CAM(Content Addressed Memory)型データをもつLISP

後藤 英一[*,**]　鈴木 正幸[**]　稲田 信幸[*]
[*]理化学研究所 情報科学　[**]東京大学 理学部

1. 序

近年，特定の計算機に依存しないデータの抽象化の必要性が強調されるようになった。古くからある言語LISP，特にMcCarthyの純LISP[1]は，CAR, CDR, CONS, EQ, ATOMの5種の基本演算の抽象的定義から全てが組み立てられている。それ故1cell 2 pointerのインプリメンティションと線型化された1cell 1 pointerのインプリメンティション[2,3]も，LISPのユーザからはまったく区別がつかない。これは純LISPにおける諸操作のたまものと言えよう。

しかし，現実のLISPにおいては、数値演算が計算機に依存する，またp-listと配列に関する操作は、抽象化と標準化が不充分であるため諸LISP間の互換性をなくす大きな原因となっている。

LISPの標準化に関してはUtah大学のHearn教授によるStandard LISP[4,5]の提案があり，我々もこの提案を取り入れていく考えである。

理研において開発が進められている数式処理用計算機はソフト，ハードともにFLATSと呼ばれ，FORTRAN，LISP，Associative processing，Tuple or Tree，Setの頭文字をとったものである。FLATSで実現されるデータ型はStandard LISPのそれとの互換性を保ち，更にいくつかの新しいデータ型を取り入れる。特に，CAM(Content Addressed Memory)の'single hit'機構をハッシングによって実現し，この上にハッシュ・データ型が組み立てられる。

2章において，FLATSにおけるデータ型およびその基本演算を示す。データ型，演算ともにStandard LISPに対して上位互換として実現する。(上位の部分はVINとハッシュ型，これらもできるだけStandard LISPに取り入れてもらうつもりである) HLISP上で開発されたハッシュ・データ型(HLIST, ASSOCIATOR, HSET, CLUB等)の概念を新たに構成しなおす。

3章において，2章で定義されたデータ型を使用した応用例を示す。

2. データ型と基本演算

FLATSにおけるデータ型は'Extern'型と'Intern'型に大別できる。Extern型のデータは全てintern操作によりIntern型となりうる。internは後に述べる同等性の定義に従い，等しいデータに対して，システム内でユニークな表現をとる事を保障する。internされたデータをGID(Generalized IDentifier)と呼ぶ。

よく知られているデータ型に対しては簡単な説明にとどめ，BNF定義などできるだけ省略する。新しいデータ型については、データ構造，インプリメントの事などは省き，外部表現，概念，同等性(equality)，基本演算について述べていく。できるだけLISPの定義(M-式，u*はuの表わすデータ，等々)にそって説明していくが、簡約化のため数学の記法も取り入れている。

尚，ハッシュ型に対するアクセスの時間計算量はOh*(1)(ハッシングにより O(1))である事は保障されているが，実現方法は[6,9,10,11]による。

DATA TYPE / OPERATION	INT	FLOATNUM	STR	LIST	VIN	AMT	CAT
predicate	intp	floatp	stringp	pairp	vinp	amtp	catp
rewitable?	NO	NO	NO	YES	YES	YES	YES
internable?	YES	YES	YES	YES	YES	YES	YES
atom	T	T	T	NIL	NIL	NIL	NIL
equality	value	value	collation	structure	entity	set	set mapping
creation	----	----	compress	cons	mkvin Oh*(n)	mkamt Oh*(n)	mkcat Oh*(n)
read	----	----	----	----	getvin Oh*(1)	getamt Oh*(1)	getcat Oh*(1)
insertion	----	----	----	----	putvin Oh*(1)	putamt Oh*(1)	putcat Oh*(1)
deletion	----	----	----	----	----	remamt Oh*(1)	remcat Oh*(1)

Fig. 1 Data Types and Operations

2.1 Extern データ型

2.1.1. STRing

標準外部表現: nH<n個の文字列> あるいは "<任意個の文字列>"

