TR-036

論理型言語による文字列処理の記述について

上田和紀（日本電気）
竹内彰一、国藤　進

1983. 12

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

論理型言語による文字列処理の記述について

上田　和紀　（日本電気（株）C&Cシステム研究所）
竹内　彰一　・　関藤　遊　・　古川　順一
（（財）新世代コンピュータ技術開発機構）

0．はじめに

本論文は，並列実行型の論理型プログラム言語Concurrent Prolog に，高効率が期待でき，しかも記述力にすぐれた文字列操作機能を導入する方法について述べる．本研究の目指すところは，複雑な文字列操作のための言語を論理型言語をベースに設計することではなく，むしろ，入出力の際のデータ表現の変換のような，汎用の論理型言語上でのごく普通の処理のための文字列操作機能を設計することにある．

まず第１章では，論理型プログラム言語が，高効率の文字列操作機能を備えなければならない理由を示す．また文字列操作機能を，特に並列実行型の論理プログラム言語上に実現することの利点も示す．

第２章では，「一般的処理のための文字列操作」という観点から，我々が文字列操作に対して求めた要求と，その実現のしかたについての指針を示す．

第３章では，第２章で要求したような機能を実現するために必要な基本メカニズム――文字列型と，それに対する基本操作――の仕様と実現法について述べる．

第４章では，文字列処理の，プログラム上での記述を改善するための方法を示す．この手法を適用すると，通常の論理プログラムとの対応づけを失わずに，強力なパターン表現能力をもったユーザ言語が得られる．

第５章では，提案した機能を用いたプログラム例を示す．

提案する機能の多くは，DEC-20 Prolog 上で動くConcurrent Prolog インタプリタ上で実験中である．

1．論理型言語と文字列操作

1．1．文字列操作の必要性

我々は，与えられた問題を解くプログラムを作成する場合，その問題に適した，さまざまなデータ構造を用いる．しかし，そのプログラムに対する入出力データに関しては，人間とのインターフェースのために，（図形や音声といった例外を別にして）ほとんどの場合，文字列という形態を用いる．そこで，処理に適したデータの内部表現と，文字列表現との変換機能が必要となる．

従来の多くの論理型言語では，このような変換操作は，システムに作りつけの入出力述語の下に隠れていたか，あるいは一文字入出力の述語を用いて手続き的に記述していた．しかし，それらの入出力述語は，実は入出力命令であり，これを含んだプログラムに対しては，宣言的解釈（論理的解釈）が行なえないという大きな問題がある．宣言的解釈を可能にするためには，入力データの全体と出力データの全体を，それぞれひとつの値とみなし，プログラム全体をこのふたつのデータ列の間の関係として記述しなければならない．

このような立場からみた入出力データは，ひとつの文字列（あるいは，文字列の列のような２次元構造も考えられる）を構成しており，それを操作するためには，文字列と文字列の関係，あるいは文字列と他のデータ構造の関係を記述する述語をユーザが定義できる必要がある［上田83，中島83］．つまり，純粋な論理型言語は，一般に文字列操作機能を必要とする．

1．2．文字列型導入の利点

さて，文字列操作を行なうためには，大別して下のふたつの方法が考えられる．

① 文字のリスト（ないしは文字を葉とする木）を文字列とみなす．新たなデータ型は導入しない．
② 文字列型というデータ型を用意する．

本質的な記述能力という点からはどちらをとっても同じであるが，我々は次の理由により，②の方式を検討する．

① 文字列というのは非常に一般的なデータ構造であり，しかも入出力と深い関連をもっているので，処理データ量が多い．従って，その処理効率は重要である．
② 文字列型を導入すれば，べたづめ表現や，機械の大型命令の採用などによって，時間効率，空間効率を大幅に向上できる可能性がある．

