Vol. 16 No. 3　　情　報　処　理　　Mar. 1975

論　文

論理式のコンパイル方式*

中　田　育　男**

Abstract

Two kinds of compiling algorithms, which generate efficient object programs for logical expressions, are presented in this paper. The result is optimal in the sense that no redundant evaluations are made.

One algorithm is presented as a recursive procedure which scans trees of logical expressions. The other is a single left-to-right scanning algorithm which is derived from the recursive procedure.

1. はじめに

FORTRAN や ALGOL などの計算機言語の要素の一つに論理式と呼ばれるものがある．それは論理要素(関係式や論理変数など)と論理演算子 (AND, OR, NOT など) で構成されるものであるが，その演算を実行するときには，AND や OR の演算をせずに，それらの演算子の左側の論理要素の値が真 (true) であるか偽 (false) であるかによって適当なところまで飛び越すようにした方が能率が良いことはよく知られている．たとえば

$$A \text{ OR } B$$

という論理式では，A の値と B の値を求め，その OR をとるのではなく，まず A の値を調べ，その値が真であればこの式全体の値が真であることがわかるから，B の値を調べるところを飛び越してしまえばよい．この方式は，B が長い式であったり，A が関係式であるとき効果が大きい．関係式の場合は，大小関係を真か偽かのビットに変換してから AND や OR をとるより，大小関係を見て適当なところまで飛び越すようにした方がずっと能率がよいわけである．

論理演算子が AND と OR だけであれば，コンパイラがこのような能率のよい目的プログラムを作り出すのは比較的簡単であるが，NOT と「かっこ」がからむと難かしくなる．実行効率を重視したコンパイラでは適当なアルゴリズムを使ってこのような目的プログラムを作り出しているのであろうが，うまいアルゴリズムはまだ発表されていない．文献 1) では一つのアルゴリズムを与えているが，そのアルゴリズムは複雑であり，誤まりもある．

(1) A OR B は，A が真なら真としてしまってよい

(2) A AND B は，A が偽なら偽としてしまってよい

(3) $\neg(A \text{ OR } B)$ は $\neg A$ AND $\neg B$

(4) $\neg(A \text{ AND } B)$ は $\neg A$ OR $\neg B$

ということを使って，構文解析と同時に，すなわち，原始プログラムを左から右へと一回だけ走査して，上記のような目的プログラムを生成するアルゴリズムを作り出すのは一見簡単なようであるが難かしい問題である．文献 1) はそれに失敗しているわけであるが，原始プログラムをいったん木構造に変換してからそのような目的プログラムを得るアルゴリズムは比較的簡単に得られる．われわれは，今回，この後者のアルゴリズムから前者のアルゴリズムを導き出すことを試みた．

われわれは本論文で，論理式に対して効率よい目的プログラムを生成する二種類のアルゴリズムを与える．一つは，論理式が構文解析されて木構造 (tree) に変換されたものに対するアルゴリズムを回帰的関数の形で与えたものであり，もう一つは，原始プログラムを左から右へと一回だけ走査してそのような目的プログラムを生成するための一般的なアルゴリズムである．後者は前者の回帰的関数から導き出される．

---

* Compiling Algorithms for Logical Expressions by Ikuo NAKATA (Systems Development Laboratory, HITACHI, Ltd.).

** (株)日立製作所システム開発研究所

2. 論理式の定義

本論文であつかう論理式は，次のようなバッカス記法 (BNF) で定義されるものとする．

〈論理式〉::=〈論理項〉|〈論理式〉∨〈論理項〉
〈論理項〉::=〈論理因子〉|〈論理項〉∧〈論理因子〉
〈論理因子〉::=〈論理1次子〉|¬〈論理因子〉
〈論理1次子〉::=〈論理変数〉|(〈論理式〉)
〈論理変数〉::=$a|b|c|d|e|f|g|h$

ここで，¬ (NOT, 否定)，∧ (AND, 論理積)，∨ (OR, 論理和) の意味は次のような McCarthy の条件式によって定義されている．

$$e_1 \vee e_2 = [e_1 \to T;\ T \to e_2]$$
$$= [e_1 \to T;\ e_2 \to T;\ T \to F] \quad (1)$$
$$e_1 \wedge e_2 = [e_1 \to e_2;\ T \to F]$$
$$= [\neg e_1 \to F;\ T \to e_2]$$
$$= [\neg e_1 \to F;\ e_2 \to T;\ T \to F] \quad (2)$$
$$\neg e_1 = [e_1 \to F;\ T \to T]$$
$$= [\neg e_1 \to T;\ T \to F] \quad (3)$$

ここで，たとえば(1)式の真中の条件式の意味は，e_1 が true なら T(true), そうでなければ e_2 の値，を $e_1 \vee e_2$ の値とするということであり，最後の条件式は，e_1 が true なら，T(true), そうでなくて e_2 が true なら T，そのどれでもないとき F(false) を $e_1 \vee e_2$ の値とするということである．

