言語処理系の生成系MYLANGのための コード生成の研究

（昭和58年11月30日　原稿受付）

情報工学教室　大庭英子
同　山之上卓
同　安在弘幸

A Study of Code Generation for the Language Processor Generator MYLANG

by Eiko OHBA
Takashi YAMANOUE
Hiroyuki ANZAI

Abstract

This paper presents a study of code generation and its optimization concerned with functional extension of our language processor generator MYLANG. This paper includes two topics. One is a problem of register assignment to an arithmetic expression. The other is a problem of code generation from a labeled automaton which is constructed by our MYLANG system.

For the optimal assignment of registers to a given arithmetic expression, this paper introduces and modifies Sethi and Ullman's algorithm which decreases load and store instructions and shortens the execution time of an object code transformed from the arithmetic expression. And this paper shows how to generate the object code from the labeled automaton, and gives some examples.

1. まえがき

我々の研究室では，言語処理系の自動生成システムMYLANGを開発した[1]。このシステムは，まず処理言語の構文と意味の記述としての属性付RTFと活動ルーチンを入力して，(属性付) オートマトンを生成，合成し，かつその簡約決定性変換を行い，次に，このオートマトンをドライバールーチンに結合して目的の言語処理系とする[1]。そして更に，こうして得られた言語処理系も上記と同様にして，原プログラムをオートマトンに変換し，これをドライバールーチンに結合して目的の処理システムとする。

この種の処理システムは，一種のインタプリタであり，その実行速度が遅いことはまぬかれない。これを改善する1つの方法は，中間形式としてのオートマトンから目的コードを，出来るだけ最適化させて，生成させることである。

以上の観点から，我々はMYLANGシステムの機能の増強の1つの問題として，コード生成とその最適化について研究した。本稿はその報告である。

コンパイラは，まず原プログラムを中間形式へ翻訳し，続いて実際の機械のための目的コードを生成する。A. V. Ahoと S. C. Johnsonは実際の計算機に有効なコードを生成するためには，次をうまく決めることが必要であると述べている[2]。すなわち，(1)どの命令を使うか，(2)どの順序で実行するか，(3)どの中間結果を一時記憶領域に格納するかである。一方，A. V. Ahoと Ravi Sethi[3]は，コード生成の理論における基本問題を次のようにまとめている。

（1） 自動コード生成
（2） 解析木からのコード生成
（3） 共通の部分木
（4） 制御の流れにおけるコード生成

解析木からのコード生成の問題については，R. Sethi

と J. D. Ullman[4] や 中田[5] らの研究がある。R. Sethi と J. D. Ullman[4] は解析木に古典的なラベル付けを行い，中間結果の一時的な格納なしで計算するために必要なレジスタの最小数を求めることから始めている。

共通式におけるコード生成は，A. V. Aho と S. C. Johnson[2], A. V. Aho, S. C. Johnson と J. D. Ullman[6], B. Prabhara と R. Sethi[7] および Reiner Güttler[8] によって研究された。

本稿では，算術式の最適化の問題および MYLANG システムの出力としてのラベル付オートマトンからのコード生成の問題の2つについて取扱う。算術式の最適化については，Sethi と Ullman[4] のアルゴリズムを用いる。算術式における中間形式は三つ組とし，三つ組からコードへ変換するようにしている。従って，中間形式が三つ組の場合のコード生成とラベル付オートマトンの場合のコード生成は互いに独立なものとして取扱っている。将来，この2通りの理論と実際をうまく組み合わせ，更に有効なコード生成系を作っていきたい。

2章では，算術式を三つ組に変換した後でこれをコードへ変換する時の，レジスタの割付けの最適化について述べる。3章では，MYLANG システムの中間形式およびラベル付オートマトンによるコード生成について論じる。

2. レジスタの最適割付け

目的プログラムの最適化においては，LOAD/STORE 命令の回数を最小にすることが重要である。このことは，値のメモリー参照の回数を減らすことで，プログラムのステップ数を最小にすることを意味する。

演算レジスタが1個の場合はスタック機械と対応させて考えることができる[9]。一般に計算機は複数個の演算レジスタをもっているので，ここでは複数個の場合について述べる。

2.1. 三つ組と解析木

中間言語は三つ組 (triple) として出力されるとする。三つ組は

(op, oprand1, oprand2)

と表される。op は演算子，oprand1 は第1オペランド，そして oprand2 は第2オペランドである。oprand1 と oprand2 は，外部で定義された値 (initial と呼ぶ) か，または他の演算によって定義された値（中間値と呼ぶ）のどちらかである。一方，op を節，oprand1 を左の子供（枝ともいう），oprand2 を右の子供と考えれば，この三つ組は解析木（2分木）とみなすことができる。必要なレジスタ数を最小にするような最適化を考えるのに，解析木の方が取扱いやすいため，以下，解析木からの最適化を論じることにする。ここで，initial は葉に，中間値は節にそれぞれ対応している。