①に関しては興味深いデータがふたつある．ひとつは言語処理系の字句解析部に関するもので，［白濱79］によれば，Trunk Pascalから作成したPascal 8000 コンパイラの処理時間の約半分は字句解析部（入力処理を含む）が占めていたそうである．白濱らは，一文字ごとの入力を行単位の入力に置き換える等の改良によって，字句解析部を約 3.5倍に高速化した．もうひとつはDEC-20 Prolog 上での測定結果（筆者・表１）で，8-Queensプログラム（全解探索）をコンパイルして走らせた場合，長さ8のリストを92個出力するための時間が，全処理時間の約１／３を占めていることがわかる．これらのデータから言えることは，入出力と，そのまわりの文字列操作の効率は，プログラム全体の性能に大きく影響するということである．しかも入出力のまわりの処理は，本質的に並列化の困難なところであるから，その比重は，将来の

表1　DEC-20 Prolog (on DEC2060)における8-Queens プログラムの実行時間（単位秒，各5回平均）

	実行時間(A)	うち出力時間(B)	B/A
インタプリタ	42.577	0.885	2.1%
コンパイラ	2.548	0.856	33.6%

並列化技術やコンパイル技術の進歩によって，ますます大きくなることが予想される．

上に示した文字列型導入の理由は，多くのProlog処理系に数値型が導入されている理由とよく似たものである．ただ，数値型に比べて文字列の厄介な点は，文字列というデータ構造が，リストの場合と同様，処理の進行に従って徐々に作られていくことが大変多いということである．中でも，入出力データに対応する文字列は，ほとんどの場合そのようなでき方をする．そこで，徐々に確定する文字列をべたづめに表現する方式の設計が，重要な課題となる．

今までにも，文字列型，ないしは文字列の概念をもっている論理型言語はあった．中でも，［横田82a,横田82b ］と［上田83，中島83］で提案されているものは，

① 不定部分を持つことができ，しかも
② 確定部分はべたづめにできる

という特徴をもつ．しかし，これらの言語においては，最初から確定している（部分）文字列はべたづめにできても，徐々に確定する文字列をべたづめに表現することができないという問題がある．我々の方法は，第3章で述べる表現法によって，この問題を解決している．

なお，文字列型が効率よくサポートできれば，テキスト処理や事務処理も，論理型言語の重要な応用分野となりうる．

1．3．文字列型導入の対象言語の選択

文字列型を導入する対象言語として，我々はConcurrent Prolog［Shapiro 83］を選んだ．これは，入出力を含むプログラムを，宣言的解釈ができるように記述するには，Concurrent Prolog のような並列実行型言語が最も適しているからである．Concurrent Prolog の枠組みを用いると，

① プログラムを入力データ列を出力データ列との関係を表わす述語として記述し，
② そのプログラムと，入力データ列を生成する述語，および出力データ列を処理する述語の3者を，入出力データ列を共有変数とする並行プロセスとして走らせ，
③ それらの共有変数の確定状況によって各プロセスの進行を制御する

ことが容易にできる．［上田83，中島83］の試みは，順次実行型のPrologに，文字列型，入出力変数，入出力プロセスの考え方を導入しようというものであるが，それはConcurrent Prolog やPARLOG［Clark 83］がもっているプロセスやストリームの概念を，順次型のPrologの入出力部分に適用したものとみなすことができる．

2．文字列操作機能実現の指針

2．1．設計・評価基準

まず，我々が，前章までの議論に基づいて定めた，文字列操作機能に対する設計・評価基準を示そう．

a. 単純な，つまりバックトラックなしで解が見つけられるような操作が，時間的にも空間的にも効率よく実行できること．特に，手続き型言語上で開発されている効率のよい算法が，論理型言語でも原理的に同じ効率で実行できることを目標とする．
b. すべての機能が，真の論理型言語――つまり宣言的解釈の可能な言語――の枠組みにおさまっているか，あるいはそれと明確な対応関係をもっていること．

ただし，a.に関しては，手続き型言語での算法が，破壊的代入のような，手続型言語特有の機能によっているならば，簡単には論理型言語で同じ効率を得ることはできない．これはやむを得ないことである．

