SRL から RL への翻訳機の実装

2017SE004 青柳裕樹 2017SE051 水野幹大 2017SE057 新美伊織
2017SE058 野橋祐人 2017SE086 竹市翔哉

1. 現状の把握

現在、フローチャートはプログラムの構造や、プロセスを表すことなどに使われている。

1.1.Reversible Flowchart

Revertsible Flowchart は、普通のフローチャートでは表すことができない可逆プログラミング言語の構造やプロセスを表すことができるフローチャートである。[1]
このフローチャートを説明、解析するために可逆プログラミング言語 SRL,RL が作られた。

1.2.SRL, RL

SRL(Structed Reversible Language)と RL(Unstructed Reversible Language)は、横山教授らによって作られたプログラミング言語[1]である。SRL は if 文や繰り返し文などから成る構造化プログラミング言語であり、RL はラベルや goto 文などを使うブロックのかたまりからなる非構造化プログラミング言語である。

1.3 SRL と RL の構文

SRL と RL の構文は BNF 記法で以下のように定義されている。

（SRL）

$p ::= b$

$b ::= a$ | if e then b else b fi e
| b b | from e do b loop b until e

（RL）

q ::= d^+

$d ::= l : k\ a^*\ j$

k ::= from l | fi e from l else l |$entry$

$j ::=$ goto l | if e goto l else l | $exit$

p が SRL のプログラム全体であり、q が RL のプログラム全体である。RL は d の列から成るため d^+ となっている。さらに、l はラベルである。
また、２つの言語で共通で使われる演算子と式の構文は以下のように定義されている。

$a ::= x \oplus = e \mid x[e] \oplus = e \mid push\ x\ x$
$\mid pop\ x\ x \mid skip$

$e ::= c \mid x \mid x[e] \mid e \otimes e \mid top\ x \mid empty\ x$

$\otimes ::= \oplus \mid * \mid / \mid = \mid < \mid > \mid <= \mid != $

$\oplus ::= + \mid - \mid \hat{}$

ここで、x は変数である。
SRL と RL は相互に翻訳可能であり、翻訳規則も示されている。しかし、実装はされていないというのが現状である。

2.問題の発見

現在、SRL から RL にプログラムを変換する翻訳規則は定義されているが、翻訳機が実装されていない。そのため、SRL から RL に変換したい場合、その都度考えなければいけないので手間が掛かり、変換ミスも起こる可能性がある。

3. 課題の設定

Ocaml 上で SRL から RL への翻訳機を実装する。

4. 解決案の立案

OCaml 上で実装に必要な関数を定義する。字句解析器は ocamllex を用いて、構文解析器は ocamlyacc を用いて実装する。
また、ocamllex と ocamlyacc を使うのに必要な .mll ファイルと.mly ファイルは自分たちで作成する。

5.解決策（代替案）の立案

.mll ファイルや.mly ファイルを自分たちで作成するのではなく、BNFC[2]という.mll ファイル、.mly ファイル、pretty-printer を自動生成するツールを用いて実装する。

6．実装

今回は、プログラミング言語の OCaml を用いて翻訳器の実装を行う。
実装は、
1．SRL と RL のデータ型の定義
2．SRL の字句解析器と構文解析器の実装
3．SRL から RL へ翻訳する関数の定義
4．pretty-print 関数とその他の関数の定義
5．トップレベルの関数の定義
(6．inversion に変換する関数の定義)
の順序で行った。
inversion に変換する関数は翻訳機の実装には必要ないが、時間があったため実装した。
inversion に変換する関数は翻訳する関数と実装の仕方は一緒のため、説明は省略する。
また、字句解析器、構文解析器の実装には ocamllex と ocamlyacc を用いた。