はじめに述べたように，論理要素として関係式が使われていたとき本アルゴリズムによって得られる目的プログラムの効果は大きい．ここでは，簡単にするために，以下には本質的でない関係式を除いてあるが，本アルゴリズムを関係式を含む場合にまで拡張するのは容易である．

3. トップダウン・アルゴリズム——木走査アルゴリズム

前章の定義から得られる論理式 e は

if e **then** S_1 **else** S_2; (4)

というかたちで使われるが，その目的プログラムは次のようにすればよい．

G ¬e **then go to** l_1;
S_1; **go to** l_2;
l_1: S_2;
l_2:

そこで，これ以後は

if e **then go ot** l; (5)

という形の原始プログラムの目的プログラムについて考えることにする．

(5)式において，e が $e_1 \vee e_2$ の形であるとき，すなわち

if $e_1 \vee e_2$ **then go to** l;

のときは，この目的プログラムは，(1)式により

if e_1 **then go to** l;
if e_2 **then go to** l;

のそれと同じにすればよい．いま，(5)式の目的プログラムを

$$T(e, t, l) \quad (6)$$

と書くことにする．ここで t は **true** を意味する．すなわち，これは，e が **true** なら l に飛ぶという目的プログラムを意味する．この記法によれば

$$T(e_1 \vee e_2, t, l) = \begin{bmatrix} T(e_1, t, l) \\ T(e_2, t, l) \end{bmatrix}$$

となる．ここで $\begin{bmatrix} A \\ B \end{bmatrix}$ は目的プログラム A, B をこの順につないだものを意味する．また

if ¬e_1 **then go to** l;

は

if e_1=**false then go to** l;

と同じである．すなわち

$$T(\neg e_1, t, l) = T(e_1, f, l)$$

となる．ここで，f は **false** を意味する．

G $e_1 \wedge e_2$ **then go to** l;

の目的プログラムは(2)式より

if ¬e_1 **then go to** l_1;
if e_2 **then go to** l;
l_1:

とすればよい．ただし，l_1 は新しく作られるラベルである．これは

$$T(e_1 \wedge e_2, t, l) = \begin{bmatrix} T(e_1, f, \text{newlabel}) \\ T(e_2, t, l) \\ \text{newlabel}: \end{bmatrix}$$

と書くことができる．ここで，右辺の最初の「newlabel」は，そこで新たに生成したラベルをさし，最後の「newlabel:」はそのラベルの値がそこで定義されることを示す．また

$$\neg(e_1 \vee e_2) = \neg e_1 \wedge \neg e_2$$

であることから

$$T(e_1 \vee e_2, f, l) = T(\neg(e_1 \vee e_2), t, l)$$
$$= T(\neg e_1 \wedge \neg e_2, t, l)$$
$$= \begin{cases} T(\neg e_1, f, \text{newlabel}) \\ T(\neg e_2, t, l) \end{cases}$$

$$= \begin{cases} \text{newlabel}: \\ \quad T(e_1, t, \text{newlabel}) \\ \quad T(e_2, f, l) \\ \text{newlabel}: \end{cases}$$

となる．同様に

$$T(e_1 \wedge e_2, f, l) = T(\neg(\neg e_1 \vee \neg e_2), t, l)$$
$$= T(\neg e_1 \vee \neg e_2, t, l)$$
$$= \begin{cases} T(\neg e_1, t, l) \\ T(\neg e_2, t, l) \end{cases}$$
$$= \begin{cases} T(e_1, f, l) \\ T(e_2, f, l) \end{cases}$$

となる．e が論理変数であるときは

if e **then go to** l;

の目的プログラムは

LOAD e
JT l

となる．ここで，JT は Jump if true という命令である．以上の結果，T という関数は **Table 1** によって回帰関数として定義される．

Table 1 の中で「∨」と「∧」に関する4つのますを1組の式で表現するために次の記法を導入する．

$$\text{cond}(p) = \begin{cases} \textbf{true} & p = \vee \text{のとき} \\ \textbf{false} & p = \wedge \text{のとき} \end{cases}$$

$$\text{label}(p, c, l) = \begin{cases} l & \text{cond}(p) = c \text{ のとき} \\ \text{newlabel} & \text{cond}(p) \neq c \text{ のとき} \end{cases}$$

$$\text{deflabel}(p, c, l) = \begin{cases} \text{label}(p, c, l): & \text{cond}(p) \neq c \text{ のとき} \\ \phi & \text{cond}(p) = c \text{ のとき} \end{cases}$$

この記法を使えば，Table 1 の上4組の式は

$$T(e_1\, p\, e_2, c, l) = \begin{cases} T(e_1, \text{cond}(p), \text{label}(p, c, l)) \\ T(e_2, c, l) \\ \text{deflabel}(p, c, l) \end{cases} \quad (7)$$