意味: 計算機で許される任意長の文字コードの列。

同等性: 文字列の並び長さが等しいこと。

2.1.2 INTeger

標準外部表現: 正の符号省略可能な10進数表現で桁数は無制限。

意味: 四則演算は必要に応じ多倍長演算をおこない, 記憶領域のあふれがない限り数学の整数と同じである。

同等性: 数学の常識どうり。

2.1.3 FLOATing point NUMber

標準外部表現: <FLOATNUM> := <MEFLOATNUM>|<MEPFLOATNUM>

<MEFLOATNUM> := <小数点付10進数>E<INT>

<MEPFLOATNUM> := <MEFLOATNUM>P<INT>

意味: FLOATNUMは任意の精度指定がおこなえ, 指数部は任意多倍長表現をとる。(指数部のオーバーフロー, アンダーフローは起こらない)。ME型(Mantissa-Exponent型, <MEFLOATNUM>のこと)の精度は仮数部の桁数により, MEP型(Mantissa-Exponent-Precision型, <MEPFLOATNUM>のこと)の精度はP部の正整数で指定される10進桁数以上をとる。MEP型を設ける理由は有効精度を示すためのゆの羅列を省略するため

である。例えば　2.150000000000E2 = 2.15E2P13

同等性:　小数点上位桁が非零数字(FORTRANの1P表現)で表わされるME表現が一致する事。(2個の〈FLOATNUM〉の差が小さな数ε,例えばε = 10.E-20以内ならば等しいとみなす様な方法はとらない。その理由は第1に等号の推移律を保障するためで,第2には指数部のとりうる負整数値に制限がないためである。)

2.1.4 LIST

標準外部表現:　LISPと同じ。

意味:　LISPでいうアトム以外のS式。特にNIL以外のアトムで終るリストは'dot-end'リストと呼ぶ。後記VIN, AMT, CATもリストの要素になりうる。

同等性:　LISPのequalによる。ただし,VIN, AMT, CATの同等性も後記の定義によって調べるものとする。

2.1.5 VIN (Vector Index Number)

標準外部表現:　$(VIN^* , v_1^* , \cdots , v_n^*)$　$v_i^* \in ANY$ (ANYは任意のデータ型の集合). VIN^* はVINであることを表わす記号。以後*で終わるアトムはデータ型記述のための記号を示すもので,内部表現がこの様なリスト構造をもつことを意味しない。(各データ型に対する基本演算は個々に定義され,それ以外の演算は許されない。例えばVINのcar, cdrを取る事は許されない。)

意味:　VINは概念上 (V^*, i, n) の3つ組として考える事ができる。V^*は$n+1$個の連続した記憶領域を示すシステムポインタ(システムの記憶管理モジュールによって管理され,ユーザはふれられない。)であり,nは非負整数であり,iは$0 \le i \le n$なる整数である。

同等性:　2つのVIN (V^*, i, n) と $(V^{*'}, i', n')$ に対して

① $n - i = n' - i'$

② $0 \le j \le n - i$ なるjに対して　equal$(V^*(i+j), V^{*'}(i'+j))$

を満たすこと。

基本演算:

creation	:	mkvin(n)　nは非負整数 n+1の長さの連続領域をとり,$(V^*, 0, n)$を値として返す。
insertion	:	putvin(vin*, j, val*)　, vin*: (V^*, i, n) $i+j \le n$ であれば　$V^*(i+j) \leftarrow val^*$ $i+j > n$ であれば　エラー
read	:	getvin(vin*, j, val*) $i+j \le n$ であれば　$V^*(i+j)$ $i+j > n$ であれば　エラー
add vin	:	advin(vin*, j) $i+j \le n$ であれば　$(V^*, i+j, n)$ $i+j > n$ であれば　NIL
upper bound	:	upvin(vin*) $n-i+1$