6.1 データ型の定義

まず、今回扱う SRL と RL のデータ型を 1.3 章の構文に従い、定義する。以下にそのプログラムを示す。

```
(*step operation and expressions*)
type var =Var of string
type oplus =
  | Plus
  | Minus
  | Caret
type otime =
  | OPlus of oplus
  | Time
  | Div
  | Equal
  | Less
  | Greater
  | Less_Eq
  | Greater_Eq
  | Not_Eq
type exp =
  | EConst of int
  | EVar of var
  | EIn of var * exp
  | ETime of exp * otime * exp
  | ETop of var
  | EEmpty of var
type step =
  | Plus_Eq of var * oplus * exp
  | Plus_In of var * exp * oplus * exp
  | Push of var * var
  | Pop of var * var
  | Skip
(*language SRL*)
type blk =
  | SStep of step
  | SIf of exp * blk * blk * exp
  | SCon of blk * blk
  | SFrom of exp * blk * blk * exp
type srl = SBLK of blk
(*language RL*)
type label = Label of string
type jump =
  | RGoto of label
  | RIf of exp * label * label
  | RExit
type from =
  | RFrom of label
  | RFi of exp * label * label
  | REntry
type rlblk =
```

```
  | RBlk of label * from * step list * jump
type rl = RLBLK of rlblk list
```

1.3 章の p が srl に、b が blk に、q が rl に、d が rlblk に、k が from に、j が jump に対応している。

演算子と式については、a が step に、e が exp に、⊗が otime に、⊕が oplus に、l が label に、x が var に対応している。

RL は非構造化プログラミング言語で、プログラム全体はラベルのついたブロック(rlblk)のかたまりから成るので、プログラム全体を表すには rlblk のリストで定義する必要があった。また、RL の構文の

$d ::= l : k\ a^*\ j$

の a^* は 0 個以上の a を表すため、これもリストで表す必要があった。

例えば SRL のプログラム、

from $x = 0$ do $skip$ loop x+= 1 until $x = 0$

をデータ型で表すと、

SFrom(ETime(EVar(Var"x"),
Equal, EConst 0), SStep Skip,
SStep (Plus_Eq (Var "x", Plus, EConst 1)),
ETime (EVar (Var "x"), Equal, EConst 10))

となる。

また、RL のプログラム、

l_0 : $entry$
    goto l_1
l_1 : from l_0
    $exit$

をデータ型で表すと、

RLBLK[RBlk(Label “l0”, REntry,
        [], RGoto(Label “l1”));
        RBlk(Label “l1”, RFrom(Label “l0”),
        [], RExit)]

となる。

6.2 字句解析器と構文解析器の実装

翻訳する関数を定義する際、入力が文字列では実装することができない。そのため与えられた文字列を 6.1 章で定義したデータ型に変換するために字句解析器と構文解析器を実装しなければならない。

字句解析器は、入力された文字列を字句の列にするものであり、構文解析器は、その字句の列を構文木にするものである。

今回は、それらを自動生成する ocamllex と ocamlyacc を利用する。ocamllex は、.mll ファイルという字句の規則を記述するファイルを受け取り、字句解析器を生成する。ocamlyacc は、.mly ファイルという構文の定義を記述するファイルを受け取り、構文解析器を自動生成する。また、.mll ファイルと.mly を自動生成する BNFC というツールがあるが、今回は手動で作成した。以下に.mll ファイルの一部を示す。

```
{
open Parser
}
rule token = parse

   (* 演算子 *)
   | '+'          { PLUS }
   | '-'          { MINUS }
   | '^'          { CARET }
   | '*'          { ASTERISK }
   | '/'          { SLASH }
   | '='          { EQUAL }
   | '<'          { LESS }
   | '>'          { GREATER }
   | "<="         { BELOW   }
   | ">="         { ABOVE }
   | "!="         { NOT }
```

今回、字句の定義は.mly ファイルに記述したため、最初の open parser で.mly ファイルを読み込んでいる。

以下に実行例を示す。

```
# let x = Lexing.from_string "from x = 0 do
skip;";;
# token x;;
- : Parser.token = FROM
# token x;;
- : Parser.token = VAR "x"
# token x;;
- : Parser.token = EQUAL
```

