) is a regular expression → L(r) ∪ L(s)
    - » (r)(s) is a regular expression →L(r) L(s)
    - » (r)* is a regular expression → (L(r))*
- It is an inductive definition!
- Distinction between regular language and regular expression

Regular expression example revisited

- Examples of 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)*
- Q: why it is an regular expression?

Precedence of operators

- * is of the highest precedence;
- Concanenation comes next;
- | lowest.
- All the operators are left associative.
- Example
  - (a) | ((b)*(c)) is equivalent to a|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+ | ε
  - Example
    - » digits→ digit digit*
    - » digits→digit+
- Zero or one instance
  - L? = L|ε
  - Example:
    - » Optional_fraction→.digits|ε
    - » optional_fraction→(.digits)?
- Character classes
  - [abc] = a|b|c
  - [a-z] = a|b|c...|z

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.

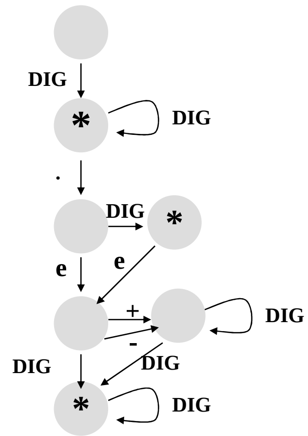
Regular expression	Regular grammar
ID: LETTER (LETTER | DIGIT)*	ID → LETTER ID_REST ID_REST → LETTER ID_REST | DIGIT ID_REST | EMPTY

Formal definition of tokens

- A set of tokens is a set of strings over an alphabet
  - {read, write, +, -, *, /, :=, 1, 2, …, 10, …, 3.45e-3, …}
- 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

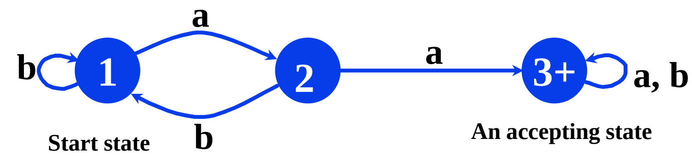
- Numeric literals in Pascal, e.g.
  1, 123, 3.1415, 10e-3, 3.14e4
- Definition of token *unsignedNum*
  DIG → 0|1|2|3|4|5|6|7|8|9
  unsignedInt → *DIG DIG**
  unsignedNum →
  unsignedInt
  ((. *unsignedInt*) | ε)
  ((e (+ | – | ε) *unsignedInt*) | ε)
- Notes:
  – Recursion is not allowed!
  – Parentheses used to avoid ambiguity
  – It's always possible to rewrite removing epsilons



- **FAs with epsilons are nondeterministic.**
- **NFAs are much harder to implement (use backtracking)**
- **Every NFA can be rewriten 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

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

an approach in C

```
#include <stdio.h>
main()
{   enum State {S1, S2, S3};
    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

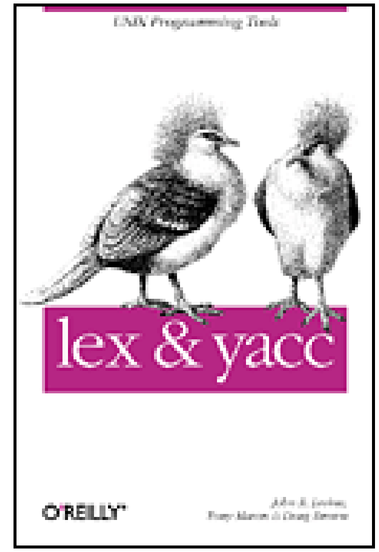
```
#include <stdio.h>
main()
{  enum State {S1, S2, S3};
   enum Label {A, B};
   enum State currentState = S1;
   enum State table[3][2] = {{S2, S1}, {S3, S1}, {S3, S3}};
   int label;
   int c = getchar();
   while (c != EOF) {
       if (c == 'a') label = A;
       if (c == 'b') label = B;
     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

Lex

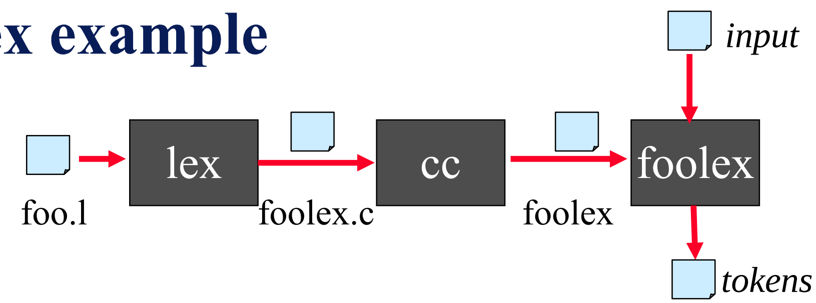
- Lexical analyzer generator
  - It writes a lexical analyzer
- Assumption
  - each token matches a regular expression
- Needs
  - set of regular expressions
  - for each expression an action
- Produces
  - A C program
- Automatically handles many tricky problems
- flex is the gnu version of the venerable 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 has a rule has a *pattern* and an *action.*
- Patterns are regular expression
- 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));
if|then|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

x character 'x'

. any character except newline

[xyz] *character class*, in this case, matches either an 'x', a 'y', or a 'z'

[abj-oZ] *character class* with a range in it; matches 'a', 'b', any letter from 'j' through 'o', or 'Z'

[^A-Z] *negated character class*, i.e., any character but those in the class, e.g. any character except an uppercase letter.

[^A-Z\n] any character EXCEPT an uppercase letter or a newline

r* zero or more r's, where r is any regular expression

r+ one or more r's

r? zero or one r's (i.e., an optional r)

{name} expansion of the "name" definition (see above)

"[xy]\"foo" the literal string: '[xy]"foo' (note escaped “)

\x 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)

rs RE r followed by RE s (e.g., concatenation)

r|s either an r or an s

<<EOF>> end-of-file