2．2．検討すべき機能

第1章で，我々はすでにConcurrent Prolog 上に文字列型を導入する，という基本方針を設定した．次に決めなければならないのは以下の3点である．

① 文字列型の内部表現
② 文字列型に対する基本操作（述語）の仕様と，その実行メカニズム
③ プログラム言語上での表記法，特に文字列パターン記法の導入法．

①と②の必要性は明らかであろう．③を考えるのは，次のような理由による．

論理型言語のひとつの利点は，データをパターンとして直截的に表現できることである．この特徴を，文字列操作に対しても生かす方法を考えることは，意義のあることである．

このパターン記法の導入にあたっては，以下のことを設計・評価基準とすることとした．

(i) パターン構成のために導入する機構が，できるだけ汎用かつ強力なものであること．特に，利用者がその言語内でパターンを定義できること

(ii) パターンマッチングに対する手続き的解釈ができること

2．3 String Unificationとその問題点

論理型言語に，文字列操作を最も理論的にすっきりと導入する方法は，連結演算子によって文字列を構成し，それに対してstring unificationを適用する方法であるかも知れない．String unification[Siekmann 82] とは，文字と，文字列変数と，連結演算子とからつくられる2本の文字列を，連結演算子の結合律を考慮してユニファイする操作である．例をあげよう．以下では，連結演算子を：（コロン），文字列変数を英大文字で表わすことにする．また文字は，引用符で囲んで表わす．

① "a":("b":"c") と X:"c" は，X を "a":"b" とすることでユニファイできる．

② "a":X と X:"a" は，X を

"a"，または"a":"a"，または"a":("a":"a"),...

とすることでユニファイ可能である（さらに，空文字列（連結演算の単位元）を許すならば，それも解となる）．

しかし，このstring unificationを導入することは見合せた．それは，string unificationは通常のユニフィケーションよりはるかに複雑で，効率の良いアルゴリズムが発見されていないからである．

たしかに，string unificationがあれば，その強力さ故，プログラムの記述は簡潔になる．たとえば，文字列の先頭から識別子（英字から始まり，そのあとに英数字がつづく文字列）を切り出すPrologのプログラムを書いてみよう．述語の呼び出し形を

```
id(X, (Y, Z))  (X:  もとの文字列
                Y:  切り出した識別子
                Z:  XからYを切り出した残り)
```

とする．

```
id(X:C:Y, (X, C:Y)) :- identifier(X),
          length(C, 1), not(alphanum(C)).
identifier(A) :- alpha(A).
identifier(X:B) :- identifier(X), alphanum(B).
```

しかし，このプログラムを

```
:- id("GOOD LUCK", (X, Y)).
```

（以下，不定部分を含まない文字列は，構成要素の文字を並べたものを引用符で囲って示す．たとえば，"GOOD LUCK" は，"G":("O":("O":...:"K")...) の略記とする．また，"" で空文字列を表わす．）

と呼んで，

```
X = "GOOD", Y = " LUCK"
```

を得るためには，処理系がよほど知的なことをしない限り，

① 文字列"GOOD LUCK" の，3個の部分への分解と

② 分解された部分文字列の検査

を，バックトラックによって交互に繰り返さなければならない．これは，我々が通常使用する識別子切り出しの算法に比べてはるかに非能率なものである．文字列の分解作業を，各部分文字列の持つ条件をまったく考慮せずに行なっているからである．実用的なプログラムを得るためには，文字列のような低位，大量のデータ構造に対しては，straightforward に解が求まるような算法を用いるべきである（4．1に，第3章で提案する機能を用いた，効率のよい id のプログラムを示す）．複雑な操作は，Prologの項のような，より高位のデータ構造に（文字列操作機能を使って）変換したあとで行なうべきである．

2．4．採用した機能

上記の考察から，我々は，文字列操作に対して次のような機能を与えることにした．

① 文字列型の内部表現として，べたづめのブロックのリストを採用する．Concurrent Prolog のプログラム・スタイルに従うと，文字列の値は，プログラムの実行前に不定であったものが，前の方から徐々に具体化するという場合がきわめて多い．そこで，ブロック内の，べたづめに表現された確定部分の後ろには，不定部分を許すものとする．第3章で述べるように，この表現法を用いると，不定部分を具体化しても，べたづめの性質が保たれる．変数が1本の文字列に高々1個，しかも最後部にしかないので，文字列型の値のユニフィケーションは容易である．

② 文字列に対する最も基本的な操作として，

- 文字列どうしのユニフィケーション（双方向的）操作
- 文字列どうしのマッチング（単方向的）操作
- 与えられた文字列を，最初の文字（コード）と残りの文字列に分解する操作，およびそれが（文字列が空であるために）できないことを知る操作
- 与えられた文字（コード）を，文字列の最初の文字として具体化する操作