このように字句解析器は入力された文字列を先頭から読んでいき、.mll ファイルで定義された字句とマッチしたら.mll ファイルの中括弧内を返す。

次に.mly ファイルの一部を以下に示す。

```
// リテラル
%token <string> VAR    // x, y, abc, ...
%token <int> CONST    // 0, 1, 2, ...

// 演算子
%token PLUS          // '+'
%token MINUS         // '-'
%token CARET         // '^'
%token ASTERISK // '*'

              (省略)

%start main
%type <Syntax.blk> main

%%

// 開始記号
main:
   | blk SEMI
      { $1 }
;
                (省略)
blk:
   // step
   | step
      { SStep $1 }

   // if e1 then b1 else b2 fi e2
   | IF exp THEN blk ELSE blk FI exp
      { SIf($2, $4, $6, $8) }

   // b b
   | blk blk
      { SCon($1, $2) }

   // from e1 do b1 loop b2 until e2
   | FROM exp DO blk LOOP blk UNTIL
exp
      { SFrom($2, $4, $6, $8) }
```

%token の部分が字句の定義である。%start の部分は先頭記号を示しており、%type の部分は返す型を示している。そこから下の部分では構文の定義と評価規則を示している。つまり、”|”の右の部分とマッチする字句列があれば中括弧内を評価してそれを返す。$1 とすることで、字句列の 1 番目の値を参照できる。

以下に実行例に示す。

```
Parser.main    Lexer.token
(Lexing.from_string "from x=0 do skip loop
x+=1 until x=10;" );;
# - : Syntax.blk =
SFrom (ETime (EVar (Var "x"), Equal, EConst
0), SStep Skip,
  SStep (Plus_Eq (Var "x", Plus, EConst 1)),
  ETime (EVar (Var "x"), Equal, EConst 10))
```

.mly ファイル内で %start main としたため、main という関数が生成された。main は字句解析器を必要とするため、第一引数は Lexer.token となっている。

6.3 SRL から RL へ翻訳する関数の定義

SRL から RL に翻訳する関数を定義する。翻訳規則は以下のように定義される。

$$T_{SRL}[\![b]\!] =$$
$$l_0 : entry$$
$$\quad goto\ l_1$$
$$T[\![b_1]\!](l_0, l_1, l_2, l_3)$$
$$l_3 = \text{from}\ l_2$$
$$\quad exit$$

$$T[\![b_1 b_2]\!](l_0, l_1, l_4, l_5) =$$
$$T[\![b_1]\!](l_0, l_1, l_2, l_3)$$
$$T[\![b_2]\!](l_2, l_3, l_4, l_5)$$

$$T[\![a]\!](l_0, l_1, l_2, l_3) =$$
$$l_1 : \text{from}\ l_0$$
$$\quad a$$
$$\quad goto\ l_2$$
$$l_2 : \text{from}\ l_0$$
$$\quad \text{goto}\ l_3$$

$$T\left[\!\!\left[\begin{array}{l} \text{if}\ e_1\ \text{then}\ b_1 \\ \text{else}\ b_2\ \text{fi}\ e_2 \end{array} \right]\!\!\right](l_0, l_1, l_6, l_7) =$$

l_1 : from l_0

if e_1 goto l_2 else l_4

$T[\![b_1]\!](l_1, l_2, l_3, l_6)$

$T[\![b_2]\!](l_1, l_4, l_5, l_6)$

l_6 : fi e_2 from l_3 else l_5

goto l_7

$$T\left[\!\!\left[\begin{array}{l}\text{from } e_1 \text{ do } b_1\\ \text{loop } b_2 \text{ until } e_2\end{array}\right]\!\!\right](l_0, l_1, l_4, l_7) =$$