と1組の式で表現される．

Table 1 Definition of $T(e, c, l)$: object code of **if** $e = c$ **then go to** l

e \ c	t (true)	f (false)
$e = e_1 \vee e_2$	$T(e_1, t, l)$ $T(e_2, t, l)$	$T(e_1, t, \text{newlabel})$ $T(e_2, f, l)$ newlabel:
$e = e_1 \wedge e_2$	$T(e_1, f, \text{newlabel})$ $T(e_1, t, l)$ newlabel:	$T(e_1, f, l)$ $T(e_2, f, l)$
$e = \neg e_1$	$T(e_1, f, l)$	$T(e_1, t, l)$
e = 変数	LOAD e JT l	LOAD e JF l

Table 2 Object code of **if** $a \wedge b \vee \neg(c \vee d)$ **then go to** l

$T(e, c, l)$	目的プログラム	
$T(a \wedge b \vee \neg(c \vee d), t, l)$		
$= T(a \wedge b, t, l)$		
$= T(a, f, l_1)$	LOAD	a
	JF	l_1
$T(b, t, l)$	LOAD	b
	JT	l
l_1:	l_1:	
$T(\neg(c \vee d), t, l)$		
$= T(c \vee d, f, l)$		
$= T(c, t, l_2)$	LOAD	c
	JT	l_2
$T(d, f, l)$	LOAD	d
	JF	l
l_2:	l_2:	

この回帰的関数 T では，論理式 e に関する値を，その部分式 (e_1, e_2) に関する値によって定義している．したがって，この関数は，木構造 (tree) の形で表現された論理式を，根 (root) から葉 (leaf) へとたどって目的プログラムを作り出すアルゴリズムを表現している．**Table 2** に例題を示す．

4. ボトムアップ・アルゴリズム—順次走査アルゴリズム

前章で得られた回帰的関数 T は，論理式 e に関する値を，その部分式に関する値に分解していくアルゴリズムを示している．すなわち，そのアルゴリズムはトップダウンのアルゴリズムである．本章では，まず，この T からボトムアップのアルゴリズムを導き出すための定理を与える．次に，その定理から，一般的なボトムアップのアルゴリズムを導き出し，それを LR(k) パーサと，演算子優先順位パーサに適用する．

4.1 基本定理

まず，論理式に含まれる部分式 $e'\,p\,e''$ を考える．すなわち，e は

$$e = \cdots(e'\,p\,e'')\cdots$$

という形をしているとする．ここで，かっこは部分式であることを示すためにつけたものであって，原始プログラムの中では，必ずしもかっこがつけられてはいない．次に，この e' の右端のオペランドを E とする．一般に，論理式 e_i がさらに部分式に分解されているときには

$$e_i = e_{i+1}\, p_{i+1}\, e_{i+1}' \quad (8)$$

または

$$e_i = \neg e_{i+1} \quad (9)$$

の形になる．ここで，p_{i+1} は「∨」または「∧」，

e_{i+1}, e_{i+1}' は論理式である．$\neg(\neg e)=e$ であるから「¬」を何個か重ねても，その個数が偶数なら「ϕ」すなわち「¬」が何も無いこと，奇数なら「¬」1個と同じ意味である．そこで(8)，(9)式をあわせて

$$e_i=z_i(e_{i+1}\,p_{i+1}\,e_{i+1}') \tag{10}$$

の形で表現できる．ここで z_i は「ϕ」または「¬」である．ただし，(9)式の形で e_{i+1} が論理変数である場合だけが(10)式では表現できない．この書き方を使えば，e' の右端のオペランドが E であるということは，次の形で表現される．

$$\begin{aligned} e'=e_0'&=z_0(e_1\,p_1\,e_1')\\ &=z_0(e_1\,p_1\,z_1(e_2\,p_2\,e_2'))\\ &=z_0(e_1\,p_1\,z_1(e_2\,p_2\,z_2(\cdots(e_n\,p_n\,e_n')\cdots))) \end{aligned} \tag{11}$$

ここで $n\geqq 0$，$e_n'=z_nE$．このとき次の定理がなりたつ．

定理 1　論理式 e の部分式 $e'\,p\,e''$ において，e' の右端のオペランドが E であるとき，すなわち，e' が(11)式の形をしているとき，$T(e,c,l)$ によって作られる目的プログラムの中で E に関するものは，

$z_0z_1\cdots z_n\mathrm{cond}(p)$ が true なら

LOAD E
JT　l'

$z_0z_1\cdots z_n\mathrm{cond}(p)$ が false なら

LOAD E
JF　l'

の形をしている．

証明：$T(e,c,l)$ を Table 1 にしたがって分解していく途中に $T(e'\,p\,e'',c_0,l'')$ という形が現われるが，これがさらに(7)によって

$$T(e'\,p\,e'',c_0,l'')=\begin{bmatrix}T(e',\mathrm{cond}(p),\mathrm{label}(p,c_0,l''))\\ T(e'',c_0,l'')\\ \mathrm{deflabel}(p,c_0,l'')\end{bmatrix}$$