を用意する．文字列操作の際には，

- 文字列の確定を待つ述語
- 文字列の長さを知る述語
- 文字列と文字列を連結する述語
- 文字列を指定された切断位置で分解する述語

なども頻繁に使われようが，それらは，上の基本操作を用いて実現することができる．

③ パターン表記については，述語の形で定義した文字列操作を，実行可能パターン［中島82］を拡張した記法（文脈依存パターン）によって，文字列パターンとしても用いる方式をとる．

3．文字列型とその基本操作

3．1．文字列型の表現

個々の文字列の格納領域は，1個または複数個のブロック（連続した記憶領域）によって構成され，文字列は，そのブロックの頭や途中を指すポインタによって表現される．ブロックの各要素には文字コードがはいるが，そのうちの3個の値を制御用に割り当てられるものと仮定する（制御用コードをなくしたり減らしたりすることは，いわゆるエスケープシーケンスの利用や，制御用タグビットの増設により容易にできる）．制御用コードは，以下の目的で用いる．

① 文字列の終端を表わすコード（◪と表記する）．
② 「ここから先が不定部分である」ことをあらわすコード（◪と表記する）．この不定部分は，将来文字列で置き換えられる可能性があるので，◪の後ろに，このブロック内であと何文字具体化できるかを示す数（余裕字数）を記録する．
③ 「ここから先の文字列は，別のブロックにある」ことを表わすコード（■と表記する）．このコードの後ろには，そのブロックへのポインタが格納される．長い文字列は，このブロック内ポインタで複数のブロックを連結することによって表現される．

文字列表現の例を図1に示す．一般に，文字列表現は，複数の文字列によって共有されうる．たとえば，図2(a) のXは，abcde から始まる文字列を表わし，Yは，それから ab を取り去った文字列を表わす．将来不定部分が具体化する（図2(b)）と，その値はXからもYからも見えるようになる．なお，図2(a) のZは，｜不定の文字列｜を表わす．これは一般の不定変数と異なり，文字列以外のものに具体化することができない．つまりZと，たとえば整数とをユニファイしようとすると失敗する．

文字列が終端まで確定した場合，◪のうしろの領域はむだとなる．しかしこのむだ領域は，少なくとも廃品回収時には回収できる．

3．2．基本操作

以下で，基本操作の意味と実現法を述べる．これらの述語は，引数X，Yが文字列型以外の値を持っ（てい）たら失敗する．また，調べたブロックの要素が■であったら，自動的に次のブロックへのアクセスが行なわれるものとする．

a. unify(X, Y)

a-1. 意味

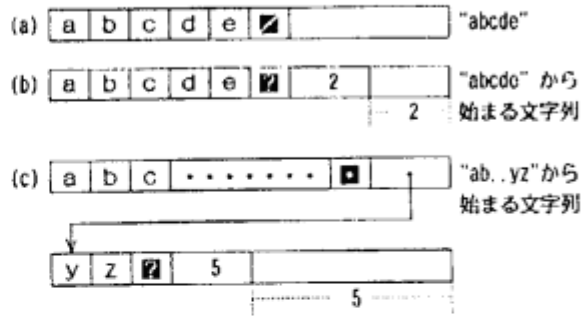

図1 文字列表現の例

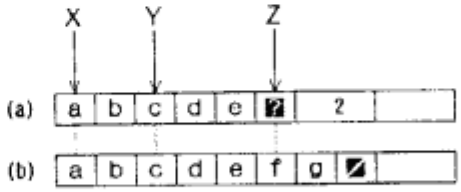

図2 文字列表現の共有

XとYを，（必要ならば，双方の不定部分を具体化することによって）ユニファイする．この操作は，述語呼び出しの際の引数のunification のときにもimplicitに呼ばれる．

a-2. 実現法

- X，Yの少なくとも一方が変数ならば，変数である方が他方を共有すればよい．
- 双方が文字列の場合は，XとYの指す要素によって，表2のような操作を行なう．

b. match(X, Y)

