述語論理型言語における副作用によらない入出力と文字列操作†

中島秀之††* 上田和紀††** 戸村　哲††***

Prolog に代表される述語論理型言語は，プログラムを，その仕様に近い形で記述できることを大きな特徴とする．しかし入出力に関しては，命令型言語と同様，副作用を通じて行うことが多かった．本論文では，述語ではなく変数を通じた，副作用によらない Prolog の入出力を論じる．また従来の多くの言語の入出力は，データの転送と，内外部表現間の変換を，まとまった機能として提供していた．本論文では，入出力はたんなる文字列の転送ととらえ，変換操作は，言語に文字列操作機能を用意し，それを用いて記述するようにした．これらの工夫により，入出力の概念が単純でわかりやすいものになり，しかもその扱いが柔軟になったと考える．順アクセス媒体との入出力は，たんなる文字列変数を通じて行えばよいが，データ構造の工夫により，窓構造をもったディスプレイへの出力も扱える．文字列操作は，Prolog のパターンマッチング機能を用いて簡潔に記述できる．実行可能パターンの考え方をとりいれてパターンと述語とを統一的に扱うので，文字列に導入した基本操作が連結のみであるにもかかわらず，パターンの記述能力は強力である．提案する機能は Prolog/KR 上に作成中であるが，さらに効率のよい作成技法にもふれる．残された課題には，複雑なパターンマッチングにおけるバックトラックの制御などがある．

1. はじめに

本論文では，Prolog[17] に代表される述語論理型言語[10]における，副作用によらない入出力と文字列操作について提案する．

この章では，述語論理型言語の従来の入出力方式の問題点を論じ，本論文で提案する方式の利点をあげる．2章では，従来の Prolog に新たに導入する機能のうち，入出力媒体の扱いを中心に述べ，3章で文字列処理について述べる．4章で，文字列処理を効率よく実現する方法を提案し，5章で新機能を用いたプログラム例を示す．

1.1 述語論理型言語における入出力のとらえかた

述語論理型言語においては，プログラムとは述語定義の組のことであり，個々の述語は，その引数，または引数間の関係が満たすべき条件を規定する．また，プログラムの実行とは，プログラム中の述語を呼び出し，それを満たすような引数の値の組の例を具体的に求めることである．この際，一部の引数の値のみを指定して呼び出すと，それに従って他の引数の値が決定される．またすべての引数の値を完全に指定して呼び出すと，それらがその述語を満たすかどうかの検査が行われる．

上述の基本的枠組みは本来，引数の値の授受が主記憶装置内で行われるか，あるいは外部媒体を用いて行われるかということとは無関係に成立するものである．つまり，ファイルや端末などの媒体上のデータを操作する場合でも，プログラムは，入出力データ列の間の関係として記述できる．しかしながら，従来の述語論理型言語では，外部との入出力を，述語の実行に伴う副作用として記述してきた．この副作用のために

（1） 述語論理型プログラミングの美しさが損われるばかりでなく，論理的な仕様と実際のプログラムとの間にギャップが生じ，検証が困難になる，

（2） 述語の手続き的な実行順序が本質的な意味をもつようになり，最適化や並列化の可能性が狭まる，

という問題が生じる．

述語論理型言語の入出力に関するもう一つの問題は，データ転送と内外部表現間の変換という二つの概念が分かれていないことである．

外部媒体上のデータの表現は，FORTRAN の書式なし入出力のように，主記憶装置内での表現に対応した形をとることもあるが，最終的には人間とのインタフェースをとるために，目に見える形，つまり文字列表現をとらなければならない．したがって，内部表現

† Applicative Input-Output and String Manipulation Facilities in Logic Programming Languages by HIDEYUKI NAKASHIMA, KAZUNORI UEDA and SATORU TOMURA (Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科
* 現在　電子技術総合研究所電子計算機部人間機械システム研究室
** 現在　日本電気(株) C&C システム研究所コンピュータシステム研究部
*** 現在　電子技術総合研究所ソフトウェア部言語処理研究室

と文字列表現との間の変換をどこかで行う必要があるが，この機能は入出力と切り離して提供すべきである．現存のプログラム言語の多くは，文字列型をもっていながら，この変換を単独で定義していないため，データを文字列として入力してしまうと，他の型に変換するのが困難になる．

入出力から分離した変換手続きは，文字列操作と変換先のデータ型の操作の両方からなる．このような手続きを，一つの言語の枠内で，自然な形で定義するためには，文字列操作が他のデータ型の操作と同じ形で記述できるのが望ましい．