図-1 は算術代入文の三つ組と解析木の例を示している。ここで，T1, T2, T3 は演算途中の結果を保持する中間値である。

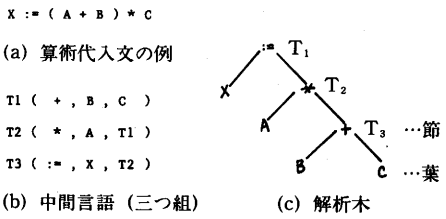


図-1 中間言語と構文解析木

なお，解析木が図-2.(a) のような場合，三つ組を生成する際に，図-2.(b) のように変換されているものとする。従って，以降2分木についてのみ考える。

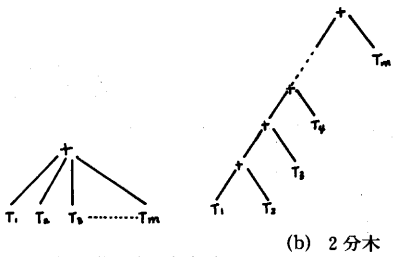


図-2 解析木と2分木

2.2. 使用する命令コード

本稿の中で使用する表現および命令コードを次のように約束する。

N: 使用可能なレジスタの個数

$\gamma_1, \gamma_2, \ldots, \gamma_N$: レジスタ名

m_i: メモリ（番地は i)

$\gamma_i \leftarrow m_j$: ロード

$m_i \leftarrow \gamma_j$: ストア

$\gamma_i \leftarrow \gamma_j$ op m_k あるいは $\gamma_i \leftarrow \gamma_j$ op γ_k: 演算

(注) $\gamma_i \leftarrow m_k$ op γ_j は許されていない。

演算子 op は四則演算＋，－，＊，／

命令コード

LOAD R_i, M_j $(R_i \leftarrow M_j)$

STORE R_i, M_j　　($M_j \leftarrow R_i$)

ADD: 加法

SUB: 減法

MULT: 乗法

DIV: 除法

2.3. ラベル付け

解析木の各節は，その節以下の枝を計算するために必要なレジスタ数を情報としてもっておく。節 η についてこの情報をラベルと呼び，$l(\eta)$ と表わす。

ラベルを付ける順序は上向き（枝から根，bottom-up）である。

(i) η が葉の場合

(a) η の親から η を見て，η が左側の子であるならば　　$l(\eta)=1$

(b) それ以外　　$l(\eta)=0$

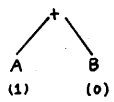


図-3　式 A+B のラベル付け（　）の中の数字がラベル

(ii) η が葉でなく，ラベルが l_1, l_2 である子をもつ場合

(a) $l_1 \neq l_2$ のとき　$l(\eta)=\max(l_1, l_2)$

(b) $l_1=l_2$ のとき　$l(\eta)=l_1+1$

次に，ラベル付けの例を示す。

例　A/(B+C)−D*(E+F)

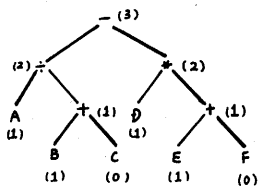
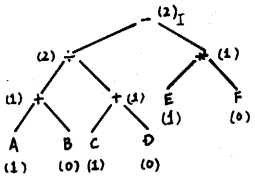


図-4　式 A/(B+C)−D*(E+F) のラベル付け
（　）の中の数字はラベル

このラベルは，左右の枝をどちらから先に計算した方がよいかを判断するのに使う。必要なレジスタ数が大きい方の枝から先に計算した方が STORE の回数が少ない。ここでは例をあげるだけでその証明は省略する[4]。

ラベル付けした後，生成されたコードをそれぞれ比較する。最初に，使用可能なレジスタは2個と断わっておく。また，コード変換の方法については後で示すアルゴリズムに従う。図-5.(a) の節 I においてラベルの小さい方からコードへ変換すると，図-5.(b) の 5 の STORE 命令のように値を一度メモリーへ格納しなければならない。そこでラベルの大きい方からコードへ変換する。すなわち，二項演算を行う順序を変えると，図-5.(c) のようにプログラムは1ステップ短くなる。

