Vol. 22 No. 5 情報処理学会論文誌 Sep. 1981

SNOBOL 3 言語のべたづめ方式処理系とその評価†

角田 博保††

SNOBOL 3 言語の新しい高速処理方式（べたづめ方式）について述べ，その評価を行う．SNOBOL 3 は SNOBOL 族の言語のうちでは一世代古いものに属するが，最近の技術を適用して実現してみたところ，処理効率が大幅に向上し，新たな応用分野が開けてくることがわかった．たとえば，実用性のあるソフトウェア工具を作るのに，SNOBOL 3 を使うことが可能になった．本文では，本処理方式の領域管理とパターン照合について言語との関連において論ずるとともに，性能についての評価を行う．その評価をもとに，いっそう高速の処理方式案を示す．また，ほかの文字列処理用言語の処理系で使われている方式との比較を行う．最後に，SNOBOL 3 の新応用分野についてもふれる．

1. 序

SNOBOL 3[1] は SNOBOL 族の言語のうちでは一世代古いものに属するが，それなりにまとまりがよく，小規模で処理系が作りやすいという面では捨てがたいものをもっている．筆者は（なかば偶然の機会から）過去数年にわたって SNOBOL 3 の新方式の処理系を開発してきた[2]．昔の言語を現代のハードウェアにおいて，新しいアイディアをとり入れて実現した結果，たとえば SNOBOL 4[3] には考えられないような応用が開けてくることがわかった．本文では，この SNOBOL 3 処理系の中心的なアイディアを記述し，実際にどの程度の性能が得られたかを示す．また効率についての検討を行い，いっそうの高速化を行うためにはどんな方式が考えられるかについても，数量的根拠を示して簡単にふれる．

本処理方式の特徴は，基本データである文字列を「べたづめ」(連続記憶領域に文字バイトを詰めて格納)によって表現し，いわゆる第3世代の電子計算機に広く普及している事務処理用命令を最大限に利用することによって処理の大幅な高速化を計っている点にある．また，全体にデータ構造が単純化されたことにより，機能の拡張性と機種独立性が高まっている．

SNOBOL 族の言語は可変長文字列を処理の基本データとしているので，文書処理等の応用に大変使いやすい．しかし，できたプログラムがあまり遅くては実用的なソフトウェア工具[4]として役立てることはできない．

いま SNOBOL 4 を SNOBOL 3 と比較してみると，SNOBOL 4 は SNOBOL 3 を改良して作られたため，言語概念的には SNOBOL 3 を含んでおり，① 共通な部分（文字列操作等），② SNOBOL 3 のと概念的には同じであるが実現のしかたが異なる部分（パターン照合等），および③新たに導入された部分（配列型等）から成っている．SNOBOL 3 は①と②を合わせたものに当たっており，機能的には小さいが，工具として使うには十分な場合も多い．③の部分は現処理方式で導入したデータ構造を利用して簡単に実現できる場合が多いので，問題は②の部分である．たとえばパターン照合について比較してみると，SNOBOL 3 では次章で述べるような高速化が可能であるが，SNOBOL 4 では，パターン型が導入されたこと，およびパターンが一般化されたことにより高速処理がむずかしくなっている．また，処理全般にわたり一様化と機能拡張を行っているので，記述能力は増し，より高度な記号処理には向いているが，必然的に実行時検査がふえ，高速性が落ちることになっている．ソフトウェア工具の領域では，SNOBOL 3 にそなわっている程度の簡単な機能で十分であるような応用も多い．そのような場合にはより高速に処理できるという点で，SNOBOL 4 よりむしろ SNOBOL 3 の方がふさわしいと考えられる．実際，実用にするに十分な速度が前者で書いたのでは得られず，後者で書けば得られる，というような工具の例は少なくない．

実際の処理系試作はまず FACOM 230-45 S において行い，のちに HITAC M 180 上へ移植した[5]．本処理方式による処理系は，現時点では FACOM 230-45 S，同 230-38（以上 SN 3/45 S 系と略す），HITAC

† A High Speed SNOBOL 3 Processor with Close-Packed Data Structure and Its Evaluation by HIROYASU KAKUDA (Department of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科

M 180, 同 M 200 H, および FACOM M 200 (以上 SN 3/180 系と略す) 上で使われている. おもに, ファイル複写やファイル内の一律文字列置き換えといった簡単な道具として, また一部はプリプロセッサ (たとえば Ratfor 用) のようなやや重い工具としても使われている.