副作用による入出力を用いていた従来は，外部の文字列に対する操作は，1文字単位の入出力をする述語をもとに，手続き的に表現するのが通例であった．しかし副作用によらない入出力を採用するのならば，文字列の連結，分解等の操作も，プログラムの手続き的解釈に依存しない，静的な型で表現すべきである．

入出力をデータ列間の関係として記述し，さらに文字列操作機能を与えると，次の利点が生まれる．

（1） あるプログラムの出力を他のプログラムの入力とすること（Unix[15]のパイプ風の機能）が，容易にできる．たとえばプログラム清書系の出力を，一時ファイルに書き出すことなく，そのまま stingy printer (5.3節) の入力に与え，二段組の出力を得ることができる．

（2） プログラムのモジュール化が図れる．既存のプログラムを少し変更して用いたいとき（たとえば余分な出力情報を捨てたいとき）でも，もとのプログラムには手を入れる必要はなく，変更のためのプログラムを外付けすればよい．

（3） 入力データの取扱いがより自由になる．たとえば，S式の入力と処理の繰返しが，空行の入力をもって終了するような会話システムを考えよう．この場合，同じ入力行に対して，空行かどうかの判断（文字列としての処理）と，そうでないときS式として読みなおすこと（S式としての処理）をしなければならないが，本方式では問題なく行える．

また，とくに述語論理型言語をベースにすることの利点として，次のものが挙げられる．

（1） 多入力多出力のプログラムが自然な形で記述でき，しかも個々の入出力をパイプ風に取り扱うことができる．

（2） パターンマッチング(unification) 機能が，文字列処理の簡潔な記述に適する．

1.2 従来の研究

副作用によらない入出力については，関数型言語では，遅延評価法[8]を応用した Friedman らの研究[4]がある．本論文で述べる入出力方式のうちの入力部分も，遅延評価法の応用とみなすことができる．

述語論理型言語に関する研究のなかでは，複数の述語の共進実行 (coroutining) 法[1),2),14),16)]が，副作用によらない入出力と深く関連している．なぜなら，本論文の方式では，利用者プログラムと，処理系のなかの入力ルーチン，出力ルーチンの3者を，変数によって結ばれたコルーチンとみなすことができるからである．この，共進実行法の入出力への応用は，次の意味で重要である．

（1） プログラム・スタイルの改善の対象が，あらゆる (trivial でない) プログラムに及ぶ．

（2） 入出力ルーチンの共進実行は実現が容易である．順次実行の Prolog 処理系があれば，そのデータ構造と unifier のまわりを少し変更するだけですむ．

文字列処理については，非命令型言語では，Poplar[11] (作用型)，Shape Up[18] (述語論理型) などが提案されている．それらと比較して，本論文で提案する方法は，パターンと述語を統一的に扱っていることが特徴である．この方法には，

（1） 言語が定義する基本操作の数が少なくてすむ，

（2） パターンの記述能力が強力で拡張性に富む，

という利点がある．

強力な文字列パターンマッチング機能を有する言語としては，ほかに SNOBOL 4[7]がある．SNOBOL 4 のパターンマッチングは確かに強力だが，

（1） パターン中に記述する変数にスコープがないので，パターン照合は容易にできても，どのように照合したかの情報（パージング情報）を得るのは容易でない，

（2） パターン照合手続きが，文の制御構造とまったく分離しており，全体として二軸的な言語になっている．しかも文の制御構造が貧弱で，プログラムの構造化が困難である，

という問題が指摘されている[5),6)]．本論文で提案する言語はこれらの問題に対する一つの解決案である．

2. Prologにおける文字列と入出力

2.1 基本概念

まず，Prologで扱うデータ構造として，新たに文字列を導入する．一般の変数を＊で始めるのに対し，1文字を表す変数は ¢ (characterのcを表す)，任意長（0文字以上）の文字列を表す変数は $ (stringのsを表す) で始めることにする．文字列リテラルは，二重引用符で囲まれた文字の列として表す．

入出力は，個々の入出力装置（ファイルや端末）と変数とを対応させることによって行う．この対応づけは，システムの用意する述語で行い，対応づけられた変数を入出力変数と呼ぶ．入力の場合は，入力装置からの文字列が入力変数と結合され，出力の場合は，出力変数と結合した文字列が外部に書き出される．

かりに，S式をその文字列表現に変換する述語をprint-imageと名づけよう．

print-image ([a, b], $x)

を実行すると，$xは

"[a, b]"

という文字列になる*．

この$xは，さらに他の述語の引数として用いることができるが，もしその述語が

=($x, $typewriter)

("="は，二つの引数同士をunifyする述語，$typewriterは，タイプライタに対応づけられた出力変数とする）であれば，$xと結合した値が，そのままタイプライタに出力される．