算術式　(A+B)/(C+D)-E*F



(a) 解析木
（　）の中の数字はラベル

1	LOAD	R1,E		LOAD	R1,C
2	MULT	R1,F		ADD	R1,D
3	LOAD	R2,C		LOAD	R2,A
4	ADD	R2,D		ADD	R2,B
5	STORE	R1,W	(W:memory)	DIV	R2,R1
6	LOAD	R1,A		LOAD	R1,E
7	ADD	R1,B		MULT	R1,F
8	DIV	R2,R1		SUB	R2,R1
9	SUB	R2,W			

(b) ラベルの小さい方からのコード生成　　(c) ラベルの大きい方からのコード生成

図-5　二項演算の順序によって異なるコード生成

2.4. フリッピング (flipping)

節 η のラベル $l(\eta)>1$ でかつ，左の子 η_1 が葉である時，もし演算子が可換性であれば，右と左の子供を交換すると，$l(\eta_1)$ を1から0へ縮小することができる。この交換のことを，Ravi Sethi と J.D. Ullman はフリッピング (flipping) と呼んでいる[4]。

図-4の解析木を flipping すると，図-6のようになる。

図-4のラベル付けでは，レジスタは3個必要であったが，flipping することで2個ですむようになった。

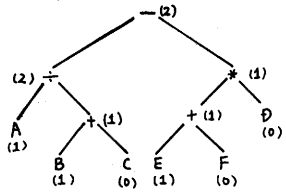


図-6　図-4の解析木の flipping 後のラベル付け

flipping 後，もう一度ラベル付けしなおして，次のコード生成へ進む。

2.5. コード生成アルゴリズム

CODE(η)=CODE(L op R)

$\gamma_i, \gamma_{i+1}, \ldots, \gamma_N$: 使用可能なレジスタ

l:η のラベル

$l=1$

(A) η が葉の場合　$\gamma_i \leftarrow \eta$

(B) η が葉でない場合

もちろん，右の子供 R は葉である。

左の子供 L について，　$\gamma_i \leftarrow$CODE(L)

$\gamma_i \leftarrow \gamma_i$ op R

$l>1$

η の左右の子供はそれぞれ，ラベル l_1, l_2 をもつする。

(C) $l_1, l_2 \geqq$N の場合

$m_j \leftarrow$CODE(R); $\gamma_i \leftarrow$CODE(L);

$\gamma_i \leftarrow \gamma_i$ op m_j;

(D) $l_1 \neq l_2$ かつ，少なくとも一方は N より小さい場合

$\gamma_j \leftarrow$CODE(MAX(l_1, l_2));

$\gamma_{i+1} \leftarrow$CODE(MIN(l_1, l_2));

$\gamma_i \leftarrow \gamma_i$ op γ_{i+1};　あるいは

$\gamma_{i+1} \leftarrow \gamma_{i+1}$ op γ_i;

(注) MAX(l_1, l_2) とは，l_1 と l_2 を比較して，大きいラベルをもつ子供のことを表す。

それに対し，MIN(l_1, l_2) は，小さいラベルをもつ子供のことである。

(E) $l_1=l_2<$N の場合

$\gamma_i \leftarrow$CODE(L)

あとは (D) と同様に行う。

このアルゴリズムへの入力は解析木である。我々はこのアルゴリズムを三つ組用に変形した。そこで問題となったのは，オペランドが initial か中間値かを区別することである。ここでは，両者を区別するために，各オペランドにフラグを付けることにする。三つ組は図-7 のような形で入力される。

(演算子，フラグ1，オペランド1，フラグ2，オペランド2)

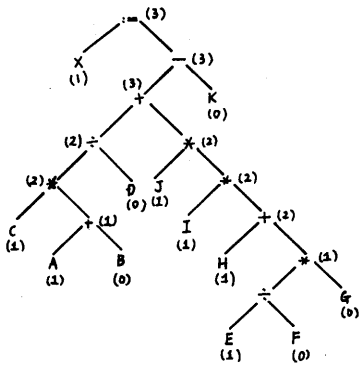
```
T1 (  +  , 0 , A  , 0 , B  )

T2 (  *  , 1 , T1 , 0 , C  )

T3 ( :=  , 0 , X  , 1 , T2 )
```

図-7　X:=(A+B)*C についての三つ組
フラグが0ならばそのオペランドは initial
フラグが1ならばそのオペランドは中間値

フラグ1はオペランド1に関して，フラグ2はオペランド2に関して，それぞれ initial か中間値かを判別するフラグである。また，左端の T1, T2 および T3 は演算の順序を示しており，演算の中間結果を記憶する役割もしている。