l_1 : fi e_1 from l_0 else l_6

goto l_2

$T[\![b_1]\!](l_1, l_2, l_3, l_4)$

$T[\![b_2]\!](l_4, l_5, l_6, l_1)$

l_6 : from l_3

if e_2 goto l_7 else l_5

これらの規則を OCaml のプログラムで実装する。以下にそのプログラムを示す。

```
open Syntax
let rec srlc b =
  let max2 a b =
    if (int_of_string a)>(int_of_string b) then a
else b
  in
  let max4 a b c d =
    if (max2 a b) > (max2 c d) then (max2 a b)
else (max2 c d)
  in
  let pl n x =
    string_of_int((int_of_string   x)
                  + (int_of_string n))
  in
  let rec srlcc c (la,lb,lc,ld) =
    let x = max4 la lb lc ld
    in
    match c with
    | SCon(b1,b2) ->
        (srlcc b1 (la,lb,pl x la,pl x lb))
        @(srlcc b2 (pl x la,pl x lb,lc,ld))
    | SStep(a) ->
        [RBlk(Label lb,RFrom(Label la),[a],
              RGoto(Label lc))]
        @[RBlk(Label lc,RFrom
          (Label lb),[],RGoto(Label ld))]
    | SIf(e1,b1,b2,e2) ->
        [RBlk(Labellb,
         RFrom(Label la),[],
         RIf(e1,Label (pl x la),Label (pl x lc)))]
        @(srlcc b1 (lb,pl x la,pl x lb,lc))
        @(srlcc b2 (lb,pl x lc,pl x ld,lc))
        @[RBlk(Label lc,
            RFi(e2,Label (pl x lb),
            Label (pl x ld)),[],RGoto(Label ld))]

              (省略)
  in
  RLBLK([RBlk(Label"1",REntry,
            [],RGoto(Label "2"))]
         @(srlcc b ("1","2","3","4"))
         @[RBlk(Label "4",
            RFrom(Label "3"),[],RExit)])
```

関数 srlc が翻訳規則の T_{SRL} に、関数 srlcc が T に対応している。関数 srlcc では、再帰呼び出しと、演算子"@"でのリストの連結を繰り返すことによりリストを返すことができる。また、関数 srlcc で新しい RL のブロックが生成される際は、ラベルの値を他のブロックと異なる値にしなければならないので、関数 max2、max4、pl を用いて異なるらべるを割り当てた。

以下に実行例を示す。

```
#srlc(SFrom (ETime (EVar (Var "x"),
       Equal, EConst 0), SStep Skip,
 SStep (Plus_Eq (Var "x", Plus, EConst 1)),
 ETime (EVar (Var "x"), Equal, EConst 10)));;
  - : Syntax.rl =
RLBLK
 [RBlk (Label "1", REntry, [], RGoto (Label
"2"));
  RBlk (Label "2",
   RFi (ETime (EVar (Var "x"),
         Equal, EConst 0), Label "1", Label "8"),
         [], RGoto (Label "5"));
  RBlk (Label "5", RFrom (Label "2"),
         [Skip], RGoto (Label "6"));
  RBlk (Label "6", RFrom (Label "5"),
        [], RGoto (Label "3"));
  RBlk (Label "7", RFrom (Label "3"),
        [Plus_Eq (Var "x", Plus, EConst 1)],
   RGoto (Label "8"));
  RBlk (Label "8", RFrom (Label "7"),
        [], RGoto (Label "2"));
  RBlk (Label "3", RFrom (Label "6"), [],
  RIf (ETime (EVar (Var "x"),
       Equal, EConst 10), Label "4", Label "7"));
  RBlk (Label "4", RFrom (Label "3"),
        [], RExit)]
```