b-1. 意味

XとYを，Xの不定部分を具体化することなくユニファイする．

b-2. 実現法

- Xが変数のときは，それが文字列に具体化するまで待ち，それからmatch(X,Y)を試みる．
- Xが文字列，Yが変数のときは，YがXを共有する．
- 双方が文字列の場合は，XとYの指す要素によって，表2のような操作を行なう．

c. del(X, (C, Y))

c-1. 意味

文字列Xの先頭文字をC，Cを除いた残りの文字列をYとする．

c-2. 実現法

① Xの先頭1文字の確定を待つ．
② そのコード値をCとユニファイする．

表2　文字列XとYに対する操作 unify(X,Y) と match(X,Y)
（括弧内は，match のときの相違点）

Xの値 ＼ Yの値	非制御コード	◪	?
非制御コード	同じ文字ならば，XとYの2文字目以降をunify (match)，異なる文字ならば失敗	失敗	Yの?を，Xへのポインタでおきかえる
◪	失敗	成功	Yの?を，◪でおきかえる
?	Xの?を，Yへのポインタでおきかえる（Xの?の具体化を待って，match(X,Y)を再試行する）	Xの?を，◪でおきかえる（同左）	Yの?を，Xへのポインタでおきかえる

③　Xの第2文字目以降の文字列をZとするとき，
unify(Z,Y)を実行する．

d.　ins((C, X), Y)

d-1.　意味

文字Cと文字列Xをつなげたものを Y とする（Yの最初の文字をCに具体化する）．

d-2.　実現法

①　Cが（制御コードでない）文字コードに具体化するまで待つ．

②　Yが不定のとき，新たに適当な大きさのブロックを割り当てて，

という構造を作成する．この?を指すポインタをZとして，unify(X, Z) を実行し，Yにこのブロックを指させる．

③　Yが文字列のとき，

- Yの指す要素が?ならば，
  - 余裕字数が0でなければ，?をコードCで置き換え，その次の要素を?とし，それを指すポインタをZとして，unify(X, Z) を実行する．
  - 余裕字数が0ならば，?を■で置き換え，②と同様のことを行なう．ただし，作成したブロックを指すのは■につづくポインタとする．
- Yの指す要素がC以外の文字か◪ならば，失敗する．
- Yの指す要素がCならば，その次の要素を指すポインタをZとして，unify(X, Z) を実行する．

e.　empty(X)

e-1.　意味

Xが空文字列かどうかを判定する．

e-2.　実現法

Xの指す要素が，?以外の値になるまで待つ．それが◪ならば成功，それ以外ならば失敗する．

4．文字列パターン表記法の工夫

4．1．述語による表現の問題点

第3章で述べた基本操作を用いて，2．2で示した識別子切り出しプログラムをConcurrent Prolog で書き直してみよう．

```
id(X, (Y, Z)) :-  del(X, (A, X2)), alpha(A),
                  ins((A, Y2), Y)
                | id2(X2, (Y2, Z)).
id2(X, ("", "")) :- empty(X) | true.
id2(X, (Y, Z)) :- del(X, (A,  X2)), alphanum(A),
                  ins((A, Y2), Y)
                | id2(X2, (Y2, Z)).
id2(X, ("", X)) :- otherwise | true.
```

述語otherwise は，他の節が選択される可能性のある間は待機し，それらが決して選択されえないことが判明した時点で成功する組込述語である．

上の述語定義は，疑いなく論理プログラムであるが，通常のPrologの特徴のひとつである，「パターン表現を用いたデータの合成／分解操作」という特徴がほとんど失われている．つまり，扱っているデータ構造が，プログラムから読みとりにくい．また，述語による表現には，多くの補助変数に対する命名という，不愉快な作業が伴う．

4．2．実行可能パターン

Prolog/KR［中島82］における実行可能パターンは，この問題を解決するためのメカニズムのひとつと考えられる．その基本的な考えは，述語呼び出しの形式

$p(t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_n)$

から，引数のひとつ（t_k とする）を省略して穴をあけたもの

$!p(t_1, \ldots, t_{k-1}, *, t_{k+1}, \ldots, t_n)$
（ここでは穴を * で示す）

を，ひとつの項とみなそうというものである（先頭の！は，それにつづく項が実行可能パターンであることを示すためのものである）．この「項」，すなわち実行可能パターンと，項 t とのユニフィケーションは，次のように定義される．