この出力は$xの値がすべてきまるまで待つこともできるが，一般には値がきまった部分から逐次出力するのが望ましい．

Prologの特徴の一つに，変数の値を部分的にきめる機能がある．上記の例では，S式の構造を解析する過程で，$xの値が，

"[" (アトムでないことがわかったとき)

"[a" (carがaであるとわかったとき)

"[a," (cdrがアトムでないとわかったとき)

"[a, b" (cadrがbであるとわかったとき)

"[a, b]" (cdrがnilであるとわかったとき)

と，順にきまってゆく．もし述語"="がprint-imageより前に実行されていれば，$xの値が具体化するたびに，新たにきまった部分を端末に出力することができる（この実現には複数の述語の共進実行法のときと同様，変数にデーモン（値の参照・結合時に起動される手続き）を埋めこんでおけばよい)．

全体の値が確定するまえに出力を開始する方式には，次の利点がある．

(1) 速度の遅い媒体への出力を早く完了させられる．

(2) 実時間処理，とくに入出力が同期をとりながら進行する処理が可能になる．

(3) すでに出力した情報は，プログラムの他の部分で参照していない限り，保持する必要がなくなるので，記憶域の節約になる．

ただ，順編成ファイルやタイプライタのような媒体は，いったん出力したデータの取消しが困難である．したがって，これらの媒体に対して上の方式を適用するときは，バックトラックを制限し，いったんきまった値の結合があとで解除されることを防がなければならない．

端末からの入力に関しても，出力の場合と同様の工夫ができる．最終的には，入力変数の値は端末から1回以上にわたって入力した文字列を連結したものになるのであるが，入力が完了するまでプログラムの実行の開始を待つ必要はない．Lispのlazy evaluator[8]が引数を評価する前に本体の実行を開始するのと同様，入力変数が未定のままでプログラムの実行を開始してしまう．そして入力変数を他の（たんなる変数でない）パターンとunifyしようとする際に，そのunificationの可否をきめるための必要最小限の部分だけを端末から読み込む．ここで，入力変数とunifyできるとは，

(1) 通常の意味でunifyできること（つまり，unification時の入力変数の値とunifyできること)，

(2) そのunificationによって，入力変数の未定部分の今後取りうる値が，まったく制限されないこと（つまり，入力変数の未定部分は，たんなる変数としかunifyしないこと)，

の両条件を満たすことと定義する．

たとえば，テキスト編集のための述語editを考えよう（**図1**)．editは，入力コマンド列（コマンドは行に1個ずつとする)，入力テキスト，コマンドを実行した結果の出力テキスト，端末への表示の四つの引

＊ S式を紙面で表現するためには，文字列表現を用いざるをえないので，ここではプリントされる前のS式は[a,b]，文字列表現のほうは"[a,b]"として区別する．なお，本論文ではリストの表記に"[","|","]"を用いる．これらの記号はLispにおける"(",".",")"にそれぞれ相当する．

```
edit ("", $in, $in, "").
edit ($next : "\CR" : $rest, $in, $out, $tty : $ttyrest)
    :- execute ($next, $in, $intermediate, $tty),
       edit ($rest, $intermediate, $out, $ttyrest).
```

図 1 述語 edit
Fig. 1 The predicate edit.

数をとる．この述語を実行する際，端末への出力を見ずに，あらかじめ全コマンドを入力してしまうことは明らかに不可能である．また，プログラムのほうでも，一つのコマンドの実行が終了するまで，次のコマンドは不要である．execute によって一つのコマンドの実行が終了し，edit の再帰呼出しが起き，二つある述語頭部のいずれに unify すべきかをきめるべき時点で初めて，$next に unify すべき次のコマンドの入力が起きればよい．これを入力すると，$rest が未定のまま，そのコマンドの実行に移る．

2.2 変数と入出力装置の結合

入出力装置として，ここでは(i)順編成ファイル，(ii)キーボード，(iii)タイプライタ，(iv)ディスプレイを考える．

(i)～(iii)，および順アクセス媒体としての(iv)の取扱いは容易である．これらの媒体上のデータは1本の文字列とみなせるから，文字列変数をそのまま対応させればよい．行やページの切れ目は改行文字，改頁文字で表す．もし行やページなどの上位構造を意識した処理を行いたければ，入力と出力の部分に，1本の文字列を行（＝文字列）のリスト（＝ページ）のリストに対応づける述語をはさめばよい．