2. SNOBOL 3 言語のべたづめ処理方式

処理系は**図1**のような構造をしている. コンパイラは SNOBOL 3 プログラムを入力し, データ領域に名前, 文字列を割り付け, 目的コード領域に目的コードを出力する. 翻訳終了後ひき続いて目的コードの実行に移る. 実行は支援手続き群を利用して行われる. 目的コードは機種の特徴を生かして, SN 3/45 S 系では機械語, SN 3/180 系では解釈コード (threaded code) から構成している.

本処理方式の特徴の一つは, 事務処理用命令を利用して, 文字列の移動, 1文字の探索, および2文字以上の探索を高速化している点である. SN 3/45 S では, それぞれ MV (Move) 命令, SRBE (Search Byte Exclusive OR) 命令, および T (Translate) 命令を利用している. SN 3/180 では, 移動には MVC (Move Characters) 命令を使い, 文字の探索 (1, 2文字) には TRT (Translate and Test) 命令を使用している.

以下, 処理系にとって基本的な「領域管理」と「パターン照合」について詳しく述べる.

2.1 領域管理

SNOBOL 3 は可変長文字列を扱い, 変数, 関数を実行時に生成できるので, 文字列, 名前をいかに管理するかが処理系構成上大切な点である.

2.1.1 データ領域

データ領域はハッシュ表, 文字列領域, 自由領域, および名前領域から成っている (**図2**). 文字列領域と名前領域とは固定した領域の両端からとられ, その間が自由領域になる. 文字列領域は番地の低い方から高い方へ (BASE→STRTOP), 名前領域は高い方から低い方へ (CEIL→NAMETOP) 伸びる. 自由領域 (STRTOP～NAMETOP) がある大きさより小さくなると領域再生が行われ, その際文字列領域が縮小され, その分自由領域がふやされる.

2.1.2 文字列領域

文字列領域には文字列を表現するためのべたづめブロックがとられる. べたづめブロックはハードウェアの語境界に合わせて, 先頭から長さ欄, 作業用欄, および値欄から構成される. 長さ欄と作業用欄は各々1語からなる. この1語はハードウェアに都合のよい大きさにとられ, SN 3/45 S では 16 ビット, SN 3/180 では 32 ビットである. 作業用欄は領域再生時に使われる. 値欄は1文字当たり1バイトで文字を格納する. **図3**に文字列 'ABC' を表わすべたづめブロックを示す.

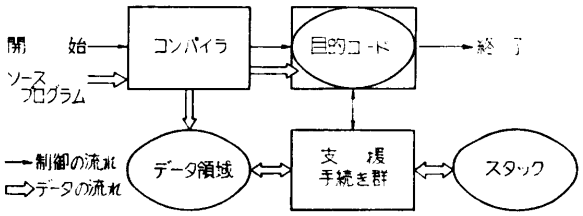

図1 処理系の構造
Fig. 1 Structure of the SNOBOL 3 processor.

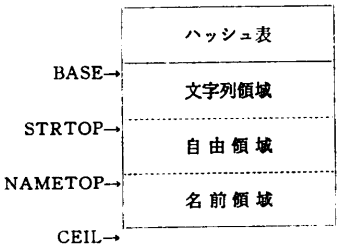

図2 データ領域
Fig. 2 Storage layout.

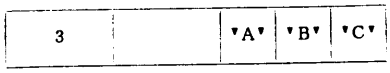

図3 べたづめブロック (文字列 'ABC')
Fig. 3 Representation of the string 'ABC'.

値欄に格納するものは文字列に限らない. 相対的データ (ポインタ) でなければ何であってもよい.