演算例	演算	結果	VINの内容
	a := mkvin[2]	(V*,0,2)	a:(VIN* , NIL_0 , NIL_1 , NIL_2)
	putvin[a;0;A]	A	a:(VIN* , A_0 , NIL_1 , NIL_2)
	putvin[a;2;C]	C	a:(VIN* , A_0 , NIL_1 , NIL_2)
	getvin[a;0]	A	a:(VIN* , A_0 , NIL_1 , NIL_2)
	b := advin[a;1]	(V*,1,2)	b:(VIN* , NIL_1 , C_2)
	getvin[b;1]	C	b:(VIN* , NIL_1 , C_2)

意義及び応用: 多くのプログラム言語に存在する連続領域を伴なうデータ型に対する領域のプロテクションは非常に大切な問題である。プログラムエラーの多くがこれをおこなっていないために起こることを考えれば，コンパイル時にチェックできない，実行時間が多少落ちるといった理由でプロテクションの機構をなくしてしまうのは危険すぎると考える。FLATSでは，動的に作られる連続領域に対するプロテクション機構，そのオーバーヘッドの減少，アドレス計算を速くするのに適していると思われるVINを導入した。

Standard LISPには実行時に作られるVECTOR（Lisp 1.5のarrayとは異なり，グローバルな名前を持たない。）が導入されている。VINによってVECTORとその演算は次の様に定義できる。

```
mkvect[n] = mkvin[n]

putv[vect;j;val] = putvin[vin;j;val]

getv[vect;j] = getvin[vin;j]

upbv[vect] = upvin[vin]
```

VECTORはインデックス i が常に 0 であるVIN（V*, 0, n）と見なす事ができる。

VINに対する演算には，そのインデックス i を減少させるものは存在せず，従ってVINを線型化されたリストとも考える事ができる。（但し，rplacd に相当する演算は存在しない。） VINの諸性質から，FLATSの"A"に対しても，配列の受け渡し，プロテクション，COMMON, EQUIVALENCE, call by refrence などの機構に有効なデータ型であると考える。

2.1.6 AMT (Address Mapping Table)

標準外部表現: (AMT* , (m_1^* ... m_n^*)) $m_i^* \in$ GID

例) (AMT* , NIL)

(AMT* , (A B C D))

(AMT* , ((A 1) (B 2) (C 3) (D 4))) (A 1),... ∈ HLIST

意味: AMTは概念上，GIDの集合（unordered set） $\{m_1^*, \ldots, m_m^*\}$ を表わすものである。

同等性: 2つのAMTを要素の集合としてみて同じである。

基本演算:

creation : mkamt ( )
(AMT*, NIL) を作り，値として返す。

insertion : putamt (amt*, m*)

$m^* \in \{m_1^*, \ldots, m_n^*\}$ ならば AMT はそのまま, NIL を返す。

$m^* \notin \{m_1^*, \ldots, m_n^*\}$ ならば AMT は $\{m_1^*, \ldots, m_n^*\} \cup \{m^*\}$ に変わり, T を返す。

deletion : remamt (amt*, m*)

$m^* \in \{m_1^*, \ldots, m_n^*\}$ ならば $\{m_1^*, \ldots, m_n^*\} - \{m^*\}$, T を返す。

$m^* \in \{m_1^*, \ldots, m_n^*\}$ ならば $\{m_1^*, \ldots, m_n^*\}$, NIL を返す。

membership : getamt (amt*, m*)

$m^* \in \{m_1^*, \cdots, m_n^*\}$ ならば T。

$m^* \notin \{m_1^*, \cdots, m_n^*\}$ ならば NIL。

演算例

演算	結果	
a := mkamt[]	(AMT* , NIL)	a:(AMT* , NIL)
putamt[a;A]	T	a:(AMT* , (A))
putamt[a;B]	T	a:(AMT* , (A B))
getamt[a;A]	T	a:(AMT* , (A B))
remamt[a;C]	NIL	a:(AMT* , (A B))
remamt[a;A]	T	a:(AMT* , (B))