実存のファイルシステムはしばしば，行の長さに制約を設けているが，出力時に，この制約を満たすように行を分割する作業も，文字列処理として記述できる．

これに対し，ディスプレイ(iv)を，固定の大きさの，再表示可能な，ランダムアクセス媒体とみなすときは，やや異なった扱いが必要である．

まず，縦横の大きさが固定の画面に，テキストの一部を整形表示させることに関しては，行長に制約のあるファイルへの出力の場合と同様の方法を取ればよい．対象が文書であれば，文書清書系の技法が利用できるし，プログラムの場合は，prettyprinter が使える．

再表示については特殊な機構が必要である．再表示とは，プログラムの実行過程で，1枚の物理的画面に表示されるデータを次々とかえてゆくことである．いいかえると，プログラムの実行中，1枚の画面に複数個の表示データが結合されるということである．そこで，画面には，毎回の（1画面分の）表示データを要素とするリストを対応させる．最初はこのリストの値は不定であり，実行が進むにつれて，通常は最初の方から要素の値が定まってゆく．このとき，端末にはリスト中の最初の不定部分の直前の要素を表示するのである．たとえばリストが

[表示データ1，表示データ2 | ＊rest]

であれば，＊rest が不定である間は表示データ2を表示し，その後もし ＊rest が

[表示データ3 | ＊rest']

と unify されたら，表示データ2を消去して表示データ3を表示する．

ランダムアクセスについては，画面上に，いくつかの固定された窓 (window) があるモデルを考える．個々の窓の中は順アクセスしか許さないが，窓同士の間には順序関係はなく，独立に表示，再表示が行える．窓単位のランダムアクセス機能である．窓の大きさは，画面全体に順アクセスしかしなければその画面の大きさとなるし，文字単位に，まったくランダムにアクセスするならば，1文字分の大きさになる．

画面は，窓の組として定義する．1画面分の表示データは，各窓の

(1) 位置，大きさ

(2) その窓への表示データのリスト

からなる（画面に対応するのは，そのような表示データのリストである）．窓の位置は固定されているが，画面全体を再表示する際にはその構成を変更することもできる．小さな窓を動かすために全体を再表示することを避けるには，窓に階層構造をもたせればよい．つまり最下位の窓以外は，下位の窓の組として定義する．こうすれば，あるレベルの窓の構成を変更するのに，すぐ上位の窓の中を再表示するだけですむ．

3. 文字列の操作

文字列操作にはパターンマッチングを用いる．文字列を，文字のリストとみなして，従来の Prolog にあるリストに対するパターンマッチングを用いることもできるが，それでは1文字と文字列の結合，分離しかできず，機能が貧弱である．文字列操作のためには，もう少し高級な基本機能を用意することが，プログラムの記述の点からも実行効率の点からも望ましい．

そこで，文字列に対する基本操作として，連結 (concatenation) を用意し，":" で表すことにする．

述語の引数として現れうる文字列項 (string term) は，文字列リテラルか，文字（列）変数か，それらを“:”で連結したものである．

たとえば，項

“a”:$x:“e”

は，$x の値が不定であれば，「“a”で始まり“e”で終わる文字列」にマッチするパターンであり，$x が（この項または他の $x を含む項の unification によって）“cut”に unify されれば，“acute”という文字列とみなすことができる．

文字列から文字列を構成する手段としては，連結だけあれば十分であるが，一般のパターンを構成する手段としては，連結のほかに選択 (alternation) が簡潔に表現できると便利である．これは，次のような考え方で導入する．

パターンでなく述語のレベルでは，引数が「“ab”か“cd”か“ef”にマッチする」ということは

p($x):- member($x,[“ab”,“cd”,“ef”]).

と書けるが，この p は「“ab”,“cd”,“ef”のいずれかにマッチするもの」と解釈することもできる．そこでこの p をパターン表現の中に取り込むことを考える．さらに一般化すると，述語 member を，第2引数に関して部分パラメタ化 (partial parameterization) したものは p と等価だから，これをパターン内に直接記述してしまってもよい．この実現には，Prolog/KR[12]における実行可能パターンがそのまま利用できる．

実行可能パターン[13]とは，！で始まるパターンで，その部分はパターンマッチングの際に述語として実行される．たとえば，“ab”,“cd”,“ef”のいずれかにマッチするパターンは，

！p($)

または

！member($,[“ab”,“cd”,“ef”])

と書けばよい．実行可能パターン内の名前のない変数（*, ¢, $ 等）がそのマッチングの相手を示す（これらの変数の有効範囲は各実行可能パターンに局所的であり，外から参照できない）．パターン

！member($,[“a”,“b”,“c”])

と，たとえば

“b”

とのマッチングは以下の手順で行われる．

1. $ が“b”にマッチする．
2. member(“b”,[“a”,“b”,“c”]) が実行され，成功する．

実行可能パターンにマッチした値を，そのパターンを含む節で参照したい場合には

！x. member($x,…)

のようにする．この場合には，$x の値を外から参照できる．

パターンの構成には，さらに任意の文字（列）の繰返しも表現できると便利である．これには

rep($x,“”).

rep($x,$x:$y):- rep($x,$y).