2.1.3 名前領域

SNOBOL 3 で使われる名前には3種類ある (変数名, ラベル名, および関数名). 字づら上は同じ名前でも使われかたによって意味が変わりうる. また, 名前は動的に生成されうる.

本処理方式では字づら上同じ名前を一括して一つの名前ブロックであらわしている. 名前領域は6語からなる単位に分割され, その1単位をそのまま扱う場合 (6語ブロック) と n 単位 ($n \geq 2$) を連結して扱う場合 (6n語ブロック) とがある. その区別は先頭の1語の先頭の1ビットが0か1かによってつける.

名前ブロックは6語ブロックであり6欄から成っている (**図4**). そのブロックで表わす名前の種類 (変数,

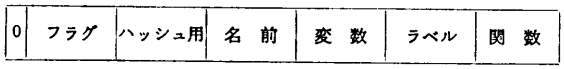

図 4 名前ブロック
Fig. 4 Layout of name blocks.

ラベル，関数）を示すフラグ欄，ハッシュ用欄，名前の字づらを表わしたべたづめブロックへのポインタを格納する名前欄，変数の値を格納する変数欄，対応するプログラム点を格納するラベル欄，および関数本体の入口番地を格納する関数欄がとられる．定義関数を表わす場合は，関数欄は関数ブロック（引数と局所的変数の個数およびそれらに対応する名前ブロックの番地を格納した $6n$ 語ブロック）を指す．

名前領域の特徴は各ブロックの再配置を行わない点（移動不可）と内部に相対データ（ポインタ）を含む点である．OS 用の入出力制御ブロック（内部にバッファを含む）等も同じ性質をもつので $6n$ 語ブロックとして名前領域内にとられる．

2.1.4 ハッシュ法と名前管理

字づら上の名前と名前ブロックとの対応づけはハッシュ法による．目的コード上では静的な名前（コンパイル時に確定）は名前ブロックの番地で代表できるが，実行時に生成される名前についてはそのつど名前ブロックとの対応づけを行う必要があり，その対応づけ操作の速度は処理系全体の効率に大きく影響する．ここでは固定長のハッシュ表を使い，衝突処理は名前ブロック内ハッシュ用欄を使った連鎖法によった．

SNOBOL 3 においてはハッシュ関数 h（可変長の文字列からハッシュ表内インデックス値への写像）をどう設定するかが重要である．SNOBOL 3 で実行時に生成される名前にはある規則に従うものが多いので，その規則性を保存してしまうような関数はよくない．よく使われる除算法では，たとえば I 0, I 1, …, K 0, K 1, … と生成された名前でたまたま $h(\mathrm{I}0)=h(\mathrm{K}3)$ となった場合，$h(\mathrm{I}1)=h(\mathrm{K}4), h(\mathrm{I}2)=h(\mathrm{K}5), \cdots$ のように衝突が激増してしまう．排他的論理和＋シフト法，加算＋シフト法等を実データをもとに検討した結果，一番衝突の少なかった乗算法を採用した[2]．すなわち文字列を 16 ビット（2 文字）単位の数値列とみなし，先頭の 2 単位を乗算しその結果（32 ビット）の中央 16 ビットに次の単位を乗算し以下同様に続け，最後に得られた値を表の大きさで割って余りをとるという方法による．

2.1.5 領域再生

領域再生は，文字列の詰め合わせと文字列へのポインタ値の調整によって行う．文字列ポインタを保持している可能性のある場所は，名前ブロックの名前欄と変数欄および実行時スタックだけであり，再帰的なポインタ探索は不要である．ポインタ保持場所の大きさによって 2 通りの方法を考えた．

方法 1. 二重探索法

1）ポインタ保持場所を順に調べ，ポインタ値が gcbase（コンパイル終了時点での STRTOP の値）以上のもの（有効ポインタと呼ぶ）に対して，それが指すブロックに印をつける（先頭 1 ビットを 1 にする）．