(a) flipping する前のラベルをもつ解析木

算術式　X:=C*(A+B)/D+J*I*(H+E/F*G)-K

(b) 三つ組

```
(   +,  0,A ,  0,B   )
(   *,  0,C ,  1,T1  )
(   /,  1,T2,  0,D   )
(   /,  0,E ,  0,F   )
(   *,  1,T4,  0,G   )
(   +,  0,H ,  1,T5  )
(   *,  0,I ,  1,T6  )
(   *,  0,J ,  1,T7  )
(   +,  1,T3,  1,T8  )
(   -,  1,T9,  0,K   )
(  :=,  0,X ,  1,T10 )
```

(c) コード

```
LOAD    R0,   A
ADD     R0,   B
MULT    R0,   C
DIV     R0,   D
LOAD    R1,   E
DIV     R1,   F
MULT    R1,   G
ADD     R1,   H
MULT    R1,   I
MULT    R1,   J
ADD     R0,   R1
SUB     R0,   K
STORE   R0,   X
```

図-8　X:=C*(A+B)/D+J*I+(H+E/F*G)−K のコード生成

2.6. 例　　題

次の算術式

X=C*(A+B)/D+J*I+(H+E/F*G)−K

について，解析木，三つ組およびコードを生成する。

図-8.(a) の算術式を入力する。図-8.(b) は図-8.(a) の算術式に対する flipping 前の解析木である。( ) の中の数字はラベルである。この例題で使用可能なレジスタは8個である。flipping 前の解析木で，この算術式を処理するのに必要なレジスタは3個であるから，この場合にはわざわざ flipping する必要はないかもしれない。ここでは，flipping の必要性については問わないことにして，flipping を行いコードを生成した。その結果が図-8.(c) の三つ組と図-8.(D) のコードとして表されている。

3. ラベル付オートマトンにおけるコード生成

MYLANG システムの中間形式はラベル付オートマトンである。この章では，このラベル付オートマトンが一体どのようなものであるかを簡単に説明し，次はどのようにしてこのオートマトンがコードへ変換されるかを示す。

3.1. ラベル付オートマトン

オートマトンは図-9の状態遷移図を用いて説明する。(α) は開始状態，(γ) は終状態，(β) は状態1から状態2へ記号 a で遷移することをそれぞれ表す。状態1から状態3で記号を入力し終われば，この入力記号列 ab はこのオートマトンによって受理されたという。

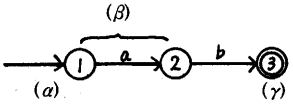


図-9　オートマトン

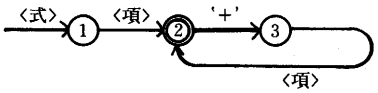


図-10　加法からなる算術式を受理するオートマトン

図-10は加法からなる算術式を受理するオートマトンの例である。＜と＞で囲まれた記号を非終端記号といい，'と' で囲まれた記号を終端記号という。非終端記号による遷移はその非終端記号に対応するオートマトンが挿入されることを表す。| と | で囲まれた記号を活動記号と呼び，遷移の段階で | と | の中の演算を実行する。状態遷移図上の終端記号，非終端記号および活動記号をまとめて名札と呼ぶ。その例を図-11に示す。

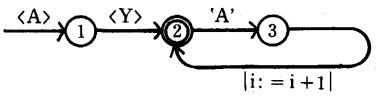


図-11　名札が活動記号であるオートマトンの例

ところで，状態1，2，…にそれぞれラベル1，2，…と割り付けることで，このオートマトンをラベル付オートマトンと呼んでいる。

3.2. システムのドライバー（マクロ命令）

MYLANG システムの構文解析系は，原プログラム

表-1　ドライバーのマクロ命令

オートマトンの名札	ラベルフィールド	コマンドフィールド	アーギュメントフィールド
'L'	LBL	PGOTO, R1	BRANCH, LBL(X)
'T'	LBL	PTERM, R1, R2, R3, R4, SR3	LINE, RLP, LRLP, SLINE, CLINE, ; BRANCH, TTABLE, BLK, INF1, INF2, ; INF3, INF4, INF5, X
'N'	LBL	PNONTERM, R1, R2	STACKS, LBL1, LBL(X)
'R'	LBL	PLAMBDA, R1, R2	BRANCH, STACKS
'X'	LBL	PERROR, R1, R2	BRANCH, STACKS
'B'	LBL	PBEG, R1	STACK1, NTABLE, X
'Q'	LBL	PQUI, R1, R2	STACKS, STACK1, NTABLE, X
'U'	LBL	PUSH, R1, R2, R3	STACK1, STACK2, X
'O'	LBL	POP, R1, R2	STACK1, STACK2, X
':'	LBL	ARITHM, R1, R2, R3, R4	STACK2, X
'P'	LBL	PERIPH, R1, R2	STACK2, X
'I'	LBL	IMED, R1, R2, R3	STACK2, X
@	LBL	CONDI, R1, R2, R3	BRANCH, STACK2, X