意義及び応用: AMTは集合を表現するものであり, 従来のLISPにおけるflagと同種の演算が定義されるデータ型である。flagとAMTの重要な違いは, 前者が特定のアトムにグローバルに固定されているのに対し, 後者はプログラム内で自由に作成し, 持ち運びができる点にある。このため, flagの表示子の名前の衝突の問題が解決されている。またユーザによるflagの管理も大巾に軽減される(不必要なAMTはガーベッヂとなり回収される)。AMTの構造も一切ユーザには見えなくなっているため, flagの場合の様なサイドイフェクトも引起さない。

2.1.7 CAT (Content Addressed Table)

標準外部表現: (CAT* , e^* , (m_1^* ... m_n^*) , (v_1^* ... v_n^*)) $e^*, m_i^* \in$ GID, $v_i^* \in$ ANY

(e^* は省略可)

例) (CAT* , NIL , NIL)

(CAT* , 0 , (X Y Z) , (1 2 3))

意味: CATは概念上 (e^*, amt*, f^*) の3組として考える事ができる。e^*は, "excise value", amt* は AMT, f^*は $f^*: m_i^* \in \{m_1^*, \ldots, m_n^*\} \to v_i^* \in$ ANY なる写像

同等性: AMTを要素の集合とみて等しく, 写像が等しいこと。

基本演算:

creation : mkcat ( ) または mkcat (e^*)

(mkamt ( ), f^*) または (e^*, mkamt ( ), f^*) を作り, 値として返す。

insertion : putcat (cat*, m*, val*)

$m^* \in$ amt* で val* = e^* ならば remcat (cat*, m*)。

val* ≠ e^* ならば $f^*(m^*) \leftarrow$ val*。

$m^* \notin$ amt* で val* = e^* ならば なにもしない。

val* ≠ e^* ならば amt* $\cup \{m^*\}$;

$f^*(m^*) \leftarrow$ val*。

read : getcat(cat*, m*)
$m^* \in amt^*$ ならば $f^*(m^*)$
$m^* \notin amt^*$ ならば e^*

deletion : remcat(cat*, m*)
$m^* \in amt^*$ ならば $amt^* - \{m^*\}$。
$m^* \notin amt^*$ ならば なにもしない。

演算例.)

演算	結果	CATの内容
a := mkcat[0]	(0,amt*,f*)	a:(CAT* , 0 , NIL , NIL)
putcat[a;TOM;5]	5	a:(CAT* , 0 , (TOM) , (5))
putcat[a;BOB;3]	3	a:(CAT* , 0 , (TOM BOB) , (5 3))
getcat[a;TOM]	5	a:(CAT* , 0 , (TOM BOB) , (5 3))
getcat[a,137]	0	a:(CAT* , 0 , (TOM BOB) , (5 3))
remcat[a;TOM]	0	a:(CAT* , 0 , (BOB) , (3))

意義及び応用: CAT はキー(key)と値(value)を持つデータ(variable, p-list, function name)を管理するデータ型である。LISPのp-listもflag同様,特定のアトムに固定されるため,やはり表示子の名前の衝突,ユーザによる管理,サイドイフェクト等の問題を持つが,AMTと同様に,CATよではこれらの問題が解決される。

更に,一般のLISPではアトムに対して,value部,p-list部,function部などが固定されている場合が多く(全てp-listで表わせば,参照時間が遅くなる。),実際にはこれらの領域が無駄になっている場合が多い。FLATSの'L'では,領域が必要になった時点で始めてCAT上に確保するため,領域の節約にもなる。(参照時間は $Oh^*(1)$)

2.2 Intern データ型

2.1で定義した全てのExternデータ型はInternデータ型(H型,GIDとも呼ぶ)に変換(この操作をinternするという,関数internあるいはhがこれをおこなう)できる。Internの際には,各データ型の同等性に従がいそのユニークエを保障し,同等性のチェックはLISP基本関数eqによって$O(1)$でおこなえる。Intern型データの書き換えはおこなえず,データのプロテクションにも利用できる。