「t と $!p(t_1, \ldots, t_{k-1}, *, t_{k+1}, \ldots, t_n)$ がユニファイ可能」
<=>
「$p(t_1, \ldots, t_{k-1}, t, t_{k+1}, \ldots, t_n)$ が成功」

例をあげよう．「整数であること」を表わす述語integer を用いて，

```
!integer(*)
```

とかくと，「整数であるようなもの」を表わす実行可能パターンとなる．これは，整数とだけユニファイできる．

なお，* のあとに変数名を書いておけば，それを含む実行可能パターンのユニフィケーションの相手の値を，その変数が受け取るものとする．この記法を用いると，

```
p(X) :- integer(X) | ...
```

というプログラムは，

```
p(!integer(*X)) :- ...
```

と書き換えられる．

さきのプログラム id を，実行可能パターンを用いて書くと

```
id(!del(*,(A, X)), (!ins((A, Y),*), Z)) :-
    alpha(A) | id2(X, (Y, Z)).
id2(!match(*,""), ("", "")).
id2(!del(*,(A, X)), (!ins((A, Y),*), Z)) :-
    alphanum(A) | id2(X, (Y, Z)).
id2(X, ("", X)) :- otherwise | true.
```

となる．

4．3．文脈依存パターン

さて，プログラム id もまた

```
!id(*,(Y, Z))
```

と，実行可能パターンとして用いることができる．が，これはまだ，「識別子を切り出す前の文字列」を表わすパターンであって，「識別子自体」を表わすパターンではない．識別子自体を表わすようにするためには，識別子の文字列を表わすＹを実行可能パターンの中に残し，Ｚを外に追い出せばよい：

```
!id(*,(Y, >))  Z
```

ここで，記号＞は，後ろに追い出した引数を表わす．

```
!id(*,(Y, >))
```

は，（パラメタＹをもった）前置演算子とみなせ，Ｚはそのオペランドとみなせる．この前置演算子は，考慮中の文脈から識別子を切り出す操作をつかさどるが，同時に，切り出した識別子を表わすパターンとみなすこともできる．そこで，この「演算子化された実行可能パターン」を，文脈依存パターンと呼ぶことにする．文脈依存パターンは，実行可能パターンの概念と演算子の概念を組み合わせたものといえる．

```
!id(*,(Y, >))  Z
```

を，たとえば文字列

```
"THIS IS A PEN."
```

とユニファイさせると，

```
Y = "THIS",   Z = " IS A PEN."
```

となる．

もうひとつ例をあげよう．

```
leftmost(S, Whole, L, R)
```

は，文字列Whole から，部分文字列Ｓの最左のものをさがしだし，その左側の文字列をＬ，右側の文字列をＲとする述語であるとする．これを文脈依存パターンとして用いるには，leftmostを中置演算子として用いればよい．

```
L  !leftmost(S, *,<,>)  R
```
（記号＜は，前に追い出した引数を表わす）

これで，!leftmost(S, *,<,>) は「最左のＳ」にマッチするパターンとみなせるようになる．左から２番目のＳをみつけるには，相手の文字列を

```
L  !leftmost(S, *,<,>)  C  !leftmost(S, *,<,>)  R
```

とユニファイさせればよい．ただし，演算子!leftmost はright-associative な演算子として定義されているものとする．

この文脈依存パターンは，単なる記法上の便宜であり，それを用いないプログラムに静的に変換できる．それには次の２段階の手続きをふめばよい．

① 追い出した引数をしまい込み，通常の実行可能パターンに変換する．

② 実行可能パターンを，補助変数を用いて通常の述語呼び出しの形に直す．

たとえば，

```
p(L  !leftmost(S, *,<,>)  C
     !leftmost(S, *,<,>)  R) :- Guard  | Body.
```

というプログラムは，まず

```
p(!leftmost(S, *, !leftmost(S , *, C, R)))
    :- Guard | Body.
```

に変換され，次に

```
p(X) :- leftmost(S, X, L, Y),
        leftmost(S, Y, C, R),
        Guard  | Body.
```