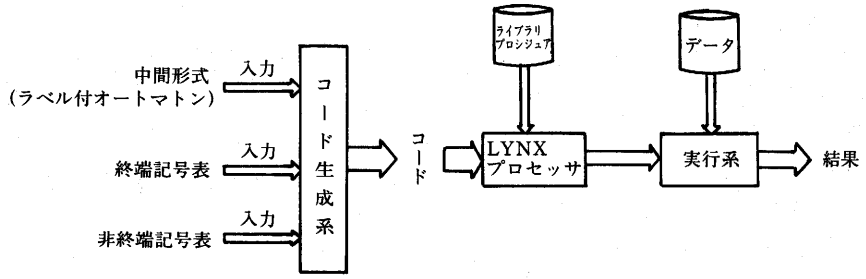


図-12 コード生成から実行までのシステムの概略図

をラベル付オートマトンに変換する。このラベル付オートマトンをドライバールーチンに結合して目的の処理システムとする。我々のコード生成系ではこのオートマトンを入力して，それに対応するマクロ命令に変換する。マクロ命令の記述や呼び出し方は表-1に示す通りである。

各マクロ命令においてコマンドフィールドとアーギュメントフィールドに書かれている要素の個数は多い。しかし，マクロ命令への引数の受け渡しのためにこのような形をとらざるをえない。

各マクロ命令の使用について以下のことに注意する。'X', 'LBL(X)', 'LBL1' の要素以外の要素についてはマクロ命令の一部としてそのまま使用する。使用者は変更することはできない。'X' はマクロ命令の中で使用される数値である。例えば中間形式が

' : 13'

であれば，

'ARITHM, R1, R2, R3, R4 stack2, 13'

というコードに変換される。ここで '13' は加法を表す。LBL(X), LBL1 は飛び先の番地である。その詳細は次の節で説明する。

コード生成から実行までを行うシステムの概略を図-12に示す。MYLANG システムの構文解析系は，中間形式，終端記号表および非終端記号表を出力する。コード生成系はこの3つを入力して実行可能なコードに変換する。LYNX プロセッサは，コードすなわち目的プログラムと別に用意しているライブラリプロシジュアとをリンクしてロードモジュールを組み立て，実行系に渡し，そこで実行させる。

3.3. 絶対番地の付け方

中間形式はすべてラベルをもっている†。このラベルをプログラムの記号番地に改める。コード変換の段階ではまだ記号番地に絶対番地は割り当てられていない。(記号番地は未定義である。) プログラム中の現在の状態を保持するカウンタをロケーションカウンタ（以後，LC と称する）と呼ぶ。中間形式が1個ずつ実行される毎に，その記号番地が LC にセットされる。ジャンプ命令を実行した後，飛び先の番地が次の LC にセットされる。そのような場合を除いて，LC は通常1ずつインクリメントされる。従って，記号番地すべてに絶対番地を割り付けねばならないわけではない。

図-13でジャンプ命令による実行の様子を簡単に述べる。LC の内容は '3' から 'NEXT' へ変化する。絶対番地 '3' の命令の実行後，次は絶対番地 'NEXT' が割り付けられている命令を実行する。

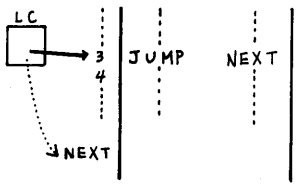


図-13 JUMP 命令によるプログラムの流れ

マクロ命令の 'P GOTO' には条件付ブランチ命令が，'P NONTERM' にはサブルーチンのコール命令が，それぞれ含まれており，'P ERROR', 'PQUI' 及び 'P LAMBDA' にはリターン命令が含まれている。これらのマクロ命令について更に説明を加える。

'L' に対応するマクロ命令は 'P GOTO' である。状態遷移図で名札 'L' がどのような処理をするかを図-14で示す。'L' ではそこにある条件式を満足するかどうかで遷移先が違う。図-14では 'L' の条件式が 'false' ならば記号番地1から2へ，'true' ならば1から100へ分岐する。つまり，'L' に含まれる条件式が 'true' ならば LC に記号番地 100をセットする。'false' ならば自動的に 2が