と定義された rep を利用すればよい．これを用いたパターン

！rep(“ab”,$)

は，

“”,“ab”,“abab”,“ababab”,…

とマッチする．このほかのパターンも，必要に応じて同じように定義できる．

実行可能パターンを用いると，たとえば，

“a”で始まり，“bc”または“de”のいずれかを何回か繰り返した後，再び“a”で終わる文字列は次のように表現できる．

“a”:！rep(！member($,[“bc”,“de”]),$):“a”

文字列のパターンマッチングは，リストの場合とちがって一意的とは限らない．たとえば，文字列“abc”と $x:$y とのマッチングには，4通りのしかたがある．

（1） $x=“”, $y=“abc”
（2） $x=“a”, $y=“bc”
（3） $x=“ab”, $y=“c”
（4） $x=“abc”, $y=“”

このため，述語呼出しを成功させるには，原理的にはバックトラックによって，適当なマッチングを探さなければならない．しかし，パターンマッチングのやり直しを無制限に許すことは，プログラムの効率上からも，読みやすさの点からも望ましくない．そこで，述語頭部におけるマッチング (unification) が一度成功して，本体の実行に移ったら，頭部のマッチングのやり直しは行わないこととする．さきの例では，$x や $y の値がまったく未定である限り，（1）のマッチング（左からの最短マッチング）がとられ，以降他の方法は試みられない．そうすると，たとえば第1引数の先頭の3文字を第2引数とする述語は

triplet($x:$y,$x):- length($x,3).

とは書けないことになる．最初の解 $x=“”が失敗しても別解が作られないからである．この解決には，2

通りの方法がある．

（1） 実行可能パターンを用いる方法

```
triplet(! x. length($ x, 3): $ y, $ x).
```

（2） 分割を別の述語にする方法

```
triplet($ s, $ x) :- characters($ s, 3, $ x).
characters($ s, 0, "").
characters(¢ c : $ s, *n, ¢ c : $ c)
    :- characters($ s, ! sub 1(*n, *), $ c).
```

このように，バックトラックが必要なときには（1）のように，そのパターン内で局所的にバックトラックを処理してしまうか（この方法が使えるなら，（2）の方式よりはるかに簡潔に記述できる），（2）のようにバックトラックを必要としないように書き換えてしまえばよいので，当該パターン外の述語の失敗によるパターンマッチングのやり直し機能は，なくても問題はないであろう．

ここで注意しなければならないのは，述語頭部のマッチング時にはバックトラックは起こりうるということである．とくに2引数以上の述語では，個々の引数のマッチングの整合性をとるために，それらの間でのバックトラックは必要である．

マッチングに必要な拘束条件，

```
length ($ x, 3)
```

を，頭部の外に（たとえばガードとして，頭部と本体の間に），書く方法も考えられる．しかし，これはあくまでパターンマッチングの時点で起動されるものであって，本体にこの条件を書くのとは本質的に異なる．

4. 文字列の実現方法

述語論理型言語の場合は，SNOBOL のように変数値の書き換えによってではなく，新しい値を次々に生成することで作業を行う．そして生成された値は，一般には結合相手の変数が有効である間保存しなければならない．したがって，とくに空間的な効率は，言語の実用化にとって重要な問題である．

文字列の表現に文字のリストを用いると，操作は単純になるが，効率は時間的にも空間的にも悪い．効率を高めるには，不定部分を含まない文字列（文字列リテラルや，unification 時に完全に値の定まる文字列変数の値．以後，定文字列という）に対してはべたづめ方式を採用し，一般の文字列は定文字列と文字列変数を葉にもつ木として表現するのがよさそうである．