と分解される．この右辺の最初の式は，$l'=\mathrm{label}(p,c_0,l'')$ とすれば，(11)式と Table 1 より

$$\begin{aligned} &T(e',\mathrm{cond}(p),l')\\ &\quad=T(e_0',\mathrm{cond}(p),l')\\ &\quad=T(z_0(e_1\,p_1\,e_1'),\mathrm{cond}(p),l')\\ &\quad=T(e_1\,p_1\,e_1',z_0\mathrm{cond}(p),l')\\ &\quad=\begin{bmatrix}T(e_1,\mathrm{cond}(p_1),\mathrm{label}(p_1,z_0\mathrm{cond}(p),l'))\\ T(e_1',z_0\mathrm{cond}(p),l')\\ \mathrm{deflabel}(p_1,z_0\mathrm{cond}(p),l')\end{bmatrix} \end{aligned}$$

となり，その最後の $T(e_1',z_0\mathrm{cond}(p),l')$ より

$$\begin{aligned} &T(e_1',z_0\mathrm{cond}(p),l')\\ &\quad=T(z_1(e_2\,p_2\,e_2'),z_0\mathrm{cond}(p),l')\\ &\quad=T(e_2\,p_2\,e_2',z_0z_1\mathrm{cond}(p),l')\\ &\quad=\begin{bmatrix}T(e_2,\mathrm{cond}(p_2),\mathrm{label}(p_2,z_0z_1\mathrm{cond}(p),l'))\\ T(e_2',z_0z_1\mathrm{cond}(p),l')\\ \mathrm{deflabel}(p_2,z_0z_1\mathrm{cond}(p),l')\end{bmatrix} \end{aligned}$$

を得る．同様の操作をくり返していけば，最後に

$$\begin{aligned} &T(e_n',z_0z_1\cdots z_{n-1}\mathrm{cond}(p),l')\\ &\quad=T(E,z_0z_1\cdots z_n\mathrm{cond}(p),l') \end{aligned}$$

が得られる．この値（目的プログラム）は Table 1 より

$$T(E,z_0z_1\cdots z_n\mathrm{cond}(p),l')=\begin{cases}\begin{bmatrix}\mathrm{LOAD}\ E\\ \mathrm{JT}\quad l'\end{bmatrix} & z_0z_1\cdots z_n\mathrm{cond}(p)=\mathrm{true}\\ & (n\geqq 0)\ \text{のとき}\\ \begin{bmatrix}\mathrm{LOAD}\ E\\ \mathrm{JF}\quad l'\end{bmatrix} & z_0z_1\cdots z_n\mathrm{cond}(p)=\mathrm{false}\\ & (n\geqq 0)\ \text{のとき}\end{cases}$$

となる．　　証明終り．

定理1と同じ仮定で，e_i に関する目的プログラムの jump 命令の jump 先については，次の定理が成り立つ．

定理 2　論理式 e の部分式 $e'\,p\,e''$ において，e' が(11)式の形をしているとき，$T(e,c,l)$ によって作られる目的プログラムの中で e' に関して作られるものを $T(e',\mathrm{cond}(p),l')$，$e_i$ に関するものを $T(e_i,\mathrm{cond}(p_i),l_i)$ とする．さらに，e' の右端のオペランド E に関して作られる jump 命令の次の番地を l_p とする．このとき次のことが成り立つ．

$$\begin{cases}l_i=l' & z_0z_1\cdots z_{i-1}\mathrm{cond}(p_i)=\mathrm{cond}(p)\ \text{のとき}\\ l_i=l_p & z_0z_1\cdots z_{i-1}\mathrm{cond}(p_i)\neq\mathrm{cond}(p)\ \text{のとき}\end{cases}$$
$$i=1,\cdots,n$$

証明：定理1の証明の途中で得られたように

$$\begin{aligned} &T(e',\mathrm{cond}(p),l')\\ &\quad=T(e_1\,p_1\,e_1',z_0\mathrm{cond}(p),l')\\ &\quad=\begin{bmatrix}T(e_1,\mathrm{cond}(p_1),\mathrm{label}(p_1,z_0\mathrm{cond}(p),l'))\\ T(e_1',z_0\mathrm{cond}(p),l')\\ \mathrm{deflabel}(p_1,z_0\mathrm{cond}(p),l')\end{bmatrix} \end{aligned}$$

である．ここで，$\mathrm{cond}(p_1)\neq z_0\mathrm{cond}(p)$ のとき $\mathrm{label}(p_1,z_0\mathrm{cond}(p),l')=l_1$ は新しく導入される newlabel であり，その値が定義されるところが

$\mathrm{deflabel}(p_1,z_0\mathrm{cond}(p),l')$ の書いてあるところである．その場所は $T(e_1',z_0\mathrm{cond}(p),l')$ の次であるが，定理1の証明より，$T(e_1',z_0\mathrm{cond}(p),l')$ の最後の命令が E に関する jump 命令である．すなわち，この newlabel の値は l_p と同じである．すなわち，$\mathrm{cond}(p_1)\neq z_0\mathrm{cond}(p)$ のとき $l_1=l_p$ である．一般に

$$T(e_{i-1}', z_0z_1\cdots z_{i-2}\mathrm{cond}(p), l')$$
$$= T(z_{i-1}(e_i\, p_i\, e_i'), z_0z_1\cdots z_{i-2}\mathrm{cond}(p), l')$$
$$= T(e_i\, p_i\, e_i', z_0z_1\cdots z_{i-1}\mathrm{cond}(p), l')$$
$$= \begin{cases} T(e_i, \mathrm{cond}(p_i), l_i) \\ T(e_i', z_0z_1\cdots z_{i-1}\mathrm{cond}(p), l') \\ \mathrm{deflabel}(p_i, z_0z_1\cdots z_{i-1}\mathrm{cond}(p), l') \end{cases}$$

