[2回目　２００２・９・１０]

字句解析の基礎：正規表現によるパターンマッチ

字句解析とは、文字列として入力されるプログラムを token の列に分解するフェーズである。前回のプログラムにおいて、字句解析を行う関数 getToken は、文字列を数字(NUM)、演算子(PLUS_OP,MINUS_OP)などの token に分けて、返す関数である。

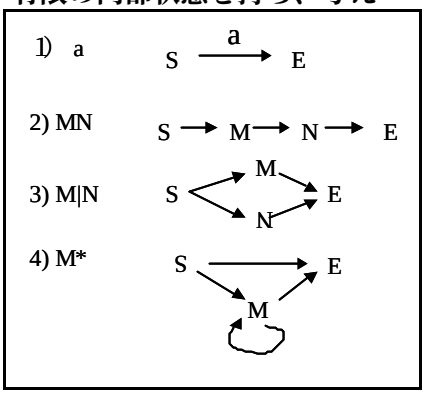
どのような文字列がどのような token になるかについては、**正規表現(regular expression)**で定義することができる。アルファベット A 上の正規表現とは、

(1) ε (空列記号)は正規表現である。
(2) A の要素 a は正規表現である。
(3) R と S が正規表現であれば、M | N、MN,　M*　は正規表現である。M | N とは、M もしくは N、MN は M の次に N がくる列、M*とは、M の 0 回以上繰り返しを意味する。

なお、(S) は、S と同等であることを意味する。

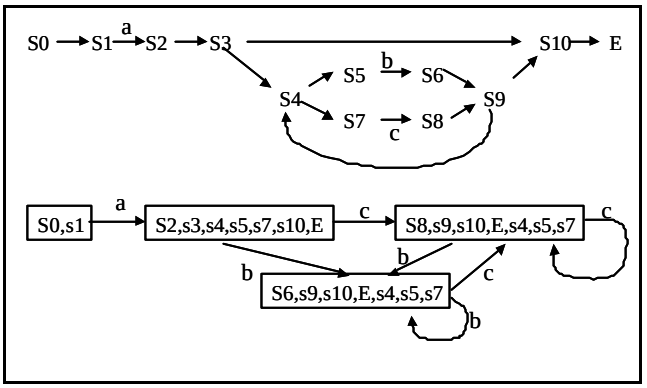
例えば、正規表現 a(b | c)*は、最初に a があり、b または c が続く文字列を表現する。abbc も abcbcc も、abcc もこの正規表現で表現される文字列である。

正規表現は、図のような規則で非決定性有限オートマトン（NDA: nondeterministic finite automaton）に変換できる。有限オートマトン(finite automaton)とは、有限の内部状態を持ち、与えられた記号列を読みこみながら状態遷移し、その記号列があるパターンをもつ列であるかを決定するものである。非決定性有限オートマトンとは、入力によらない状態遷移（空列記号に対する状態遷移）をもち、それは非決定的に遷移してもしなくてもよいと状態をもつものである。図にそれぞれの規則に対応するオートマトンを図示する。



先にあげた a(b | c)*について、この規則で、NDA に変換した結果も示す。NDA では、空の状態遷移に対して、状態遷移するかしないかの両方の可能性をしらべなくてはならないので、実際にそのまま実装すると効率がわるい。そのため、非決定的な遷移を取り除き、**決定性有限オートマトン(DFA: deterministic finite automaton)**に変換する。DFA とは、以下の条件を満たす有限オートマトンである。

（１）εによる遷移がない。
（２）一つの状態から同じ記号による異なった状態への遷移はない。



変換には次の規則を適用し、上の条件にあったオートマトンに変換する。

（１）初期状態から、εによる遷移を一まとめにした集合を初期状態とする。
（２）状態の集合からの遷移は、その集合からの遷移の集合の合併とする。つまり、状態の集合 D={x1,x2,...}からの a による遷移の行き先は、x1 から a で遷移した状態 y もしくは、y からε で遷移する状態の集合になる。
（３）２を繰り返し新しい遷移が得られなくなるまで繰り返す。

このアルゴリズムで得られる FDA は必ずしも、最小のオートマトンとはならない。最小にするには、同じ遷移が２つあった場合には、冗長なので１つにまとめることができ、これを繰り返すことにより、最小化ができる。

自動字句解析生成プログラム：lex

正規表現が与えられた時に、その言語（文字のパターン）を認識する DFA を機械的に作り出すことができる。そのアルゴリズムをプログラムにしたものが字句解析器生成系(lexical analyzer generator)である。このプログラムとして、有名なのが、lex である。lex では、定義ファイルは以下の形式でかく。

```
%{
任意の C プログラム。定数や C のマクロの宣言をここにかく。
%}
下の定義で使う lex のマクロの定義
```

%%
正規表現による入力パターンの定義
正規表現パターン　アクション　という形で書く
%%
任意の C プログラム

例えば、a(b|c)*の正規表現を認識する字句解析は、

```
%{
%}
%%
a(b|c)*  { printf("OK¥n"); }  /*このパターンを入力したら OK を出力する*/
.  {printf("NG¥n");}  /*.は以上以外のパターンの場合のアクションを指定 */
%%
/*   なにもなし  */
```

でよい。これをコンパイルするには、

```
% lex test.l
% cc lex.yy.c -ll
```

とする。これでできた a.out を実行すると、abc にたいしては OK, xxx には NG と出力される。Linux の flex のときには、-ll の代わりに、-lfl をつかうこと。

前回示した数式の字句解析の部分は以下のように書ける。

```
%{
#include "expr.h"
%}
digit   [0-9]  /* マクロの定義，digit を 0-9 の数字の集合と定義する  */
%%
{digit}+  return NUM;    /*0-9 の１回以上の繰り返しは、NUM と認識する  */
"+"     return PLUS_OP; /*+は PLUS_OP +は繰り返しの意味なので、""で囲む*/
"-"     return MINUS_OP, /* -は MINUS_OP */
[ ¥t]                    /* 空白は無視  */
.      printf("error?¥n");   /* error */
%%
main(){
    yystdin = stdout;
    ....
    r = yylex(); /* token の列は yytext に入る  */
    printf(" token is %d,'%s'¥n",r,yytext); }
```

lex は、字句解析ルーチンとして、yylex を生成する。このルーチンは、action で指定された return 分で返された値を返す。

lex の使い方については、

```
% man lex
```

として、マニュアルを参照のこと。

演習課題 2：

標準入力から、文字列を入力し、浮動少数点数を入力した場合、YES、それ以外の場合、NO を標準出力するプログラムを lex を使って作りなさい。浮動少数点の正規表現は、

浮動少数点 := 少数点数(ε|指数部) | 数字列　指数部
少数点数　:= (ε | 数字列)　.　数字列 | 数字列　.
指数部　:= E (ε | 符号) 数字列
数字列　:= 数字 | 数字列　数字
符号　:= - | +

なお、数字は 0 から 9 までの数字、浮動少数点数の符号は考えない。