と展開される．なお，述語頭部とガード部に現れる実行可能パターンのなかには，ガード部に展開しなければならないものと，本体部に展開しなければならないものとがある（本体部に現れるものは，本体部に展開すればよい）．この区別は，実行可能パターンごとに宣言する必要がある．

SNOBOL4 [Griswold 71] にも，SPAN，BREAK 等の文脈依存パターンがあり，文脈自由なパターンとバックトラックを用いたパターンマッチングよりも効率的なパターンマッチングを可能にしている．しかし，SNOBOL4 では文脈依存パターンを利用者が定義することができない．また，文脈自由パターンの定義に際しても，そのマッチング戦略（制御情報）を指示することができない [Griswold 80]．その点，ここに提案した文脈依存パターンは，通常の述語をベースにしているから，利用者が定義可能であり，制御情報も自由に与えることができる．

```
wait(X) :- empty(X) | true.
wait(X) :- del(X, (C, Y)) | wait(Y).

length(X, 0) :- empty(X) | true.
length(X, N) :- del(X, (C, Y))
     | length(Y, N1), N := N1+1.

decomp(0, X, ("", X)).
decomp(N, X, (S, Y)) :-
     N>0, del(X, (C, X2)),
     ins((C, S2), S), N1 := N-1
     | decomp(N1, X2, (S2, Y)).

comp((S, X), X) :- empty(S) | true.
comp((S, X), Y) :-
     del(S, (C, S2)), ins((C, Y2), Y)
     | comp((S2, X), Y2).
```

Fig. 3 Some Fundamental Predicates

5．プログラム例

図3に，2．4で言及した基述語の定義例を示す．

wait(X) は，文字列Xの確定を待つ述語である．

length(X, N)は，文字列Xの長さを調べ，その値をNとする．文字列の長さは，それが確定するまでわからないから，この述語は，wait(X) の機能もあわせ持っている．

decomp(N, X, (S, Y))は，Xの先頭N文字の確定を待ち，そのN文字からなる文字列をS，残りをYとする．

comp((S, X), Y) は，文字列Sの確定を待ち，それにXを連結したものをYとする．

図4は，S式の読み込みプログラムの記述例である．

sexpr(X, (S, Y)) は，文字列Xから，LispのS式の構文に従う文字列を切り出す．Sは，切り出したS式に対応する Prolog のリスト構造になり，Yは残りの文字列になる．

図4の中の文脈依存パターンは，すべて前置演算子であり，述語頭部に現れるものの展開場所はガード部である．

このプログラムでは多くの文脈依存パターンが使われているが，その読み方は容易である．たとえば，sexprtail 中の

```
!sexpr(*,(S,>))  !skip(*,>)  !del(*,(41,>))  Y
```

は，S式を表わす文字列，空白列（最長一致），右括弧，残りの文字列の並んだものと読める．

```
%%%%% ASCII Code:  32=' ', 40='(', 41=')', 46='.' %%%%%

sexpr(!skip(*,>) !del(*,(40,>)) X,  (S, Y)) :- sexprtail(X,  (S, Y)).
sexpr(!skip(*,>) !id(*,(I,>)) Y,  (I, Y)).
sexprtail(!skip(*,>) !del(*,(41,>)) X,  ([], X)).
sexprtail(!skip(*,>) !del(*,(46,>)) X,  (S, Y))
        :- X  =  !sexpr(*,(S,>)) !skip(*,>) !del(*,(41,>)) Y.
sexprtail(!sexpr(*,(Car,>)) X,  ([Car | Cdr], Y)) :- sexprtail(X,  (Cdr, Y)).

skip(!del(*,(32,>)) X,  Y) :- skip(X,  Y).
skip(X,  X) :- otherwise | true.
```

Fig. 4 The S-expression Reader

6．まとめ

本論文では，まず論理型言語における文字列処理機能の重要性

と文字列型導入の意義を明らかにし，文字列処理機能に対する設計・評価基準を設けた．それから文字列型の表現と操作を検討・提案し，最後に文脈依存パターンの構成法を示した．