† 2. 3 の'ラベル' の意味とは違う。

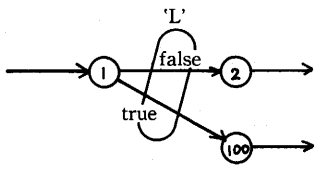


図-14　名札が 'L' の場合のオートマトン
'L' の中に含まれる条件が true の時 100 へ遷移する

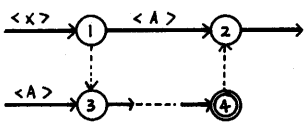


図-15　名札が非終端記号の場合のオートマトン

セットされる。このようにすると，マクロ命令 'PGOTO' では，飛び先の記号番地に絶対番地を割り付けておかなければならない。

'N' に対応するマクロ命令は 'P NONTERM' である。'X', 'Q' および 'R' にはそれぞれマクロ命令 'PERROR', 'P QUI' および 'P LAMBDA' が対応している。図-15 の状態遷移図でプログラムの流れをみる。〈X〉で始まる状態遷移図は状態 1 から非終端記号〈A〉により状態 2 へ遷移する。しかし，この遷移は仮想的な状態遷移で，実際は〈A〉を名前としてもつ状態遷移を呼び出している。この状態の遷移をたどって終状態 4 にたどりつくと，状態 2 へ戻る。この呼び出しや戻りの様子は通常のプログラムにおけるいわゆるサブルーチン・コール命令とリターン命令にそれぞれ対応している。LC を記憶するスタックを設け，コール命令で LC の内容をスタックにプッシュし，リターン命令でスタックからポップして LC にセットする。この方式では，サブルーチンの開始番地とサブルーチンから戻る時の戻り先の番地が必要であり，従ってそれぞれに対応する記号番地に絶対番地を割り付けるようにする。

条件付ブランチ命令およびサブルーチンのコール命令とリターン命令のために必要な記号番地に絶対番地を割り付けるために，我々のコード生成系は 2-パスで処理を行う。

3.4. 名札が活動記号である場合

オートマトンの名札が活動記号（ここでは算術式と論理式に限る）である場合，MYLANG システムはどのような方法で処理しているかについて述べる。

3.4.1. 算 術 式

名札が算術式である場合，それは中間形式としての逆ポーランド記法に変換される。逆ポーランド記法の例を図-16に示す。逆ポーランド記法の処理にはスタック機

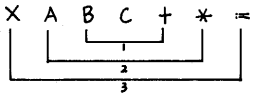


図-16　X:=A*(B+C) の逆ポーランド記法
数字は演算の順序を示す

```
opnd :  PUSH(opnd) ;
opr2 :  POP(y) ; POP(x) ;
        PUSH( x opr2 y ) ;
```

図-17　スタック機械

械が適している。二項演算のスタック機械は図-17の通りである。スタックを 2 回ポップして，2 個の項を得，それに演算を施し，その結果をプッシュする。

3.4.2. 論 理 式

例えば，IF＜条件式＞THEN…のような場合，オートマトンを生成する時，遷移は比較判定を行う遷移と条件ジャンプを行う遷移の 2 つに分けられる。図-18でそれを示す。条件ジャンプを行う遷移は，名札を 'A' と '¬A' をもつ部分である。この遷移の一つ前の遷移で比較判定を行っている。その比較条件が true か false かに応じてフラグをセットする。このフラグが true ならば名札 'A' false ならば名札 '¬A' に遷移する。この条件ジャンプは前節の 'L' に対応する。

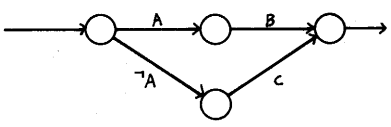


図-18　if A then B else C に対応するオートマトン

3.5. 実 行 結 果

例として，文脈依存言語 $A^n B^n C^n$, $n \geqq 0$ の認識系を図-19に示す。

```
(*?RTF*)
  <S>    =!(I:=0)!('A'!(I:=I+1))*
         !(J:=0)!('B'!(J:=J+1))*
         !(K:=0)!('C'!(K:=K+1))*
         <T(I,J,K/)>=! ;
  <T(I,J,K/)>=!?(I=J)!!?(I=K)!<DMM> ;
  <DMM>=!DMM1 ;
(*?END*)
(*?ACTION*)
  (*!DMM1*)
    BEGIN WRITELN END ;
(*?END*)
```