であり，

$$l_i = \mathrm{label}(p_i, z_0z_1\cdots z_{i-1}\mathrm{cond}(p), l')$$

は

$$\mathrm{cond}(p_i) = z_0z_1\cdots z_{i-1}\mathrm{cond}(p)$$

すなわち，

$$z_0z_1\cdots z_{i-1}\mathrm{cond}(p_i) = \mathrm{cond}(p)$$

のとき $l_i = l'$ であり，

$$\mathrm{cond}(p_i) \neq z_0z_1\cdots z_{i-1}\mathrm{cond}(p)$$

すなわち

$$z_0z_1\cdots z_{i-1}\mathrm{cond}(p_i) \neq \mathrm{cond}(p)$$

のとき l_i は newlabel であり，その値の定義されるところ (deflabel($p_i, z_0z_1\cdots z_{i-1}\mathrm{cond}(p), l'$) の書いてあるところ) は $T(e_i', z_0z_1\cdots z_{i-1}\mathrm{cond}(p), l')$ の目的プログラムの次，すなわち $l_i = l_p$ である．証明終り．

4.2 基本アルゴリズム

ここでは，原始プログラムの形のままになっている論理式を，左から右へと一度だけ走査 (scan) することによって，能率のよい目的プログラムを作り出すための基本アルゴリズムを，前節に与えた定理から導き出す．

原始プログラムを左から右へと走査しながら，回帰的関数 T によって得られる目的プログラムと同じ目的プログラムを作り出すことを考える．その際，目的プログラムはできるだけ早目に作り出すことにする．それによって，一般的にアルゴリズム (プログラム) が簡単になり，処理速度も上るからである．たとえば，論理変数 E を見たとき直ちに「LOAD E」の命令を作り出すことにする．次に，その E に関する「JF」か「JT」かの jump 命令を作れるのは，前節の定理1によって，E の直後のオペレータ p (「∨」か「∧」) を見たときである．ただし，そのとき，その jump 先はまだわからない．そこで，とりあえず新しいラベル m を導入し，「JT m」または「JF m」という命令を作ることとする．この命令を「JT」とするか「JF」とするかは定理1の $z_0z_1\cdots z_n\mathrm{cond}(p)$ すなわち，p の左側のオペランド e' の中で，E にかかっている「¬」の数と p によってきまる．以下のアルゴリズムでは，その「¬」の数が偶数のとき true，奇数のとき false の値をとる変数 N を設定する．すなわち，$N = z_0z_1\cdots z_n$ **true** である．

次に，このようにして作った jump 命令の行先 (飛先) の決め方，すなわち，m の値を定義する場所の決め方が問題になる．今，左から右に走査していって，オペレータ p を見たとき決められる飛先について考えてみる．定理2は，p の左側の式 e' の中で作られた jump 命令の飛先は二通りに分けられることを示している．一つは l' と同じであるが，これは定理1の証明からわかるように，e' の右端のオペランド E に関して作られる jump 命令の飛先と同じものである．もう一つは l_p と同じ，すなわち E に関して作られる jump 命令の次の番地と同じである．そこで，e' の中で導入されたラベルが，p を見たとき，このように二つのグループに分けられていればよい．それぞれのグループのラベルを chain でつないだ list とする．その中で，定理2における $z_0z_1\cdots z_{i-1}\mathrm{cond}(p_i)$ が true であるような l_i をつないだものを t-list，false であるものをつないだものを f-list と呼ぶことにする。

これらの N, t-list, f-list を使って，LOAD 命令，jump 命令を作る操作と，飛先を決める操作を，BNF 表現による論理式の定義の上で考えてみる．何らかの操作をするのは，non-terminal を認識したときと，「∨」,「∧」のオペレータを見たときである．それらの操作を P_i で表わし，それを BNF 表現の中に，次のように書き込んでみる．

$$\begin{array}{ll} 0) & b \rightarrow \vdash b_0 \dashv \{P_0\} \\ 1) & b_0 \rightarrow b_1 \\ 2) & b_0 \rightarrow b_0 \vee \{P_1\}\, b_1\, \{P_2\} \\ 3) & b_1 \rightarrow b_2 \\ 4) & b_1 \rightarrow b_1 \wedge \{P_3\}\, b_2\, \{P_4\} \\ 5) & b_2 \rightarrow b_3 \\ 6) & b_2 \rightarrow \neg b_2\, \{P_5\} \\ 7) & b_3 \rightarrow i\, \{P_6\} \\ 8) & b_3 \rightarrow (b_0) \end{array} \qquad (12)$$

