

Novembar 2002.

1. Ovo je još jedan od automata koji se odnosi na deljivost prirodnih brojeva, tako da se konstruiše na prilično uobičajen način.

Neka konstruišemo transduktor koji deli broj a , zapisan u osnovi m brojem n . Ideja je da nekako simuliramo algoritam deljenja.

Neka je broj $a = (a_1 a_2 \dots a_r)_m$. Označimo sa o_k ostatak pri deljenju broja $(a_1 \dots a_k)_m$, brojem n , tj. neka je o_k ostatak pri deljenju broja zapisanog sa prvih k cifara broja a brojem n . Konstrukcija se zasniva na činjenici da se samo na osnovu ostatka o_k i na osnovu k -te cifre broja a tj. a_k može odrediti ostatak o_{k+1} , kao i k -tu cifru količnika tj. b_k . Zaista, spuštanje k -te cifre predstavlja dodavanje a_k na proizvod dosadašnjeg ostatka i broja m . Novi ostatak je ostatak pri deljenju tako dobijenog broja brojem n , dok je k -ta cifra količnika cifra koja se dobije kao celobrojni količnik tako dobijenog broja brojem n . Dakle važe sledeće jednakosti:

$$o_k = (o_{k-1} \cdot m + a_k) \bmod n$$

$$b_k = (o_{k-1} \cdot m + a_k) \operatorname{div} n$$

Pri tom važi da je $o_0 = 0$.

Ilustrujmo to primerom. Neka je $n = 3$, a $m = 10$. Znači delimo dekadni broj trojkom. Uzmimo npr. $a = 25365$.

```
25384 : 3 = 08461
25
13
18
04
1
```

Izračunavanje teče ovako : $o_0 = 0$. Pošto je $a_1 = 2$, na osnovu gornjih jednakosti sledi da je

$$o_1 = (0 \cdot 10 + 2) \bmod 3 = 2$$

a

$$b_1 = (0 \cdot 10 + 2) \operatorname{div} 3 = 0$$

Dalje važi:

$$o_2 = (2 \cdot 10 + 5) \bmod 3 = 1$$

$$b_2 = (2 \cdot 10 + 5) \operatorname{div} 3 = 8$$

$$o_3 = (1 \cdot 10 + 3) \bmod 3 = 1$$

$$b_3 = (1 \cdot 10 + 3) \operatorname{div} 3 = 4$$

$$o_4 = (1 \cdot 10 + 8) \bmod 3 = 0$$

$$b_4 = (1 \cdot 10 + 8) \operatorname{div} 3 = 6$$

$$o_5 = (0 \cdot 10 + 4) \bmod 3 = 1$$

$$b_5 = (0 \cdot 10 + 4) \operatorname{div} 3 = 1$$

Sada može da se konstuiše transduktor. Njegova stanja će da predstavljaju sve ostatake pri deljenju brojem n . Stanja su, dakle, $0, 1, \dots, n-1$. Na osnovu gore navedenog se lako definišu funkcija prelaska i ispisa.

Zapišimo transduktor pomoću tabele. Neka vrste tabele predstavljaju stanja transduktora, a kolone slova oktalne azbuke. Svako polje tabele sadrži dve vrednosti zapisane u obliku a/b . Vrednost a predstavlja novo stanje transduktora, dok b predstavlja slovo koje se emituje na izlaz.

sta/isp	0	1	2	3	4	5	6	7
0	0/0	1/0	2/0	3/0	4/0	5/0	6/0	0/1
1	1/1	2/1	3/1	4/1	5/1	6/1	0/2	1/2
2	2/2	3/2	4/2	5/2	6/2	0/3	1/3	2/3
3	3/3	4/3	5/3	6/3	0/4	1/4	2/4	3/4
4	4/4	5/4	6/4	0/5	1/5	2/5	3/5	4/5
5	5/5	6/5	0/6	1/6	2/6	3/6	4/6	5/6
6	6/6	0/7	1/7	2/7	3/7	4/7	5/7	6/7

Slika 1: Transduktor

Problem ovog transduktora je to što se na početku količnika ispisuje možda nepotrebna 0. To se može zaobići uvođenjem novog početnog stanja, $0'$ koje ima iste prelaze kao i stanje 0, sa tim što da su ispisi jednaki ε , gde god su bili 0.