図-19 (a)　RTF とアクションルーチンの入力

```
TTABLE  TEXTC C'A'
        TEXT  C'      '
        TEXTC C'B'
        TEXT  C'      '
        TEXTC C'C'
        TEXT  C'      '
NTABLE  TEXTC C'S'
        TEXT  C'      '
        DATA            0
        DATA            3
        TEXTC C'T'
        TEXT  C'      '
        DATA            0
        DATA            3
        TEXTC C'DMM'
        TEXT  C'      '
        DATA            0
        DATA            0
START   EQU     $
        LI,R1   81
        STW,R1  RLP
        LI,R1   1
        STW,R1  STACKS
        STW,R1  STACK1
        STW,R1  STACK2
        PBEG,R1         STACK1,NTABLE,1
        PNONTERM,R1,R2  STACKS,LBL3,LBL4
LBL3    LW,R1   BRANCH
        CI,R1   1
        BNE     FIN1
        M:WRITE M:LO,(BUF,MESSAGE1),(SIZE,20),(WAIT)
        M:EXIT
FIN1    M:WRITE M:LO,(BUF,MESSAGE2),(SIZE,20),(WAIT)
        M:EXIT
LBL4    IMED,R1,R2,R3           STACK2,0
        IMED,R1,R2,R3           STACK2,0
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,15
        PGOTO,R1        BRANCH,LBL9
        PERROR,R1,R2            BRANCH,STACKS
LBL9    PTERM,R1,R2,R3,R4,SR3       LINE,RLP,LRLP,SLINE,CLINE,BRANCH,;
          TTABLE,BLK,INF1,INF2,INF3,INF4,INF5,1
        PGOTO,R1        BRANCH,LBL12
        PERROR,R1,R2            BRANCH,STACKS
LBL12   IMED,R1,R2,R3           STACK2,0
        PUSH,R1,R2,R3       STACK1,STACK2,0
        IMED,R1,R2,R3           STACK2,1
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,11
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,15
        PGOTO,R1        BRANCH,LBL19
        PERROR,R1,R2            BRANCH,STACKS
LBL19   PTERM,R1,R2,R3,R4,SR3       LINE,RLP,LRLP,SLINE,CLINE,BRANCH,;
          TTABLE,BLK,INF1,INF2,INF3,INF4,INF5,1
        PGOTO,R1        BRANCH,LBL26
        IMED,R1,R2,R3           STACK2,1
        IMED,R1,R2,R3           STACK2,0
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,15
        PGOTO,R1        BRANCH,LBL26
        PERROR,R1,R2            BRANCH,STACKS
LBL26   PTERM,R1,R2,R3,R4,SR3       LINE,RLP,LRLP,SLINE,CLINE,BRANCH,;
          TTABLE,BLK,INF1,INF2,INF3,INF4,INF5,2
        PGOTO,R1        BRANCH,LBL29
        PERROR,R1,R2            BRANCH,STACKS
LBL29   IMED,R1,R2,R3           STACK2,1
        PUSH,R1,R2,R3       STACK1,STACK2,1
```

図-19 (b) コードの出力 その1

```
        IMED,R1,R2,R3           STACK2,1
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,11
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,15
        PGOTO,R1        BRANCH,LBL36
        PERROR,R1,R2            BRANCH,STACKS
LBL36   PTERM,R1,R2,R3,R4,SR3       LINE,RLP,LRLP,SLINE,CLINE,BRANCH,;
          TTABLE,BLK,INF1,INF2,INF3,INF4,INF5,2
        PGOTO,R1        BRANCH,LBL29
        IMED,R1,R2,R3           STACK2,2
        IMED,R1,R2,R3           STACK2,0
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,15
        PGOTO,R1        BRANCH,LBL43
        PERROR,R1,R2            BRANCH,STACKS
LBL43   PTERM,R1,R2,R3,R4,SR3       LINE,RLP,LRLP,SLINE,CLINE,BRANCH,;
          TTABLE,BLK,INF1,INF2,INF3,INF4,INF5,3
        PGOTO,R1        BRANCH,LBL46
        PERROR,R1,R2            BRANCH,STACKS
LBL46   IMED,R1,R2,R3           STACK2,2
        PUSH,R1,R2,R3       STACK1,STACK2,2
        IMED,R1,R2,R3           STACK2,1
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,11
        ARITHM,R1,R2,R3,R4      STACK1,STACK2,15
        PGOTO,R1        BRANCH,LBL53
        PERROR,R1,R2            BRANCH,STACKS
LBL53   PTERM,R1,R2,R3,R4,SR3       LINE,RLP,LRLP,SLINE,CLINE,BRANCH,;
          TTABLE,BLK,INF1,INF2,INF3,INF4,INF5,3
        PGOTO,R1        BRANCH,LBL46
        PUSH,R1,R2,R3       STACK1,STACK2,0
        PUSH,R1,R2,R3       STACK1,STACK2,1
        PUSH,R1,R2,R3       STACK1,STACK2,2
        PNONTERM,R1,R2          STACKS,LBL59,LBL62
LBL59   PGOTO,R1        BRANCH,LBL61
        PERROR,R1,R2            BRANCH,STACKS
LBL61   PLAMBDA,R1,R2           BRANCH,STACKS
LBL62   PBEG,R1                 STACK1,NTABLE,2
        POP,R1,R2,R3        STACK1,STACK2,2
        POP,R1,R2,R3        STACK1,STACK2,1
        POP,R1,R2,R3        STACK1,STACK2,0
        PNONTERM,R1,R2          STACKS,LBL67,LBL68
LBL67   PQUI,R1,R2      STACKS,STACK1,NTABLE,2
LBL68   PUSH,R1,R2,R3       STACK1,STACK2,0
        PUSH,R1,R2,R3       STACK1,STACK2,1
        CONDI,R1,R2,R3      BRANCH,STACK2,0
        PGOTO,R1        BRANCH,LBL73
        PERROR,R1,R2            BRANCH,STACKS
LBL73   PUSH,R1,R2,R3       STACK1,STACK2,0
        PUSH,R1,R2,R3       STACK1,STACK2,2
        CONDI,R1,R2,R3      BRANCH,STACK2,0
        PGOTO,R1        BRANCH,LBL78
        PERROR,R1,R2            BRANCH,STACKS
LBL78   PNONTERM,R1,R2          STACKS,LBL79,LBL82
LBL79   PGOTO,R1        BRANCH,LBL81
        PERROR,R1,R2            BRANCH,STACKS
LBL81   PLAMBDA,R1,R2           BRANCH,STACKS
LBL82   PBEG,R1                 STACK1,NTABLE,3
        PNONTERM,R1,R2          STACKS,LBL84,LBL85
LBL84   PQUI,R1,R2      STACKS,STACK1,NTABLE,3
LBL85   EQU     $
        PGOTO,R1        BRANCH,LBL88
        PERROR,R1,R2            BRANCH,STACKS
LBL88   PLAMBDA,R1,R2           BRANCH,STACKS
        END     START
```