ここで，たとえば 2) 式は第2章の

〈論理式〉::=〈論理式〉∨〈論理項〉

に対応し，「∨」を見たとき P_1 の操作を行ない，$b_0 \vee b_1$ を認識したとき P_2 の操作を行なうことを示す．0) 式の「⊢」と「⊣」は論理式の前後の区切り記号で，たとえば「⊢」は **if**，「⊣」は **then** または **then go to** l にあたる．各 P_i は次のように決めればよい．ここでは，t-list, f-list を一時格納しておくためにスタック S を使う．

P_1: (i) 新しいラベル l を導入し

N=**true** なら「JT l」を作る；
N=**false** なら「JF l」を作る

(ii) f-list に入っているすべての l_i の値を(i)で作った命令の次の番地とする．すなわち，「l_i:」を作る (for all $l_i \in f$-list).

(iii) t-list に l をつないだものを S に push down する

(解説) N, t-list, f-list は正しく求められていると仮定する．この仮定の正しいことは $P_i(i=1, \cdots, 6)$ によって保証される．

(i)は，$N=z_0z_1\cdots z_n$**true**$=z_0z_1\cdots z_n$cond(∨) と定理1より．

(ii)は，BNF 表現の2)式の右辺の b_0 に関して t-list, f-list が求められていることと，その f-list が，定理2 (その e' が今の b_0 にあたる) において $z_0z_1\cdots z_{i-1}$cond(p_i)=**false**$\neq$cond(p)=cond(∨) となる i に関する l_i の list であることから，

(iii)の t-list に l をつなぐというのは，t-list が，定理2において $z_0z_1\cdots z_{i-1}$cond(p_i)=**true**=cond(p)=cond(∨) となる i に関する l_i の list であることと，定理2の l' は今の l にあたることから．push down するのは，あとで P_2 で使うため．

P_2: S から list を一つ pop up して，それを t-list につなぐ．

(解説) 2)式の b_1 に関して t-list, f-list が求められている．これから2)式の左辺の b_0 に関する t-list, f-list を求めるわけであるが，この b_1 と b_0 の間に「¬」は入っていないから，b_1 の t-list, f-list をそのまま b_0 の t-list, f-list とし，2)式の右辺の b_0 に関する list を，cond(∨)=**true** であるから t-list につなげばよい．

P_3: (i) 新しいラベル l を導入し
N=**true** なら「JF l」を作る
N=**false** なら「JT l」を作る

(ii) t-list に入っているすべての l_i の値を(i)で作った命令の次の番地とする．すなわち，「l_i:」を作る (for all $l_i \in t$-list)

(iii) f-list に l をつないだものを S に push down する

(解説) cond(∧)=**false** で cond(∨)=**true** だから，P_1 における t(**true**) を f(**false**) でおきかえたものになる．

P_4: S から list を一つ pop up して，それを f-list につなぐ．

P_5: $N \leftarrow \neg N$ (N の値を反転する)
t-list と f-list を交換する (t-list の内容を新たに f-list とし，f-list を t-list とする)．

(解説) $N_0=z_0z_1\cdots z_n$**true**, $N_1=z_1\cdots z_n$**true** とすれば，$z_0=\neg$ のとき $N_0=\neg N_1$.
$z_0z_1\cdots z_{i-1}$ cond(p_i)=**true** であるような l_i の list ((6)式の左辺の b_2 の t-list) は，$z_0=\neg$ なら $z_1\cdots z_{i-1}$cond(p_i)=**false** であるような l_i の list ((6)式の右辺の b_2 の f-list) である．

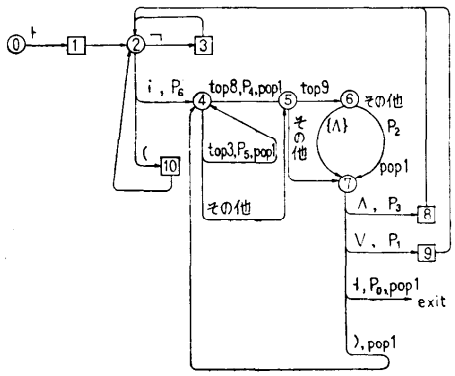
P_6: 「LOAD i」を作る
$N \leftarrow$ **true**; t-list←null; f-list←null;

(解説) N, t-list, f-list の initialization

P_0: 「┤」=**then go to** l_0 なら P_1 と同じ (ただし P_1 の l を l_0 でおきかえる)．
「┤」=**then** なら P_3 と同じ．

4.3 LR(k) パーサ

前節で与えたアルゴリズムを LR(k) パーサとして具体化すれば **Fig. 1** が得られる．Fig. 1 は LR(k) アナライザ[2] を使って得られたものをさらに人手で最適化したものである．**Table 3** (次頁参照) に例題を示す．Table 3 の最後でスタック S に残ったリスト $\{l_2, l_0\}$ からは，l_0 の値 (l_0 の場所) が決まったところで
l_2: l_0:
が作られる．