2）文字列領域を gcbase より番地の高い方へ順に調べ，印付きブロックのみを詰め合わせた場合に得られるはずの新番地をその作業用欄に格納する．

3）再度有効ポインタ保持場所を探し，そのポインタ値をそれが指すブロックの作業用欄の値（新ポインタ値）で置き換える．

4）文字列領域内を再度たぐり，印付きブロックの印消しと移動を行う．

この方法はポインタ保持場所の大きさに余分な要求をつけていない．つまり文字列ポインタ値を表現するに足るビット数あればよい．これに対し，もしポインタ保持場所の大きさが任意の番地を格納できるほどあれば，次の高速な方法が使える．

方法 2. 逆ポインタ法

1）有効ポインタ保持場所を順次探し，それが指すブロックに印を付けるとともに，そのブロックの作業用欄からそれを指すポインタ保持場所をたぐれるようにポインタの付けかえを行う．

2）文字列領域内をたぐり，各印付きブロックに対して，まず詰め合わせ後に得られるはずの新番地を計算し，作業用欄からリンクをたどってポインタ保持場所に書き込む．次に，印を消し，作業用欄に 0 を入れる．最後にそのブロックの移動を行う．

方法 2 の手順 2）でポインタの変更とブロック移動を同時に行えるのは，移動されるブロック内にポインタを含んでいないことによる．

SN 3/45 S 系ではポインタ保持場所の番地を表わすのに 17 ビット必要となり，作業用欄（16 ビット）におさまらないので，方法 1 によった．SN 3/180 では番地は 24 ビット，1 語は 32 ビットなので方法 2 が使える．

2.2 パターン照合

SNOBOL 3 のパターンは構文上区別されるパターン要素の任意の列として構成される．パターン要素に

は，文字列定数，任意文字列変数，平衡文字列変数(balanced string variable)，固定長文字列変数，および戻り参照の5種がある．各パターン要素はその種類に対応して照合 (forward matching) 規則と再照合 (rematching) 規則とが定義されており，パターン照合は要素の照合が成功すれば次の要素の照合へ，失敗すれば直前の要素の再照合へという流れにそって行われる[1]．ただしパターン照合処理としてはこの手続き通りの照合結果が得られればよいので，以下で述べるように照合過程を変更して高速化することが可能である．

(1) 新パターン要素の導入

任意文字列変数の直後に空でない文字列定数があるパターンはその文字列の探索を意味しているとみてよい(図5の例1)．これを定義通りに照合すれば任意文字列変数の再照合が繰り返されることになる（例1では4回)．そこでパターン照合時にこういう状態の文字列定数をみつけて「探索文字列」にすりかえ，その照合に探索命令を使うことで大幅な高速化ができる．同様に平衡文字列変数が直前にある場合(図5の例2)にも「平衡探索文字列」の概念を導入して処理することにした．これはカッコのバランスをとりながら文字列を探す操作であり，実現は '(',')'，あるいはその文字列の先頭1文字のどれかを探す命令（変換命令を利用して実現）を利用して行った．

(2) 再照合処理の削減

要素の照合失敗には，対象となる文字列の長さ不足による場合（長さ失敗と呼ぶ）とそうでない場合がある．長さ失敗の場合，その要素に対応している対象文字列内位置 (cursor position) が以後パターン照合処理を続行中もはや前戻りしないなら，結局照合全体が失敗に終わることになる．長さ失敗後も照合を続ける必要がある場合を調べてみると，①それ以前で平衡文字列変数がカッコ対を含んだ文字列に照合している場合（図5の例3）と②戻り参照がおこっている場合(図5の例4）の2種類しかないことがわかる．①の場合はその変数の直前の要素の再照合に，②の場合は戻り参照される変数の再照合に移ればよいわけである．そこで，パターン照合中に上記①，②の状況になったときは再照合すべき位置を登録しておくことにした．これにより，長さ失敗がおこった時点で登録があればその位置からの再照合をし，なければ全体の照合を失敗に終わるというように処理の高速化が行える．(SNOBOL 4 における同様の処理に対する討論は5章で行う.)

```
Ex.1:    'ABCDE'      *A* 'E'
Ex.2:    '(ABC)DE'    *(B)* 'E'
Ex.3:    '((ABC)D)E'  *A* *(B)* 'D'
Ex.4:    'ABCDC'      *A* *B* 'D' B
```

図5 パターン照合文の例

Fig. 5 Examples of pattern matching statements.