たとえば，

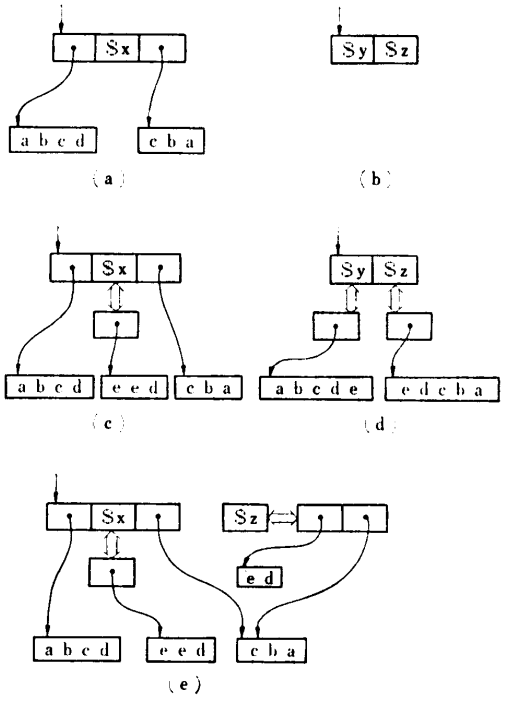


図 2 木を用いた文字列表現
Fig. 2 Representation of strings using trees.

```
"abcd" : $ x : "cba"
```

という文字列は図2（a）のように表現する．また

```
$ y : $ z
```

は図2（b）のようになる．このふたつの文字列の unification を

```
$ x="eed", $ y="abcde"
```

という束縛が起きた後に行うと，

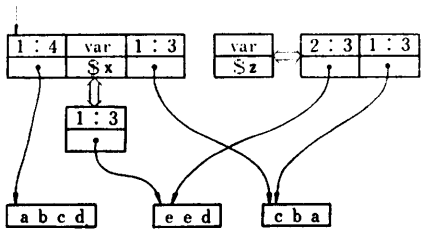
```
$ z="edcba"
```

となる．この状態での内部表現は，図2（c），（d）のようになるわけだが，これからも明らかなように，べたづめ表現をとる限り，文字列の共有は不可能であり，複写した値を変数に束縛する方式をとらねばならない．

しかし，文字列が長い場合，その分割や，他の文字列との連結操作のたびに複写が起きるのでは不経済である．長い文字列に対しては，その部分文字列を他と共有できる構造を導入すべきである．

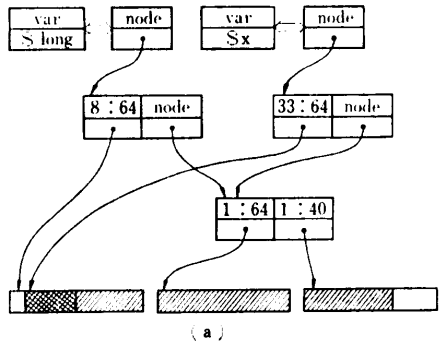
そこで，定文字列を必ずべたづめにするという条件をはずして上の $ z の表現を考えてみる．すると，図2（e）のように，まず "cba" の部分が共有できる．さらに，べたづめにされた文字列の一部を他と共有す

ることを許せば，**図3**のように“ed”の部分も共有できる．

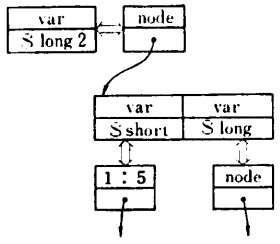


var 一変数を表す
$\alpha:\beta$—文字列の始点と終点を表す

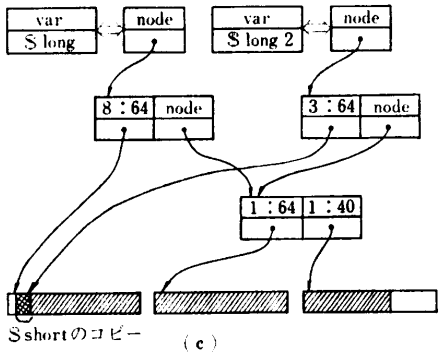
図 3 部分文字列の共有
Fig. 3 Sharing of substrings.



（a）



（b）



（c）

var 一変数を表す
node—節へのポインタを表す
$\alpha:\beta$—文字列の始点と終点を表す

図 4 長い文字列の表現
Fig. 4 Representation of long strings.

図3の方式は，**$ long** を長い文字列，**$ short** を短い文字列とするとき，**$ long** と

$ short : $ x

とを unify した際の **$ x** の表現法に応用できる（**図4**（a））．図4において node というタグは，直下の箱の内容が，文字列を表す木構造の非末端節へのポインタであることを示す）．

一方，

$ short : $ long

を **$ long 2** と unify する場合には，図4（b）のような表現をとってもよいが，もし **$ long** の表現が，図4（a）のように，その先頭文字の前部に空き領域をもっていれば，その空き領域を利用して **$ long 2** の表現を作ることができる（図4（c））．図4（b）の方式だと，短い文字列を，長い文字列の前に次々と連結してゆくと（後ろに連結する場合でも，問題は本質的に同じである），実質的にリスト表現と変わらない空間効率になってしまう．が，図4（c）のように空き領域を積極的に利用する（利用できないときは，その後の連結操作のために，空き領域付きで新たな文字列領域を割り当てる）と，空間効率を高めることができる．