ここで，Ⓙ, [j] の j は状態番号であり [j] は番号 j をスタックすること
線上の terminal symbol は，それを走査(scan)したときに状態遷移が起こること．
P_i は前節の P_i の操作をすること
top j はスタックの top が j のときに状態遷移が起こること
{∧}は先読み(look ahead)して「∧」であったら状態遷移が起こることを示す．

Fig. 1 LR(k) parser for logical expressions

Table 3 An Example of Parsing and code generation process by $LR(k)$ Parser: **if** $a \wedge b \vee \neg(c \vee d)$ **then go to** l_0;

読込記号	状態番号	状態番号のスタック	操作名	スタック S	N	t-list	f-list	目的プログラム
	0							
if	1,2	1						
a	4		P_6		true	null	null	LOAD a
	5,7							
$\wedge$	8,2	1,8	P_3	$\{l_1\}$				JF l_1
b	4		P_6		true	null	null	LOAD b
	5,7	1	P_4	ϕ			$\{l_1\}$	
$\vee$	9	1,9	P_1	$\{l_2\}$				JT l_2
	2							l_1:
$\neg$	3,2	1,9,3						
(	10,2	1,9,3,10						
c	4,5,7		P_6		true	null	null	LOAD c
$\vee$	9,2	1,9,3,10,9	P_1	$\{l_2\}\{l_3\}$				JT l_3
d	4,5		P_6		true	null	null	LOAD d
	6,7	1,9,3,10	P_2	$\{l_3\}$		$\{l_3\}$		
)	4	1,9,3	P_5		false	null	$\{l_3\}$	
	4,5,6,7	1,9	P_2	ϕ		$\{l_2\}$		
then go to l_0	*exit*	1	P_0	$\{l_2, l_0\}$				JF l_0
								l_3:

4.4 演算子優先順位パーサ

従来，一般によく行なわれてきた演算子の優先順位による構文解析を行なう場合は，(12)式は次のように解釈できる．

演算子のスタックから「⊢」を pop up するとき P_0 を行なう．

「∨」を見たとき，その「∨」の左側の式の処理をすませてから P_1 を行ない，「∨」を pop up するとき P_2 を行なう．(「∧」と P_3, P_4 についても同様)．

「¬」を pop up するとき P_5 を行なう．

i (論理変数) を見たとき P_6 を行なう．

これから次のアルゴリズムを得る．それが演算子優先順位にしたがったパーサである．そこでは演算子スタックが使われている．それは基本アルゴリズムで使われているスタック S と別のものであるが，両者を共用してもよい．

0) 原始プログラムを左から右へと走査する．今見ている記号を χ とする．

1) χ=変数 なら P_6 を行なう

2) χ=演算子 なら

まず，通常の演算子順位(「¬」,「∧」,「∨」の順であり，「¬」が一番強い)にしたがった pop up を行なう(pop up されるものがないこともある)

pop up されるものが

a)「(」なら，単に pop up

b)「¬」なら P_5 を行なう

c)「∧」なら P_4 を行なう

d)「∨」なら P_2 を行なう

e)「⊢」なら P_0 を行なう

pop up するものが無くなったら，χ を push down する．さらに

f) χ=「∨」なら P_1 を行なう

g) χ=「∧」なら P_3 を行なう

Table 4 (次頁参照) に例題を示す．そこで最後にスタック S に残ったリスト $\{l_6, l_7, l_2, l_3\}$ の中の各 l_i の値 (l_i の定義されるところ) は，**then** に対応する **else** 以下の目的プログラムの先頭番地である．

5. 回帰的関数 T の拡張

ALGOL には，「∨」,「∧」,「¬」の他に論理演算子として「⊃」(imply) と「≡」(equivalent) とがある．このうち，「⊃」に関して回帰的関数 T を拡張するのは容易であるが，「≡」に関しては，あまりうまく拡張できない．

次に，全称記号「∀」と存在記号「∃」を含んだ述語論理式に拡張することを考えてみる．「$\forall J\, e$」は「すべての J について e が true である」ことを，「$\exists J\, e$」は「e が true であるような J が存在する」ことを意味する．ここで e は一般に J の関数である．これらに関して関数 $T(e, c, l)$ を **Table 5** (次頁参照) のようにして拡張することができる．ここで「**for all** J M」は，M をループの本体として，J に関するループを目的プログラムとして作り出すことを意味する．たとえば

Table 4 An Examlpe of Parsing and code generating process by operator precedence parser: **if** $(a\vee b)\wedge\neg((c\vee d)\wedge\neg(e\vee\neg f\wedge g)\vee h)$ **then**