以上の手法による高速化の度合はパターンの使われかたによって色々と変化するが，経験によれば，プリプロセッサのような応用では総合してパターン照合処理は10倍位高速になっている[2]．

3. 処理方式の評価と効率に対する検討

SN 3/45 S 系処理系をもとにして，①処理効率に対する事務処理用命令の貢献度の調査，②他言語との処理効率の比較，および ③SNOBOL 3 文の典型的な使われかたの調査，を行った．調査には本処理系内蔵の計測機構（文単位で実行の成功・失敗回数および実行時間を計測するもの）を活用した．計測データの整理は本処理系そのもので行った．

3.1 事務処理用命令の貢献度

SN 3/45 S 処理系中の事務処理用命令をそれを使わない同等の手続きへの呼び出しで置き換えて測定用処理系を作成した．典型的な例題 13 件を比較してみたところ，遅くなった割合の最大のもの(ファイル複写)で2.7倍，最小のもの（一斉射撃のプログラム）で1.3倍の時間がかかった．次にこの測定用処理系を改造して，もとの処理系での事務処理用命令の総実行時間を計測したところ，全処理時間の19% から3% の時間（上記プログラムに対応）を費やしていることがわかった．このことから事務処理用命令のシミュレートに約10倍かかることがわかる．この10倍の高速化により事務処理用命令は処理速度に大きく貢献している．しかし，事務処理用命令がこれ以上高速化されたとしても処理時間は高々現在の 81% にしかならない．処理系をさらに高速化するには残りの 81% を消費している部分を細部にわたって検討する必要がある．

3.2 他言語との処理速度の比較

本処理系をファイル内のあるパターンの変換といった簡単な処理に使った場合に実用性のあることはすでに実験済みである[2]（特定の環境下での時間比は，アセンブラ：Cobol：SNOBOL 3：PL/I：Fortran=3：4：12：15：140 であった)．ここではより複雑な処理（ファイルから名前と区切りを字句としてとり出す処理）をとり上げ，Pascal (Trunc Pascal をもとにした

```
LOOP  BUF *CHAR/'1'* =          /F(NEXT)
      '01 ... 9A ... Z' CHAR    /F(DELIMITER)
      TOKEN = TOKEN CHAR        /(LOOP)
```

(a)

```
1: case BUF[i] of
  '0','1', ... ,'9','A', ... ,'Z':
    begin TOKEN[j]:=BUF[i]; i:=i+1; j:=j+1 end;
  '!', ... ,'>':
    begin i:=i+1; goto 2 end;
   end; goto 1;
```

(b)

図 6 SNOBOL 3 と Pascal のプログラム例 (字句解析)

Fig. 6 Sample lexical analyzers written in (a) SNOBOL 3 and (b) Pascal.

FACOM 230-45 S 用処理系を利用) との比較を行った.

入力カード 68 枚に対し, Pascal 版 478 ms, SNOBOL 3 版 3,115 ms で約 7 倍の差があった. この速度差は字句の切り出し部分を比較してみれば納得がいく (**図 6**). Pascal 版では配列要素の参照と代入だけで済むところで, SNOBOL 3 版では文字列の切りとりと追加が行われる. このような文字列の追加, 先頭からの削除といった, よく利用され, かつ Pascal 等では簡単に処理できるような文を最適化することが望ましい.

3.3 SNOBOL 3 文の典型的使われかた

典型的なプログラム 2 例について文の種類で分類した利用状況を**表 1** に示す. その処理に応じて文の利用状況は色々であるが, 代入文のうち追加型のもの, 単純なもの (右辺単項) が多く使われ, また置き換え文では先頭からの削除が多いのがわかる. 一律なコード生成ではなくよく使われる文を特別に高速化するような最適化をするべきであり, 単純な文がよく使われていることからみて, それはたしかに可能である.

