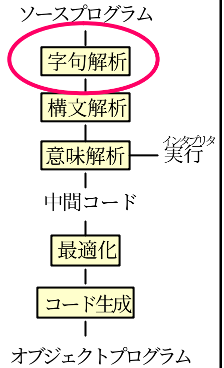
第２回（平成１５年度９月９日）
字句解析の基礎
lexのつかい方
top-down parserのつかい方
tiny Cの概要とデータ構造

筑波大学　佐藤

言語処理系の基本構成

- 字句解析(lexical analysis): 文字列を言語の要素（トークン、token）の列に分解する。
- 構文解析(syntax analysis): token列を意味を反映した構造に変換。この構造は、しばしば、木構造で表現されるので、抽象構文木（abstract syntax tree）と呼ばれる。ここまでの言語を認識する部分を言語のparserと呼ぶ。
- 意味解析(semantics analysis): 構文木の意味を解析する。インタプリターでは、ここで意味を解析し、それに対応した動作を行う。コンパイラでは、この段階で内部的なコード、中間コードに変換する。

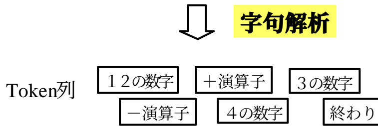


字句解析

- 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。
- 「式」という言語では、tokenとして、数字と"+"や"-"といった演算子がある。

入力された文字列　'1','2','+','3','-','4'

字句解析

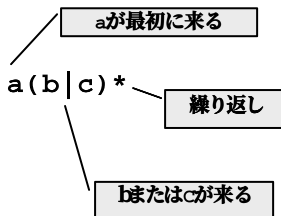


正規表現

- 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、
  - e (空列記号)は正規表現である。
  - Aの要素aは正規表現である。
  - MとNが正規表現であれば、以下が正規表現
    - M | N　　Mもしくは N
    - M N　　Mの後にNが来る
    - M*　　Mの0回以上の繰り返し
    - (S)　　はSと同じ

正規表現の例

- a(b|c)*は、aの後にbまたはcの繰り返し
  - abbc
  - abcbbcc

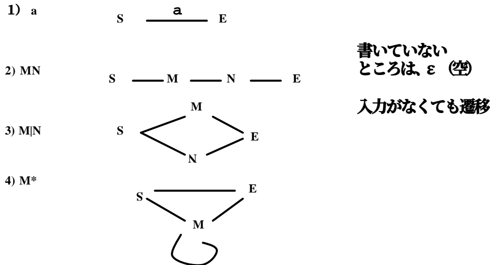


正規表現とNDA

- 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
- 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
  - 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

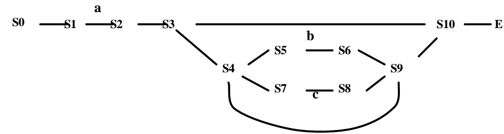
正規表現の規則とNDA

◆ それぞれの規則に対応するオートマトン



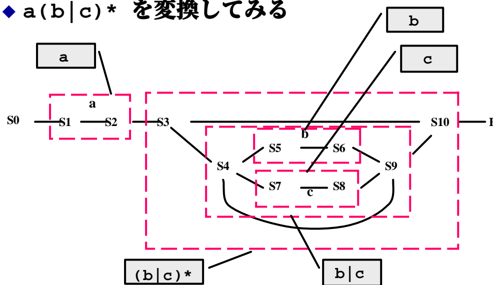
正規表現からNDAへの変換

◆ a(b|c)* を変換してみる



正規表現からNDAへの変換

◆ a(b|c)* を変換してみる



NDAからDFAへの変換

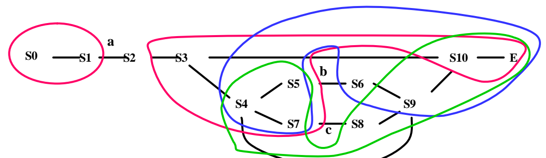
- ◆ NDAでは、空の状態遷移に対して、状態遷移するかしないかの両方の可能性をしらべなくてはならないので、実際にそのまま実装すると効率がわるい
- ◆ 非決定的な遷移を取り除き、<u>決定性有限オートマトン(DFA: deterministic finite automaton)</u>に変換する
- ◆ DFAとは
  1. εによる遷移がない。
  2. 一つの状態から同じ記号による異なった状態への遷移はない。

NDAからDFAへの変換

- ◆ NDAからDFAへの変換規則
  1. 初期状態から、e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e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

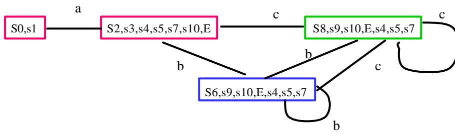
NDAからDFAへの変換の例

- ◆ a(b|c)* を変換してみる
  1. 初期状態から、e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e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



NDAからDFAへの変換の例

- まとめると



最小化

- 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)

- lexが有名

lexの書き方

- 定義ファイルに記述

```
%{
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
```


lexの例

- a(b|c)* を記述してみる

パターン　アクション

```
%{
%}
%%
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
%%
/*   Cのプログラムは、なにもなし  */
```


lexのつかい方

- Lexコマンドの使い方
  - lexの定義ファイルをtest.lとする。
  - *.l は、lexのextention
  - lex.yy.cというCのプログラムを生成
  - これを-llでリンク

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

lex.yy.cを生成

-lflかも？

- 字句解析ルーチンyylex()を生成する。
- 特にmainを指定しない場合には、文字列を入力してactionを実行するプログラムを生成する。

lexの例

- 前回の字句解析ルーチン

```
%{
#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
"-"      return MINUS_OP, /* -はMINUS_OP */
[ ¥t]                     /* 空白は無視 */
.        printf("error?¥n");   /* error */
%%
```


yylexの呼び出し

- yylex()を作る
- action の中のreturnで返されるtokenを返す

```
main()
{
    yystdin = stdin;
    ....
    r = yylex(); /* tokenの列はyytextに入る */
    printf(" token is %d,'%s'¥n",r,yytext);
 }
```


lexのマニュアル

- man コマンド

```
% 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(e|指数部) | 数字列 指数部
少数点数 := (e | 数字列) . 数字列 | 数字列 .
  指数部 := E (e | 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。