図-19 (c) コードの出力 その2

4. あとがき

本稿では，中間形式が三つ組である場合とラベル付オートマトンである場合のコード生成について述べた。三つ組について理論中心に説明したが，現在まだ我々のMYLANGシステムには導入されていない。

プログラムは(1)データの流れ (data flow): 代入文，入出力文，(2)制御の流れ (control flow): 代入文，繰返し文，手続き文…から成り立っている。(1)の最適化について，本稿によるレジスタ割付け問題の他，共通部分式の削除，畳み込み（定数計算のコンパイル時実行）等がある。共通部分式の削除については，コードを生成する前の三つ組で，2つの共通式が等価であれば一方を削除していくようにすればよい。

(2)の最適化についてはオートマトンで解決できる。繰返し文のループの交差についてはオートマトンの簡約決定性変換の時点で解決されている。IF 文のような論理

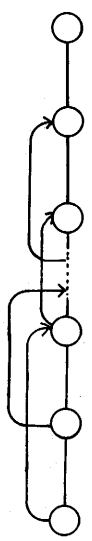


図-20 プログラムの流れ

式についても 3. 4. 2. で述べているようにオートマトンで解決できる。図-18からわかるように，オートマトンは一般に二次元で構成される。コード生成系ではこれを一次元の命令列に変換しなければならない。そのため，この命令列すなわちプログラムには図-20のようなループができることになる。このようなループに対する最適化が今後の課題である。

参 考 文 献

1) 竹中豊弘："言語処理系の自動生成に関する研究 —MYLANG システムの開発" 九工大卒業論文，(1983)
2) Aho, A. V. and Johnson, S. C.: "Optimal code generation for expression trees." J. A. C. M, Vol. 23, No. 3, pp. 488–501 (1976)
3) Aho, A. V. and Sethi, Ravi.: "How hard is compiler generation?" Lecture Notes in Computer Science 52 (1977)
4) Sethi, R. and Ullman, J. D.: "The generation of optimal code for arithmetic expressions." J. A. C. M, Vol. 17, No. 4, pp. 715–728 (1970)
5) 中田育男：コンパイラ，産業図書 (1981)
6) Aho, A. V., Jonson, S. C. and Ullman, J. D.: "Code generation for expressions with common subexpressions." J. A. C. M, Vol. 24, No. 1, pp. 146–160 (1977)
7) Prabhara, B. and Sethi, R.: "Efficient computation of expression with common subexpressions." ACM. Sym. 5th on POPL (1978)
8) Reiner Güttler: "Erzeugung optimalen codes for series-parallel graphs." Lecture Notes in Computer Science 104 (1981)
9) Bruno, J. and Sethi, R.: "Code generation for a one-register machine." J. A. C. M, Vol. 23, No. 3, pp. 502–610 (1976)