6.4 pretty-print 関数の定義

翻訳した後のデータ型で表された RL のプログラムをそのまま表示するのは読みにくいため、綺麗な文字列で返すための関数を定義する。

```
open Syntax
(*step operation and expressions Pretty Printer*)
let print_v = function
  | Var(s) -> s

let print_op = function
  | Plus -> "+"
  | Minus -> "-"
  | Caret -> "^"

let print_ot = function
  | OPlus(n) -> print_op n
  | Time -> "*"
  | Div -> "/"
  | Equal -> "="
  | Less -> "<"
  | Greater -> ">"
  | Less_Eq -> "<="
  | Greater_Eq -> ">="
  | Not_Eq -> "!="

let rec print_e = function
  | EConst(n) -> string_of_int n
  | EVar(x) -> print_v x
  | EIn(x,e) -> (print_v x) ^ "[" ^
   (print_e e) ^ "]"
  | ETime(e1,ot,e2) ->
   (print_e e1) ^ " " ^(print_ot ot) ^
    " " ^ (print_e e2)
  | ETop(x) -> "top " ^ (print_v x)
  | EEmpty(x) -> "empty " ^ (print_v x)

let print_a = function
  | Plus_Eq(x,op,e) -> (print_v x) ^ " " ^
     (print_op op) ^ "= " ^ (print_e e)
  | Plus_In(x,e1,op,e2) ->
     (print_v x) ^ "[" ^ (print_e e1) ^ "] " ^
     (print_op op) ^ "= " ^ (print_e e2)
  | Push(x1,x2) -> "push " ^ (print_v x1) ^ " " ^
                   (print_v x2)
  | Pop(x1,x2) -> "pop " ^ (print_v x1) ^ " " ^
                   (print_v x2)
  | Skip -> "skip"
```

```
(*RL Pretty Printer*)

let rec rl_prints =
let rec rl_print =
let print_l = function
  | Label(s) -> s
in
let rec print_a2 = function
  | [] -> ""
  | [x] -> (print_a x) ^ "¥n¥t"
  | x::xs -> (print_a x) ^ "¥n" ^ (print_a2 xs) ^
"¥n¥t"
in
let print_j = function
  | RGoto(l) -> "goto " ^ (print_l l)
  | RIf(e,l1,l2) ->
      "if " ^ (print_e e) ^ " goto " ^ (print_l l1) ^
" else " ^ (print_l l2)
  | RExit -> "exit"
in
let print_k = function
  | RFrom(l) -> "from " ^ (print_l l)
  | RFi(e,l1,l2) ->
      "fi " ^ (print_e e) ^ " from " ^ (print_l l1) ^
" else " ^ (print_l l2)
  | REntry -> "entry"
in
let print_blk = function
  | RBlk(l,k,a,j) ->
      (print_l l) ^ ": ¥t" ^ (print_k k) ^ "¥n¥t" ^
(print_a2 a)  ^ (print_j j)
in
function
| [] -> ""
| [x] -> print_blk x
| x :: xs -> ((print_blk x) ^ "¥n" ^ (rl_print xs))
in
function
| RLBLK(rlblk) -> print_string ((rl_print rlblk)
^ "¥n")
```

