Chapter 4
Lexical analysis

Scanner

• Main task: identify tokens
  – Basic building blocks of programs
  – E.g. keywords, identifiers, numbers, punctuation marks
• Desk calculator language example:
  
  read A
  sum := A + 3.45e-3
  write sum
  write sum / 2

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 comprehension 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

Regular Expressions

A regular expression (RE) is:

• A single character
• The empty string, \(\varepsilon\)
• The concatenation of two regular expressions
  – Notation: \(RE_1 \cdot RE_2\) (i.e. \(RE_1\) followed by \(RE_2\))
• The union of two regular expressions
  – Notation: \(RE_1 \cup RE_2\)
• The closure of a regular expression
  – Notation: \(RE^*\)
  – \(*\) is known as the Kleene star
  – \(*\) represents the concatenation of 0 or more strings
• Caution: notations for regular expressions vary
  – Learn the basic concepts and the rest is just syntactic sugar
### 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 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

```
#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

```
DIG [0-9]
ID [a-zA-Z][a-zA-Z0-9]*

{DIG}+                  printf("Integer\n");
{DIG}+\.[DIG]*          printf("Float\n");
{ID}                    printf("Identifier\n");
[ \t\n ]+               /* skip whitespace */
.                       printf("Huh?:\n");

main(){yylex();}
```

A Lex Program

```
DIG [0-9]
ID [a-zA-Z][a-zA-Z0-9]*

%\
%\
... definitions ...

%\
%\
... rules ...

%\
%\
... subroutines ...
```

For more details and examples, refer to the course material or the provided diagrams.
Simplest Example

```c
%%
\|\n  ECHO;
%%

main()
{
  yylex();
}
```

Strings containing aa

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

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.

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
/* 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);
[ \\n]+                    /* skip whitespace */
.                       printf("Unrecognized: %s\n",yytext);

main(){yylex();}