表 1 SNOBOL 3 文の使われかた

Table 1 A performance profile of SNOBOL 3 statements.

文の種類	例1. ベクトル処理		例2. 字句解析	
	実行回数率 (%)	平均実行時間 (μs)	実行回数率 (%)	平均実行時間 (μs)
単項	7.5	435	7.3	260
パターン照合	13.0	876	29.3	480
代入	34.8	551	30.5	405
追加	34.7	701	60.0	362
右辺単項	30.1	389	16.0	201
置き換え	44.7	1,313	32.9	675
先頭削除	78.8	868	77.8	667

4. 処理系の高速化について

本文で示した SNOBOL 3 処理系はすでに約 5 年間にわたって実用に供されてきたが, その経験に基づいて見なおしてみると, 一般的利用状況のもとでの平均的処理効率はさらに大幅に向上できることがわかる. ここでは現処理方式についてどのような改良が可能であるかを簡単に指摘しておくことにする. その詳細については, これらの考えを取り入れた新処理系 (目下開発中) が完成した段階で, あらためて報告することにしたい.

処理系の高速化にはプログラム全般に一律に効くものと特定の利用状況にのみ効くものが考えられる. 一律な高速化手法として, ①目的コード基本操作の詳細化およびその場展開 (inline expansion), ②パターン要素の実行時判定処理の軽減, および ③スタックあふれ検査回数の縮小, を考案した.

現在, 目的コード基本操作はスタックを介して情報交換する 28 種のルーチンからなり, 複雑な文も簡単な文も一様に扱われている. これをレジスタを介せるように大幅に操作数を増し, 単純な文はより高速に処理できるようにする. 新規導入パターン (探索等) の判別をできるだけコンパイル時に行えるように, 各要素間の連結関係を詳細に分類してパターン要素数を 22 にふやす. 平均的によく使われる文に対して新目的コードでは机上計算で大体 2 倍以上高速化する見通しが得られた.

特定の状況に効くものとして, ①間接参照の高速化, ②組み込み関数の静的把握, ③文字列の使われかたに応じた内部表現の最適化, および ④自己繰り返し文の最適化, を考案した.

現処理系では間接演算が行われたとき対応する名前ブロックがなければそれを生成するようにしているが, 単なる参照の場合には生成を省略することも可能である. SNOBOL 3 には関数の定義, 変更, 動的呼び出しを行える関数があるので, コンパイル時に組み込み関数を把握することは原則的に不可能であるが, その関数の呼び出しがなかったり, あっても引数がリテラルであったりすれば, 静的把握できる場合がある. 把握できれば最適コードが出せる.

③では文字列表現にいくつかの型をもうける (この型は内部処理用のもので, プログラマからは見えない). 例をあげれば, 非共有可変型では, 文字列はほかの変数に共有されない適当な大きさのブロックで表現

され，追加，削除等はそのブロック内で処理される．こういう特定の処理では Pascal 版と比べても十分に対抗できる処理速度が得られよう．また，入出力型では入出力用バッファをそのまま文字列表現に使う．これらの型は新たな領域を導入することなく，現行のデータ構造で簡単に実現できる．領域再生も若干の変更で現行の方式を適用可能である．

5. 討　論

本処理方式は序で述べたように SNOBOL 4 にも応用できる部分が多い．事務処理用命令の活用は SNOBOL 4 の場合でも文字列をべたづめブロックで扱うことによって可能であるし，配列，表，使用者定義データ型等は名前ブロックで扱える．ただし，パターン照合の高速化はむずかしい．

SNOBOL 4 ではパターンは実行時に任意に構成可能な一つのデータ型として扱われているので，コンパイル時に静的に把握することはかなり無理である．また，直接代入演算子，非評価式，およびパターン照合処理制御子を使ってパターン照合過程そのものを利用した処理をすることができるので，定まった照合過程からほとんど逸脱することはできず，SNOBOL 3 でとったような照合過程の最適化や高速処理パターン要素の導入はかなりむずかしい．SNOBOL 4 では QUICKSCAN モードという照合形態を導入して長さ失敗を取り入れた高速化を計ってはいるが，パターン照合処理制御子を使わない場合でももとのモード (FULLSCAN) との一致性がなく，的確な解決にはなっていない．