文字列を連結する演算子 ”^” を使い、再帰呼び出しをすることで文字列を返し実装した。

しかし、これらの関数では RL のプログラムのラベルの値がばらばらで、プリントされるものが見にくいものとなってしまう。そのため、ラベルの値をブロックごとに 1 から順番につけ直す関数 opt_rl を作成した。次ページにその関数を示す。

```
open Syntax
let opt_rl1 =
let rec opt rl c =
  match rl with
  | [] -> (-1,-1)
  | x::xs ->
      begin
        match x with
        | RBlk(Label(n),f,s,j) ->
            begin
              let a = try int_of_string n with
                          | _-> -1
              in
              if (c = a) || (a = -1) then opt xs
(c+1)
              else (int_of_string n,c)
              end
      end
in
function
| RLBLK(rl) -> opt rl 1

let opt3 rl n =
let rec opt2 rl (c1,c2) =
  let opt_l l (c1,c2) =
    match l with
    | Label(n)->
        let a = try int_of_string n with
                  | _-> -1
        in
        if a = c1 then Label("l"^(string_of_int
c2)) else Label(n)
  in
  let opt_f f (c1,c2) =
    match f with
    | RFrom(l) -> RFrom(opt_l l (c1,c2))
    | RFi(e,l1,l2) -> RFi(e, opt_l l1 (c1,c2),
opt_l l2 (c1,c2))
    | REntry -> REntry
  in
  let opt_j j (c1,c2) =
    match j with
    | RGoto(l) -> RGoto(opt_l l (c1,c2))
    | RIf(e,l1,l2) -> RIf(e, opt_l l1 (c1,c2), opt_l
l2 (c1,c2))
    | RExit -> RExit
  in
  match rl with
  | [] -> []
  | x::xs ->
      begin
        match x with
        | RBlk(l,f,s,j) ->
            [RBlk(opt_l l (c1,c2), opt_f f (c1,c2),
s, opt_j j (c1,c2))]
            @ (opt2 xs (c1,c2))
      end
in
match rl with
| RLBLK(x) -> RLBLK(opt2 x n)

let rec opt_rl rl =
  let x = opt_rl1 rl
  in
  if x = (-1,-1) then rl
  else opt_rl(opt3 rl x)
```

まず、関数 opt は RL のブロックのリストと整数 c を受け取り、int 型のペアを返す関数である。リストの先頭の要素を取り出し、ラベルの値が整数 c と等しければ c の値を 1 増やし、再帰呼び出しをする。等しくなければその時点のラベルの値と整数 c を返す。そして、リストが空の場合(-1,-1)を返す。この関数の引数 c に 1 を適用させることで、順番になっていないラベルの値と、正しいラベルの値のペアを返すことができる。そして、すべてのラベルの値が 1 から順番になっていれば、(-1,-1)を返す。また、try with 構文は他の関数と組み合わせる際に例外が発生してしまうため必要であった。

関数 opt2 は RL のブロックのリストと整数のペアを受け取り、RL のブロックのリストを返す関数である。つまり、受け取った整数のペアの 1 つ目の値と同じラベルの値を整数のペアの 2 つ目の値に置き換えたリストを返す関数である。また、ラベルの値を置き換えるために RL の構文ごとに関数を定義する必要があった (opt_l, opt_f など)。さらに、一度置き換えたラベルの値が変更されないようにラベルの始めに”l”をつける必要があった。

そして、関数 opt_rl1 と opt2 を組み合わせることにより、受け取った RL のリストの

ラベルの値が 1 から順番になるまで置換を繰り返す関数 opt_rl を定義した。opt_rl1 から(-1,-1)が帰ってくるまで置換を繰り返すことにより全てのラベルの値が等しくなる。これらの関数の他に、より見やすくなるように、ラベルの値に”l”がついてないラベルに”l”をつける関数も定義したがここでは省略する。

6.5 トップレベルの関数の定義

これまでに定義した関数を組み合わせて、入力された SRL プログラムの文字列を RL に翻訳し、出力するというトップレベルの関数 read_print を定義した。以下がその関数である。

```
let rec read_print () =
   let srl2rl s = rl_prints(label_rl(opt_rl(srlc s)))
   in
   let rl_inver s =
      rl_prints(label_rl(opt_rl(invertr(srlc s))))
   in
   let srl_inver s = srl_prints(invert s)
   in
   print_string "->";
   flush stdout;
   let line =
      Parser.main Lexer.token
       (Lexing.from_channel stdin )
   in
   print_string("¥nSRL:¥n" ^ srl_prints(line) ^
"¥n¥n");
   print_string("SRL inversion:¥n" ^ (srl_inver
line) ^ "¥n¥n");
   print_string("RL:¥n");
   srl2rl line;
   print_string "¥n";
   print_string("RL inversion:¥n");
   rl_inver line;
```