Internデータ型は,Externデータ型の名前の前に"H"をつけて表わす。またExternデータ型の定義のANYの部分をGIDに置したものがInternデータ型の定義となる。

2.3 Sweep 演算

LIST, AMT, CATおよびそれらのH型に対して,sweep演算を定義する。sweepはこれらデータ型の要素を順に取りだし,それに対してユーザの定義した演算を作用させる。LISPのmapcに類似した演算である。特にAMT, CATに対するsweepは非常に強力である。

LISTに対するsweepは次の様に定義する。

```
list ∈ (LIST ∪ HLIST)
sweep[list; [[m]; op]
sweep[list; [[m]; op] = mapc[list; [[m]; prog[[]; op]]]
```

同様に AMT, CAT に対しては

```
a ∈ (AMT ∪ HAMT)
sweepamt[a; [m]; op]
c ∈ (CAT ∪ HCAT)
sweepcat[c; [m;f]; op]
```

m,f は sweep 中の要素，値を設定する変数であり，op は prog の本体と同様に定義できる。要素がなくなった時点（全ての要素が sweep された）で終了する。sweep の詳細な例は 3 章に示す。

3. 応用例

前章で導入したデータ型を使用して，これらのデータ型及び sweep の有効性を示すとともに，いくつかの応用例としてプログラムを示す。プログラムで用いられている詳しいアルゴリズム等は文献[10,11]を参照されたい。(sweep 演算は，アルゴリズムを簡明に説明するため及びプログラム記述の面からも強力なものである。)

LISP には普通宣言が存在しないが，型宣言がある方がアルゴリズムの意味を把握し易い。ここでは型宣言のために データ型を表わす大文字をその型に属する データの集合とみなし，集合算記法を使用する。例えば x ∈ (CAT ∪ HCAT) は，x が CAT または HCAT 型のデータであることを表わす。型宣言のある prog 中では次の略記法を許す。(宣言がないと，関数との名前の混同が生ずる恐れがある)
また四則演算，集合算には infix 記法も許す。

```
putcat[c;m;v] ------- c(m):=v        setq[x;plus[x;5]] ------- x:=x+5
getcat[c;m] --------- c(m)
getamt[a;m] --------- m∈a
```

3.1 x ∈ (CAT ∪ HCAT) の要素数を調べる関数 sizecat[x] は次の様に表わす事ができる。AMT, HAMT についても同様に定義される。

```
sizecat[x] = prog[[n]; x∈(CAT ∪ HCAT)
                       n:=0;
                       sweepcat[x;[]; n:=n+1];
                       return[n]              ]
```

sizecat の時間計算量は O(n) である。

3.2 x ∈ (CAT ∪ HCAT) をコピーする関数 catcopy[x;e] (e は excise value) は，

```
catcopy[x;e] = prog[[c]; x∈(CAT ∪ HCAT); c∈CAT;
                         c:=mkcat[e];
                         sweepcat[x;[m;f];c(m):=f];
                         return[c]                  ]
```

catcopy の時間計算量は $Oh^*(n)$

3.3 $x,y \in (CAT \cup HCAT)$ の全ての要素について，その値の和をとり新しい CAT を作成する関数 catadd[x;y]（差をとる関数 catsub は '+' を '-' に置き換える。）

```
catadd[x;y] = prog[[c]; x,y∈(CAT∪HCAT); c∈CAT;
                        c:=catcopy[x;0];
                        sweepcat[y;[m;f]; c(m);=c(m)+f];
                        return[c]                        ]
```

catadd および catsub は文献[10.11]の SP表現多項式に対して，加減算をおこなえる。

```
catadd[(CAT*,0,(A B),(2 3));(CAT*,0,(A C),(1 4))]
```