次に，文字列処理用言語の処理系として内部構造が公開されているものの中から，MAINBOL[6]（マクロ作成版 SNOBOL 4），MACRO SPITBOL[7]，および Icon[8] をとりあげ，文字列のとり扱いに注目して処理方式の比較を行ってみよう．

MAINBOL では各名前は常変数 (natural variable) という単位で表現され，字づらをあらわす文字列を内蔵している．この文字列は値としても使われる．処理系は移植性を重んじて内部構造の一律化を計っているが，名前と文字列を同一領域で扱っているため領域再生では名前の位置移動もおこる．処理速度は遅い．

MACRO SPITBOL は高速 SNOBOL 4 である SPITBOL をマクロで生成することによって移植性を持たせたものである．処理形態では間接解釈コード (indirect threaded code) を使って実行時の判定作業を軽減したことが特徴の一つである．名前は静的領域，値は動的領域で扱っている．動的領域にポインタの格納も許しているので領域再生ではその分手間どる．総じて本処理方式に最も似ているが，この最後の点で大きく異なる．

Icon は SNOBOL 4 の後継者として考案された言語である．Icon 処理系[9] では文字列領域に文字列本体を，文字列表示領域に文字列表示（長さと本体へのポインタ）を格納する．文字列は文字列表示へのポインタによって表現される．こうすると文字列の切断等は簡単に行えるが，領域再生処理では文字列表示領域内をポインタ値に関して整列する必要が生ずる．

最後に本処理系の応用面について簡単にふれてみよう．まず工具そのものとして使う場合，つまり Fortran のプログラム中から FORMAT を抜き出したいといった，特定の仕事を早く処理すればよい場合には大変重宝である．次に工具を作るための道具としては，たとえばファイルの整形印刷とか変換作業といった簡単な工具を作るのに実用的に使われている．より複雑な工具としては，マクロ処理系や Ratfor プリプロセッサ等が作られ利用されているが，このへんが現処理系の限界であろう．4 章で示した現在進行中の高速化が果たせれば，より高度な応用が開けると思われる．

謝辞　本論文をまとめるにあたり木村泉助教授から数々の助言を頂いた．ここに深く感謝します．

参 考 文 献

1) Farber, D. J., Griswold, R. E. and Polonsky, I. P.: The SNOBOL 3 Programming Language, Bell System Technical Journal, Vol. 45, pp. 895–944 (July-Aug. 1966).
2) 角田博保：SNOBOL 言語の新処理方式に関する研究，修士論文，東京工業大学理学部情報科学科 (Mar. 1976).
3) Griswold, R. E., Poage, J. F. and Polonsky, I. P.: The SNOBOL 4 Programming Language, Second Edition, 256 p., Prentice-Hall, Englewood Cliffs, N. J. (1971).
4) Kernighan, B. W. and Plauger, P. J.: Software Tools, 338 p., Addison-Wesley, Reading, Mass. (1976).
5) 角田博保：べたづめ方式 SNOBOL 3 処理系の移植について，情報処理学会記号処理研究会，4-2 (May 1978).
6) Griswold, R. E.: The Macro Implementation of SNOBOL 4, 310 p., W. H. Freeman, San Francisco, Calif. (1972).
7) Dewar, R. B. K. and McCann, A. P.: Macro

SPITBOL-a SNOBOL 4 Compiler, Software-Practice and Experience, Vol. 7, pp. 95–113 (1977).
8) Griswold, R. E., Hanson, D. R. and Korb, J. T.: The Icon Programming Language An Overview, SIGPLAN Notices, Vol. 14, No. 4, pp. 18–31 (1979).
9) Hanson, D. R.: A Portable Storage Management System for the Icon Programming Language, Software-Practice and Experience, Vol. 10, pp. 489–500 (1980).

(昭和55年12月5日受付)
(昭和56年4月27日採録)