AWK implementacija sledi prilično jednostavno.

```

BEGIN { FS="[ \t]+"
        prelaz["-1","0"]="0"
        prelaz["-1","1"]="1"
        prelaz["-1","2"]="2"
        prelaz["-1","3"]="3"
        prelaz["-1","4"]="4"
        prelaz["-1","5"]="5"
        prelaz["-1","6"]="6"
        prelaz["-1","7"]="0"
        prelaz["0","0"]="0"
        prelaz["0","1"]="1"
        prelaz["0","2"]="2"
        .....
        prelaz["6","5"]="4"
        prelaz["6","6"]="5"
        prelaz["6","7"]="6"

        prevod["-1","0"]=" "
        prevod["-1","1"]=" "
        prevod["-1","2"]=" "
        prevod["-1","3"]=" "
        prevod["-1","4"]=" "
        prevod["-1","5"]=" "

```

```

        prevod["-1","6"]=" "
        prevod["-1","7"]="1"
        prevod["0","0"]="0"
        prevod["0","1"]="0"
        prevod["0","2"]="0"
        .....
        prevod["6","5"]="7"
        prevod["6","6"]="7"
        prevod["6","7"]="7"
    }

    { for(i=1; i<=NF; i++)
      /*Prepoznavamo sve oktalne brojeve*/
      if ($i ~ /^[0-7]+$/)
      { /* Simuliramo rad transduktora */
        stanje="-1"
        x=$i
        printf("%s=7*", $i)
        while (x!="")
        { prvi=substr(x,1,1)
          a=prevod[stanje,prvi]
          printf("%c",a)
          stanje=prelaz[stanje,prvi]
          x=substr(x,2)
        }
        printf("+%c\n",stanje)
      }
    }
}

```

2.

3. Kao što se iz samog teksta zadatka moglo videti, potrebno je bilo konstruisati relaciju \Rightarrow^+ , koja, kako se iz same oznake vidi, predstavlja tranzitivno zatvorenje relacije neposrednog izvodjenja. Najveći deo posla je predstavljalo konstrukcija same relacije izvodjenja na osnovu sadržaja ulazne datoteke. Tranzitivno zatvorenje se lako nalazi Varšalovim algoritmom. Ono što je opredeljivalo dalju izradu zadatka je struktura podataka relacije. Najjednostavnije rešenje (dovoljno samo za najniže ocene), je bilo da koristimo matricu unapred zadate dimenzije. Pošto je relacija definisana nad skupom neterminala, potrebno je bilo nekako indeksirati neterminale. To je ponovo najlakše bilo uraditi smeštajući ih u statički niz, tako da indeks neterminala odgovara njegovoj poziciji u tom nizu.

```

%{
#include<stdio.h>
#include<string.h>
#define yyerror printf
#define YYSTYPE int

```

```

#define MAXNET 50

extern char* yytext;

char* neterms[MAXNET];
int relation[MAXNET][MAXNET];

/* Pomocna promenjiva koja pamti kod neterminala
   koji se pojavio sa leve strane pravila koje se
   upravo obradjuje */
int leftnet;

/* Najveci kod neterminala koji smo do sada pronasli */
int max_net_code=0;

int findNCode(char*);
void warshall(int[][]);
%}

%token ARROW SEMI TERM NETERM BAR
%start rules
%%
rules    :    rule rules
          |
          ;

rule      : neterm {leftnet=$1;
                   /* Upamtimo ovaj neterminal*/ }
          ARROW rightsides SEMI
          ;

rightsides : rightside BAR rightsides
           | rightside
           ;

rightside  : neterm
           { /* Modifikujemo relaciju izvodjenja */
             relation[leftnet][$1]=1;} rightside
           | TERM rightside
           |
           ;

neterm    : NETERM { /* Odredjujemo kod ovog neterminala.
                     Ako neterminal ne postoji u nizu,
                     dodaje se. Kod se zatim postavlja na
                     interni stek */
                     $$=findNCode(yytext);
                   }

%%