x	演算スタック	操作名	スタック S	N	t-list	f-list	目的プログラム
if	**if**						
(	**if** (						
a		P_6		t	null	null	LOAD a
$\vee$	**if** ($\vee$	P_1	$\{l_1\}$				JT l_1
b		P_6		t	null	null	LOAD b
)	**if** (	P_2	ϕ		$\{l_1\}$		
	if						
$\wedge$	**if**$\wedge$	P_3					JF l_2
			$\{l_2\}$				l_2:
$\neg$	**if**$\wedge\neg$						
(	**if**$\wedge\neg$(						
(	**if**$\wedge\neg$((						
c		P_6		t	null	null	LOAD c
$\vee$	**if**$\wedge\neg$(($\vee$	P_1	$\{l_2\}\{l_3\}$				JT l_3
d		P_6		t	null	null	LOAD d
)	**if**$\wedge\neg$((	P_2	$\{l_2\}$		$\{l_3\}$		
	if$\wedge\neg$(						
$\wedge$	**if**$\wedge\neg$($\wedge$	P_3					JF l_4
			$\{l_2\}\{l_4\}$				l_3:
$\neg$	**if**$\wedge\neg$($\wedge\neg$						
(	**if**$\wedge\neg$($\wedge\neg$(						
e		P_6		t	null	null	LOAD e
$\vee$	**if**$\wedge\neg$($\wedge\neg$($\vee$	P_1	$\{l_2\}\{l_4\}\{l_5\}$				JT l_5
$\neg$	**if**$\wedge\neg$($\wedge\neg$($\vee\neg$						
f		P_6		t	null	null	LOAD f
$\wedge$	**if**$\wedge\neg$($\wedge\neg$($\vee$	P_3		f			
	if$\wedge\neg$($\wedge\neg$($\vee\wedge$	P_3	$\{l_2\}\{l_4\}\{l_5\}\{l_6\}$				JT l_6
g		P_6		t	null	null	LOAD g
)	**if**$\wedge\neg$($\wedge\neg$($\vee$	P_4	$\{l_2\}\{l_4\}\{l_5\}$			$\{l_6\}$	
	if$\wedge\neg$($\wedge\neg$(	P_2	$\{l_2\}\{l_4\}$		$\{l_5\}$		
	if$\wedge\neg$($\wedge\neg$						
$\vee$	**if**$\wedge\neg$($\wedge$	P_5		f	$\{l_6\}$	$\{l_5\}$	
	if$\wedge\neg$(	P_4	$\{l_2\}$			$\{l_5, l_4\}$	
	if$\wedge\neg$($\vee$	P_1					JF l_7
			$\{l_2\}\{l_6, l_7\}$				l_5: l_4:
h		P_6		t	null	null	LOAD h
)	**if**$\wedge\neg$(	P_2	$\{l_2\}$		$\{l_6, l_7\}$		
	if$\wedge\neg$						
then	**if**$\wedge$	P_5		f	null	$\{l_6, l_7\}$	
	if	P_4	ϕ			$\{l_6, l_7, l_2\}$	
	ϕ	$P_0=P_2$	$\{l_6, l_7, l_2, l_3\}$				JT l_3

Table 5 An Extension of $T(e, c, l)$

e \ c	t	f
$e=\forall_J e_1$	**for all** J 　$T(e_1, f,$ newlabel); **go to** l; newlabel:	**for all** J 　$T(e_1, f, l)$
$e=\exists_J e_1$	**for all** J 　$T(e_1, t, l)$	**for all** J 　$T(e_1, t,$ newlabel); **go to** l; newlabel:

for all J　　$T(e_1, f, l)$

は

get first J;
while there is J
　do $T(e_1, f, l)$; **get next** J **end while**

とすればよい.

6. むすび

論理式に関して，飛び越し型の能率よい目的プログラムを作り出すための，コンパイラのアリゴリズムを与えた．まず，直感的にわかりやすい，トップダウン型のアルゴリズムを回帰的関数 T として与えた．こ

れは，原始プログラムをいったん木構造 (tree) の中間語に変換して，その中間語から目的プログラムを作り出すコンパイラに適用することができる．次にこの回帰的関数 T から，ボトムアップ型のアルゴリズムを導き出した．このアルゴリズムは BNF による論理式の定義式の上で，何を認識したときに何を実行すればよいかという形で与えてあるので，各種のパーサ（構文解析ルーチン）に適用することができる．

回帰的関数 T の原型となったアルゴリズムは HITAC 5020 の PL/IW コンパイラに適用された．4.4 節のアルゴリズムの原型は HITAC 5020 の HARP (FORTRAN) コンパイラに適用した．4.4 節のアルゴリズムは HITAC 8700/8800 の FORTRAN コンパイラに適用された．それらのアルゴリズムは試行錯誤をくり返して得られたものであったが，本論文はそのアルゴリズムの意味づけ，あるいは正しいことの証明を与えたものと言えよう．

終りに，本論文の全般にわたって種々のアイデアを出していただいた日立製作所システム開発研究所の野木兼六，および日立製作所中央研究所の霜田忠孝，二村良彦の諸氏に深謝いたします．

参 考 文 献

1) H. D. Husky: Compiling Techniques for Boolean Expression & Conditional Statements in ALGOL 60, Comm. ACM. Vol. 4, pp. 70～75 (1961)

2) 小島，加藤，中田: LR(k)-Parser 生成システムとその FORTRAN コンパイラへの応用，情報処理，Vol. 15, No. 2 (1974)

(昭和 49 年 5 月 24 日受付)
(昭和 49 年 7 月 31 日再受付)