5. プログラム例

5.1 Squash（図5）

squash（**$ in, $ out**）は，入力文字列 **$ in** の中の文字列“＊＊”をすべて“↑”に置き換えたものを **$ out** とする．副作用による入出力を用いた場合[3),9)]とちがい，入力文字のバッファリングを利用者が行わなくてすむ点に注目されたい．

なお，この述語を squash（**$ out, $ in**）と呼び出せば，逆に，**$ in** の中の文字“↑”をすべて“＊＊”で置き換えたものが **$ out** になる．

5.2 Parindrome（図6）

parin（**$ x**）は，**$ x** が回文（parindrome）ならば成功し，そうでなければ失敗する．また，！parin（**$**）は，

```
squash ("", "").
squash ("**": $x, "↑": $y) :- squash ($x, $y)
squash (℄c: $x, ℄c: $y) :- squash ($x, $y).
```

図 5 述語 squash
Fig. 5 The predicate squash.

```
parin ("","").
parin (℄c).
parin (℄c: $x: ℄c) :- parin ($x).
```

図 6 述語 parin
Fig. 6 The predicate parin.

```
stingy ([{i=1,120|$ line[i]}|*irest],
       [[{i=1,60|$ line[i]: "  ": $ line[i+60]}]|*orest])
       :- stingy (*irest, *orest).
stingy ([ ], [ ]).
stingy (*in, *out)
       :- subtract (120, ! len (*in, *), *shortage),
          append (*in, ! repSexp (" ", *shortage, *), *full),
          stingy (*full, *out).
```

図 7 述語 stingy
Fig. 7 The predicate stingy.

回文にマッチする実行可能パターンである.

5.3 Stingy printer (図7)

stingy (*in, *out) は，行のリスト *in をうけとり，二段組出力 *out を，ページ (行のリスト) のリストとして作成する．出力の 1 ページには，入力の 120 行分が，60 行ずつ左右に分かれてはいる.

*out は，そのままではタイプライタにもディスプレイにも出力できない．出力媒体に合った出力用述語を経由して書き出す.

ここで

{名前=始値，終値|形式}

は，"形式" の中の "名前" を始値から終値までで置き換えたものを，書き並べたものと等価である．たとえば，

{i=1, 120|$ line [i]}

は，$ line [1] から $ line [120] までを書き並べたものに等しい．これは単なる記法上の便宜であり，その展開はプログラムの読込み時に行える.

なお，述語 repSexp (*x, *n, *y) は，*n 個の S 式 *x からなるリストが *y であることを表す.

5.4 端末へのデータ表示 (図8)

$ keyboard をキーボードからの入力文字列，*display をディスプレイへの 1 画面分の表示文字列のリスト (2.2 節参照) とする.

display ($ x, $ keyboard, *display)

を実行すると，$ keyboard から復帰 (return) 文字が入力されるたびに，$ x の次の 24 行が *display の次の一要素として確定し，端末に表示される.

表示を途中でやめたいときは，キーボードから，文字列の終了を示す記号を入力すればよい．逆に，復帰

```
display ({i=1,24|$ line[i]: "\CR": }$ irest,
        "\CR": $ control,
        [{i=1,24|$ line[i]: "\CR": }" "|*orest])
         :- display ($ irest, $ control, *orest).
display ($ in, " ", [ ]).
display ($ in, "\CR": $ control, [$ in]).
```

図 8 述語 display
Fig. 8 The predicate display.

```
read (" " " " ": $ atom : " " " " " ": $ rest, $ atom, $ rest).
read ("[": $ x, *s, $ rest) :- readtail ($ x, *s, $ rest).
readtail (! read ($, *s, $ x), [*s|*srest], $ rest)
        :- readtail ($ x, *srest, $ rest).
readtail ("]": $ rest, [ ], $ rest).
readtail ("|": ! read ($, *s, "]": $ rest), *s, $ rest).
```

図 9 述語 read
Fig. 9 The predicate read.

文字を $ x の表示に必要な数より多く入力した場合，余分な文字は読み捨てられる.

5.5 S 式の入力 (図9)

read ($ x, *s, $ xrest)

は，S 式の構文に従う文字列 (ただし，アトムは引用符で囲まれた文字列とし，S 式中に空白や改行文字は現れないものとする) を $ x の先頭から切り出し，文字列を要素とする S 式に変換して *s の値とする．$ xrest は，S 式を切り出したあとの，$ x の残りの部分である.