```

```

main()
{
    int i,j;
    /* Relacija izvodjenja je prazna u pocetku */
    for (i=0; i<MAXNET; i++)
        for (j=0; j<MAXNET; j++)
            relation[i][j]=0;

    yyparse();

    /* Nalazimo tranzitivno zatvorenje relacije izvodjenja */
    warshall(relation);

    /* Neterminal je tranzitivan ako je u R+ sam sa sobom */
    for (i=0; i<max_net_code; i++)
        if (relation[i][i])
            printf("%s je rekurzivan\n",neterms[i]);
}

int findNCode (char* n)
{
    int i;
    for (i=0; i<max_net_code; i++)
        if (strcmp(neterms[i],n)==0)
            return i;
    neterms[max_net_code]=strdup(n);
    return max_net_code++;
}

/* Varsalovim algoritmom se konstruise tranzitivno
   zatvorenje relacije */
void warshall(int rel[MAXNET][MAXNET])
{
    int i,j,k;
    for (k=0; k<max_net_code; k++)
        for (i=0; i<max_net_code; i++)
            for (j=0; j<max_net_code; j++)
                if (rel[i][k] && rel[k][j])
                    rel[i][j]=1;
}

```

Bolje rešenje koristi dinamičku matricu za čuvanje relacije. Takođe pošto nam je potrebno često nalaženje indeksa neterminala, indeksi se ne čuvaju u nizu, već u pretraživačkom binarnom drvetu.

```

%{
#include<stdio.h>
#include<string.h>
#define STEP 10
#define yyerror printf

```

```

#define YYSTYPE int

extern char* yytext;

/* Cvor pretrazivackog drveta koje sadrzi neterminale
   i njihove kodove */
typedef struct NC
{
    char* Net;
    int Code;
    struct NC *left, *right;
} NCNODE;

/* Koren drveta kodova neterminala */
NCNODE* nctree=NULL;

/* Relaciju izvodjenja pamtimo u dinamickoj matrici */
typedef struct RL
{
    /* Elementi matrice (0 ili 1) */
    int **rel;
    /* Velicina alociranog prostora za matricu */
    int size;
} RELATION;

/* Relacija izvodjenja */
RELATION relation;

/* Pomocna promenjiva koja pamti kod neterminala
   koji se pojavio sa leve strane pravila koje se
   upravo obradjuje */
int leftnet;

/* Najveci kod neterminala koji smo do sada pronasli */
int max_net_code=0;

/* Funkcije za rad sa drvetom kodova neterminala */
NCNODE* newNC(char*);
void deleteNC(NCNODE*);
int findNCode(NCNODE*, char*);
char* findNName(NCNODE*, int);

/* Funkcije za rad sa dinamickom matricom */
void reallocateR(RELATION*,int);
void printR(RELATION*);
void deleteR(RELATION*);
void setR(RELATION*,int,int);
void warshallR(RELATION*);
%}

%token ARROW SEMI TERM NETERM BAR

```

```

%start rules
%%
rules      :   rule rules
            |
            ;

rule       : neterm {leftnet=$1;
                    /* Upamtimo ovaj neterminal*/ }
            ARROW rightsides SEMI
            ;

rightsides : rightside BAR rightsides
            | rightside
            ;

rightside  : neterm
            { /* Modifikujemo relaciju izvodjenja */
              setR(&relation, leftnet,$1);
            }
            rightside
            | TERM rightside
            |
            ;

neterm     : NETERM { /* Odredjujemo kod ovog neterminala. Ako ne
                      postoji u drvetu, novi cvor se pravi */
                      /* Kod se zatim postavlja na interni stek */
                      if (nctree!=NULL)
                        $$=findNCode(nctree,yytext);
                      else
                        { nctree=newNC(yytext);
                          $$=nctree->Code;
                        }
                      }

%%

main()
{   int i;
    /* Relacija izvodjenja je prazna u pocetku */
    relation.size=0;
    relation.rel=NULL;

    printf("Pocinje parsiranje ...\n");
    yyparse();
    printf("Gotovo parsiranje \n");

    printR(&relation);