この関数は、まず、入力された SRL をそのまま出力し、次にそのプログラムの inversion、RL のプログラム、そのプログラムの inversion を出力する。

以下に 2 つの実行例を示す。

実行例 1

```
->from x=0 do skip loop x +=1 until x = 10;
SRL:
from x = 0 do skip loop x += 1 until x = 10

SRL inversion:
from x = 10 do skip loop x -= 1 until x = 0

RL:
l1:     entry
        goto l2
l2:     fi x = 0 from l1 else l6
        goto l3
l3:     from l2
        skip
        goto l4
l4:     from l3
        goto l7
l5:     from l7
        x += 1
        goto l6
l6:     from l5
        goto l2
l7:     from l4
        if x = 10 goto l8 else l5
l8:     from l7
        exit

RL inversion:
l1:     entry
        goto l2
l2:     fi x = 10 from l1 else l4
        goto l5
l3:     from l7
        goto l4
l4:     from l3
        x -= 1
        goto l2
l5:     from l2
        goto l6
l6:     from l5
        skip
        goto l7
l7:     from l6
        if x = 0 goto l8 else l3
l8:     from l7
        exit
```

実行例 2

```
->from k = n do skip loop k -= 1 from j = 0 do skip loop
if x[j] > x[k] then x[j] -= 1 else skip fi x[j] >= x[k]   j
+= 1 until j = k   j -= k until k = 0;

SRL:
from k = n do skip loop k -= 1 from j = 0 do skip loop
if x[j] > x[k] then x[j] -= 1 else skip fi x[j] >= x[k]   j
+= 1 until j = k   j -= k until k = 0

RL:
l1:     entry
        goto l2
l2:     fi k = n from l1 else l20
        goto l3
l3:     from l2
        skip
        goto l4
l4:     from l3
        goto l21
```

```
l5:     from l21
        k -= 1
        goto l6
l6:     from l5
        goto l7
l7:     fi j = 0 from l6 else l17
        goto l8
l8:     from l7
        skip
        goto l9
l9:     from l8
        goto l18
l10:    from l18
        if x[j] > x[k] goto l11 else l13
l11:    from l10
        x[j] -= 1
        goto l12
l12:    from l11
        goto l15
l13:    from l10
        skip
        goto l14
l14:    from l13
        goto l15
l15:    fi x[j] >= x[k] from l12 else l14
        goto l16
l16:    from l15
        j += 1
        goto l17
l17:    from l16
        goto l7
l18:    from l9
        if j = k goto l19 else l10
l19:    from l18
        j -= k
        goto l20
l20:    from l19
        goto l2
l21:    from l4
        if k = 0 goto l22 else l5
l22:    from l21
        exit
```

２つ目の実行例は長くなるため、inversion の部分は省略した。

7. まとめ

今回は SRL から RL への翻訳機を実装し、Inverter も実装することもできた。また、出力も綺麗にすることができた。
今後の課題としては、プログラムに”skip”があった場合は省略するように出力したり、RL から SRL への翻訳機を実装したりすることが考えられる。

8. 参考文献

[1]
「[YoAG16] Yokoyama, T., Axelsen, H.B., and Glück, R.: Fundamentals of reversible flowchart languages, Theoretical Computer Science, Vol.611, pp.87-115,」（2016）
<http://owari.se.nanzan-u.private/seminar2019/191128/YoAG16-TCS.pdf>
[2]
「The BNF Converter」
<https://bnfc.digitalgrammars.com/>
[3]
亀山幸義「Functional Programming」
<http://www.logic.cs.tsukuba.ac.jp/jikken/index.html>

役割分担

レポート作成：青柳、新美、竹市
発表資料作成：水野、野端

実装
データ型、関数定義：新美
BNFC：竹市