入力文字列が空白や改行文字を含むときは，それらを除去する述語を通してからこの述語を用いればよい.

6. おわりに

Prolog の入出力を，副作用によらずに，変数を通じて行う方法と，Prolog に文字列操作機能を導入する方法について提案した.

変数の値がきまるつどそれを表示する機能については，Prolog/KR のインタプリタに手を入れて実験済みである．入力の側についても同様である.

文字列パターンマッチングも，Prolog/KR 上で実験した．ただ，マッチングの手順やマッチング中のバックトラックの扱いに関して，すべての場合にうまくゆく方法は見つかっていない．少なくとも一方が，値の確定した文字列か，たんなる変数である場合は問題ないが，値が未定の変数を含むパターンや実行可能パターンどうしの unification が問題である．文字列パターンマッチングのセマンティクスと実現方法については，今後詳細に検討し，報告したい.

いくつかの例題 (5 章のプログラム，KWIC (Key Word In Context)，相互参照プログラムなど) を記述してみて，1.1 節であげた利点が確認された．とくに，仕事をデータ列の加工という観点から細分割し，その各部分を独立に記述できるのが好都合であった．モジュール性が高いので，プログラムの理解，変更が容易である．その一方，次のようなこともわかった.

(1) 対象をテキスト処理に限ったとしても，すべての処理を文字列レベルで行うのは賢明でない．単語，行などの上位構造があれば，入力部分でそれを認識した上で残りの処理をすべきである．

(2) 入力文字列の構文解析を行うとき，正しい入力を正しく解析するプログラムはパターンマッチング機能で簡潔に書けるが，誤った入力に対して適切な処置を施し，解析を続行するのは容易でない．

(1)については，単語の列などに対して文字列の場合と同様のパターンマッチング機能を与えることでうまく解決できそうである．が，(2)の問題は，誤りに関する情報を次段のプログラムに伝える手法とともに，今後の研究課題である．

参考文献

1) Clark, K. L. and Gregory, S.: A Relational Language for Parallel Programming, Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, pp. 171-178 (1981).

2) Clark, K. and McCabe, F.: IC-PROLOG—Language Features, Proc. Logic Programming Workshop, pp. 45-52 (1980).

3) Conway, M. E.: Design of a Separable Transition-Diagram Compiler, *Comm. ACM*, Vol. 6, No. 7, pp. 396-408 (1963).

4) Friedman, D. P. and Wise, D. S.: Aspects of Applicative Programming for File Systems, Proc. ACM Conf. on Language Design for Reliable Software, *Sigplan Notices*, Vol. 12, No. 3, pp. 41-55 (1977).

5) Griswold, R. E.: A History of the SNOBOL Programming Language, *Sigplan Notices*, Vol. 13, No. 8, pp. 275-308 (1978).

6) 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).

7) Griswold, R. E., Porse, J. F. and Polonsky, I. P.: The SNOBOL 4 Programming Language, 2nd ed., p. 256, Prentice-Hall, Englewood Cliffs, N. J. (1971).

8) Handerson, P. and Morris, J. Jr.: A Lazy Evaluator, Proc. 3rd ACM Symp. on Principles of Programming Languages, pp. 95-103 (1976).

9) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666-677 (1978).

10) Kowalski, R. A.: Predicatic Logic as Programming Language, Proc. IFIP Congress 1974, North-Holland, Amsterdam, pp. 569-574 (1974).

11) Morris, J. H., Schmidt, E. and Wadler, P.: Experience with an Applicative String Processing Language, Conf. Record of the 7th ACM Symp. on Principles of Programming Languages, pp. 32-46 (1980).

12) Nakashima, H.: Prolog/KR User's Manual, METR 82-4, Dept. of Mathematical Engineering, Univ. of Tokyo (1982).

13) Nakashima, H.: Prolog/KR—Language Features, Proc. 1st International Logic Programming Conference, pp. 65-70 (1982).

14) Porto, A.: Epilog: A Language for Extended Programming in Logic, Proc. 1st International Logic Programming Conference, pp. 9-18 (1982).

15) Ritchie, D. M. and Thompson, K.: The UNIX Time-Sharing System, *Comm. ACM*, Vol. 17, No. 7, pp. 365-375 (1974).

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

17) Warren, D. H. D., Pereira, M. L. and Pereira, F. C. N.: Prolog—The Language and Its Implementation Compared with Lisp, Proc. SIGPLAN/SIGART Symp. on Artificial Intelligence and Programming Language, pp. 109-115 (1977).

18) 横田 実，梅村 護：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).

(昭和57年12月23日受付)
(昭和58年4月19日採録)