```

```

/* Nalazimo tranzitivno zatvorenje relacije izvodjenja */
warshallR(&relation);

printf("Tranzitivno zatvorenje je : \n");
printR(&relation);

/* Neterminal je tranzitivan ako je u R+ sam sa sobom */
for (i=0; i<max_net_code; i++)
    if (relation.rel[i][i])
        printf("%s je rekurzivan\n",findNName(nctree,i));

/* Uklanjammo relaciju */
deleteR(&relation);

/* Uklanjammo kodove neterminala */
deleteNC(nctree);
}

/* Funkcija koja kreira novi cvor drveta kodova na osnovu
   imena neterminala. Kod je prvi prirodan broj koji ne
   predstavlja vec kod nekog neterminala */
NCNODE* newNC(char* n)
{
    NCNODE* t=(NCNODE*)malloc(sizeof(NCNODE));
    t->Net=strdup(n);
    t->Code=max_net_code++;
    t->left=t->right=NULL;
    return t;
}

/* Funkcija koja uklanja drvo na koje pokazuje
   pokazivac n */
void deleteNC(NCNODE* n)
{
    if (n!=NULL)
    {
        deleteNC(n->left);
        deleteNC(n->right);
        free(n);
    }
}

/* Koristeci sortiranost po abecedi, funkcija vraca kod
   neterminala cije joj je ime prosledjeno */
int findNCode (NCNODE* t,char* n)
{
    int cmp=strcmp(t->Net,n);
    if (cmp==0)
        return t->Code;
    else if (cmp<0)
    {
        if (t->left!=NULL)
            return findNCode(t->left,n);
    }
}

```



```

        else
        {   t->left=newNC(n);
            return t->left->Code;
        }
    }
    else
    {   if (t->right!=NULL)
        return findNCode(t->right,n);
        else
        {   t->right=newNC(n);
            return t->right->Code;
        }
    }
}

```

/* Funkcija nalazi ime neterminala na osnovu
prosledjenog koda */

```

char* findNName (NCNODE* t, int c)
{   char* f;
    if (t==NULL) return NULL;
    if (t->Code==c)
        return t->Net;
    f=findNName(t->left,c);
    if (f) return f;
    f=findNName(t->right,c);
    if (f) return f;
    return NULL;
}

```

/* Funkcija vrši realociranje dinamičke matrice
na veličinu sxs */

```

void reallocateR(RELATION* R, int s)
{   int i,j;
    int **newR;

```

```

    if (R->size>=s)
        return;

```

```

    /* Kreiramo novu, vecu matricu */
    newR=(int**)malloc(s*sizeof(int*));
    for(i=0; i<s; i++)
        newR[i]=(int*)malloc(s*sizeof(int));

```

```

    /* Kopiramo elemente stare matrice, a na ostala
    mesta upisujemo 0 */

```

```

    for(i=0; i<R->size; i++)
    {   for(j=0; j<R->size; j++)
        newR[i][j]=R->rel[i][j];
    }

```

```

        for( ; j<s; j++)
            newR[i][j]=0;
    }
    for ( ; i<s; i++)
        for(j=0; j<s; j++)
            newR[i][j]=0;

    /* Brisemo staru matricu */
    deleteR(R);

    /* Sadržaj postavljamo na novu matricu
       i azuriramo velicinu */
    R->rel=newR;
    R->size=s;
}

/* Funkcija brise dinamicku matricu */
void deleteR(RELATION* R)
{
    int i;
    if (R->rel==NULL) return;
    for(i=0; i<R->size; i++)
        free(R->rel[i]);
    free(R->rel);
}

/* Funkcija za ispis matrice na ekran. Ne ispisuje
   se sadržaj cele matrice, vec samo onog dela koji
   se odnosi na koriscene neterminale */
void printR(RELATION* R)
{
    int i,j;
    for(i=0; i<max_net_code; i++)
    {
        for(j=0; j<max_net_code; j++)
            printf("%d ",R->rel[i][j]);
        printf("\n");
    }
}

/* Funkcija postavlja polje R->rel [i][j] na 1. Ako
   polje ne postoji, vrsi se realokacija memorije */
void setR(RELATION *R, int i, int j)
{
    /* m je veci od brojeva i i j */
    int m=i;
    if (j>m) m=j;

    if (m>=R->size)
        reallocateR(R,m+STEP);
    R->rel[i][j]=1;
}

```

```

/* Varsalovim algoritmom se konstruise tranzitivno
   zatvorenje relacije */
void warshallR(RELATION *R)
{
    int i,j,k;
    for (k=0; k<R->size; k++)
        for (i=0; i<R->size; i++)
            for (j=0; j<R->size; j++)
                if (R->rel[i][k] && R->rel[k][j])
                    R->rel[i][j]=1;
}

```

Oba rešenja koriste leksički analizator implementiran u lex-u.

```

%option noyywrap
%{
#include "y.tab.h"
%}
%%
"<"[a-zA-Z]+">" return NETERM;
[a-zA-Z]+      return TERM;
"->"          return ARROW;
";"           return SEMI;
"|"           return BAR;
[ \n\t]
%%

```