提案した方式を，2．1の評価基準にてらすと，次のことがいえる．

a. 空間効率については，徐々に具体化する文字列をブロック単位にべたづめに表現することができた．時間効率については，文脈依存パターンの導入によって，パターンマッチングによる文字列操作を，非決定的な文字列の分解操作によらずに行なうことが可能になった．たとえば，4．3で示したパターン !id(*,(Y,>)) のマッチング操作は，4．1のプログラム id に基づいて行なわれる．そして，このプログラム id は，通常の手続き的算法に対応するものである．

b. 論理型言語という観点からは，まずConcurrent Prolog をベースにすることで，副作用による入出力をプログラムから排除した．また，文脈依存パターンは，4．3で示したように，それを用いない形に静的に変換可能である．

文脈依存パターンが，さらに2．2で示したパターン導入法の基準を満たしていることは明らかであろう．導入した概念は，実行可能パターンの概念と，演算子記法の概念のみであり，しかもそれらは，文字列型を扱う述語以外にも適用可能なものである．

今後の課題のなかで最も重要なのは，バックトラックを要する処理の問題であろう．今回は，ベースとして選んだ言語Concurrent Prolog がバックトラック機能を持たないので，文字列操作機能にもバックトラックを全く含めなかった．しかし，本質的にバックトラックを要する文字列操作もある．これをどう記述するかは，ベースにする言語機能と一緒に考える必要がある．そのほかの課題としては，

- 文字列型データに対する記憶管理．
- 文字列型に対する基本操作の最適化．
- より簡潔なパターン表記法の開発．

といったものがある．

謝辞

本研究の機会を与えて下さった日本電気（株）C&Cシステム研究所の箱崎部長，山本課長，ならびに（財）新世代コンピュータ技術開発機構の渕所長に感謝いたします．また熱心な討論と有益な助言をいただいたICOT核言語設計タスクグループの諸氏，ならびにICOT招聘研究者の Dr. E. Y. Shapiro (Weizmann Institute)， Dr. K. L. Clark (Imperial College), Dr. S. Gregory (同) に謝意を表します．

参考文献

[上田83] 上田和紀，中島秀之，戸村哲：Prologにおける作用的入出力と文字列処理，Proc. Logic Programming Conference '83, ICOT(1983).

[中島83] 中島秀之，上田和紀，戸村哲：述語論理型言語における副作用のよらない入出力と文字列操作，情報処理学会論文誌，Vol.24, No.6, pp.745-753 (1983) （[上田83] の改訂版）．

[小長谷83] 小長谷明彦，梅村 恭：ShapeUp の文字列照合アルゴリズムについて，情報処理学会記号処理研究会資料24-7 (1983).

[白濱79] 白濱律雄，前野年紀：字句解析部の高速化について——ソフトウェアの調整技法——，第20回プログラミング・シンポジウム報告集，情報処理学会プログラミング・シンポジウム委員会，pp.8-18 (1979).

[中島82] 中島秀之：Prolog/KR User's Manual, METR82-4,東京大学工学部計数工学科(1982).

[横田82a] 横田 実，梅村 恭：PROLOGの記号処理への機能拡張，情報処理学会記号処理研究会資料19-2(1982).

[横田82b] 横田 実，梅村 恭：拡張PROLOG (ShapeUp)の実現について，情報処理学会記号処理研究会資料20-3 (1982).

[Clark83] Clark, K.L., Gregory, S.: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology (1983).

[Griswold71] Griswold, R.E., Porge, J.T., Polonsky, I.P.: The SNOBOL4 Programming Language, 2nd ed., p.256, Prentice-Hall, Englewood Cliffs, N.J.(1971).

[Griswold80] Griswold, R.E. and Hanson, D.R.: An Alternative to the Use of Patterns in String Processing, ACM Trans. Prog. Lang. Syst., Vol.2, No.2, pp.153-172 (1980).

[Shapiro83] Shapiro, E.Y.: A Subset of Concurrent Prolog and Its Interpreter, Tech. Report TR-003, Institute for New Generation Computer Technology (1983).

[Siekmann82] Siekmann, J., Szabo, P.: Universal Unification and a Classification of Equational Theories, in Loveland, P.W.(ed.),6th Conf. on Automated Deduction, Lecture Notes in Computer Science 138, Springer-Verlag, pp.369-389(1982).