をおこなえば，(CAT*,0,(A B C),(3 3 4)) が得られる。（(2a+3b)+(a+4c)= 3a+3b+4c）
catadd の時間計算量は $Oh^*(|x|+|y|)$ である。

3.4 タイパーの打ったテキストと，正しく入力されているテキストとの違いを次のプログラムによって調べる事ができる。t(text), d(dictionary) ∈ (LIST ∪ HLIST) とし，d には現われていない単語を要素，出現度数をその値に持つ CAT を作成する。その CAT は lexsort という関数によって辞書式に編集されるものとする。

```
   d,t∈LIST; a∈AMT; c∈CAT;
   a:=mkamt[];
   c:=mkcat[0];
1: sweep[d;[m]; putamt[a;m]];
2: sweep[t;[m]; [m∈a ⟶ c(m):=c(m)+1]];
   print[lexsort[c]];
```

例えば，d = (THE QUICK BROWN FOX JUMPS OVER A LAZY DOG) に対して，何回かテキストを入力した結果は，(CAT*,0,(HTE SOG),(2 3)) と出力されるだろう。

3.5 3.3と同様に表現される2つの多項式 p,q ∈ (CAT ∪ HCAT) に対する乗算をおこなう関数 catmul を示す。（文献[10,11]参照）

```
catmul[p;q]=prog[[c]; p,q∈(CAT∪HCAT); c∈CAT;
                      c:=mkcat[0];
                      sweepcat[p;[tp;fp];
                               sweepcat[q;[tq;fq];
                                        prog[[t];
                                             t:=h[catadd[tp;tq]];
                                             c(t):=c(t)+fp*fq]]]
                      return[c]                                  ]
```

p = (CAT*,0,((HCAT*,0,(A B C),(1 2 4))),(3))　($3ab^2c^4$) に対して,

q = (CAT*,0,((HCAT*,0,(A B D),(3 1 5))),(10))　($10a^3bd^5$)

catmul[p;q] = (CAT*,0,((HCAT*,0,(A B C D),(4 3 4 5))),(30))　($30a^4b^3c^4d^5$)

が得られる。catmulの時間計算量は $O(n^*(\Sigma t_{p,i} * \Sigma t_{q,i}))$ となる。

3.6 やはりSP表現多項式P(非負次数)を微分.積分する関数 catdif[p;v], catint[p;v] は次の様になる。

```
catdif[p;v]=prog[[c]; p∈(CAT∪HCAT); c∈CAT;
                      c:=mkcat[0];
                      sweepcat[p;[t;f]; prog[[ct;ev]; ct  CAT;
                                                      ct:=catcopy[t;0];
                                                      ev:=ct(v);
                                                      ct(v):=ev-1;
                                                      c(h[ct]):=f*ev]]
                      return[c]                                        ]
```

catint[p;v] は上のプログラムの '-1' を '+1', '*ev' を '/(ev+1)' と置き換える。

p = (CAT*,0,((HCAT*,0,(X Y),(3 1))(HCAT*,0,(X Y),(2 2))),(3 2)), v = X に対して

catdif[p;v] = (CAT*,0,((HCAT*,0,(X Y),(2 1))(HCAT*,0,(X Y),(1 2))),(9 4))

という結果が得られる。($d(3x^3y+2x^2y^2)/dx = 9x^2y+4xy^2$)

実際の数式処理システムでは上記のような演算に際しては、必ず項の整理や同類項の検索,簡約化等の諸操作が複雑に繰返されているが,CAT,HCATによるSP表現多項式においては以上のとおりである。

3.7 VINは前に述べたとうり, Standard LISPのVECTORを表現する事ができる(インデックス七がゼのVINがVECTORに対応する。)が更にFORTRANにおける記憶領域の共有を次の様に実現できる。

```
DIMENSION  A(100), B(50)
EQUIVALENCE  (A(51),B(1))
```

のような宣言文は,

a:=mkvin[99] ---------- (VIN*, NIL_0 , NIL_1 ,..., NIL_{99})

b:=advin[a;50] -------- (VIN*, NIL_{50} , NIL_{51} ,..., NIL_{99})

次のFORTRAN文は,

```
A(10) = B(10) + 1
```

次の様なLISPプログラムとして実行される。

```
putvin[a;9;add1[getvin[b;9]]]
```

4. まとめ

本稿では、FLATSにおいて実現されるデータ型を抽象化(概念と演算)した形で述べてきた。特にGIDの重要さを示した。このGIDを有効に利用したデータ型AMT, CAT及びsweep演算は、諸集合演算, 数式処理等に応用でき, 速度向上とともに, アルゴリズムの記述にも強力である。 CATは数式の表現ばかりでなく多くの応用が期待できる(索表計算[12]はその一例)。

またVIN(線型化されたリスト表現)を導入することにより, 記憶領域のプロテクションの問題を解決し, Standard LISPのVECTORを完全に包含し, かつ, FORTRANの共通領域, 配列領域を表現する事が可能になった。

数式処理用計算機FLATSが実動した暁には、ここで述べてきたデータ型に対する演算(ハッシング, データ型の動的チェック, スタック操作等々)が強力に, サポートされ, FLATSソフトウェア(LISP, FORTRAN, REDUCE)は高速に実現されることになろう。

References

[1] McCarthy, J., Recursive Functions of Symbolic Extressions and Their Computation by Machines, Part 1, Cacm, Bol. 3, pp.184-195, 1960.

[2] Knight, T., The CONS Micro Processor, MIT AI Working Paper 80, MIT, Cambridge, Mass., Nov. 1974.

[3] Van der Poel, W. L., A Manual of HISP for the PDP-9, Univ. of delft, Netherlands, 1974.

[4] Hearn, A. C., STANDARD LISP, Stanford Artificial Intelligence Report Memo AI-90, May 1969.

[5] Marti, J. et al., STANDARD LISP REPORT, UCP-60, Univ. of Utah, Utah, Jan. 1978.

[6] Goto, E., Monocopy and Associative Algorithms in an Extended LISP, Information Science Lab., Technical Report 74-03, Univ. of Tokyo, Apr. 1974.

[7] Hearn, A. C., REDUCE-2 User Manual (2nd ed.), UCP-19, Univ. of Utah, Utah, Mar. 1973.

[8] McCarthy, J. et al., LISP 1.5 Programmer's Manual, The MIT Press, Cambridge, Mass., 1962.

[9] Sassa, M. and Goto, E., A Hashing Method for Fast Set Operations, Information Processing Letters, Vol. 5, No. 2, pp.31-34, 1976.

[10] Goto, E. and Kanada, Y., Hashing Lemmas on Time Complexities with Application to Formula Manipulation, Proc. of ACM SYMSAC 76, Yorktown Heights, N. Y., pp.154-158, 1976.

[11] Goto, E., Algorithms and Programming with CAMs (Content Addressable Associative Memories), Dept. of Information Science, Technical Report 78-05, Univ. of Tokyo, Aug. 1978.

[12] Goto, E. and Terashima, M., MTAC - Mathematical Tabulative Automatic Computing, Dept. of Information Science, Technical Report 77-03, Univ. of Tokyo, Oct. 1977.

[13] Ida, T. and Goto, E., Overflow Free and Variable Precision Computing in FLATS, Journal of Information Processing, Vol. 1, No. 3, 1978.

[14] Terashima, M. and Goto, E., Genetic Order and Compactifying Garbage Collectors, Information Processing Letters, Vol. 7, No. 1, pp.27-32, Jan. 1978.

[15] Ida, T. and Goto, E., Performance of a Parallel Hashing Hardware with Key Deletion, Proc. of IFIP Congress 77, North-Holland, 1977.

[16] Ida, t. and Goto, E., Analysis of Parallel Hashing Algorithms with Key Deletion, Journal of Information Processing, Vol. 1, No. 1, pp.25-32, 1978