ソフトウェア工学　60－1
（１９８８．６．２２）

複雑なデータの表現に向く構文データ型とこれを導入した言語ｆｉｘ

大竹　和雄*、　福岡　秀幸*、　岸川　徳幸*、　中川　雅視**
（*日本電気㈱ C&Cシステムインタフェース技術本部、**日本電気技術情報システム開発㈱）

階層化した構造を持つ情報を処理するために、構文によるデータ構造の定義と構文解析をプログラミング言語に導入する方式を考察し、この方式を実現したプログラミング言語ｆｉｘとその処理系を開発した。構文データ型を言語に導入することの利点は、情報の持つ階層構造がＢＮＦとしてプログラムに陽に現われること、ＢＮＦの名前を用いて情報にアクセスできること、構文解析器の自動生成技術を用いてプログラミングの工数を軽減できることである。構文型を導入した言語はマルチメディア文書を扱うシステム、ネットワークで情報をやり取りするシステム、オペレーティングシステム等のコマンド言語を扱うシステム、データベースシステムの開発などに有効である。

Syntax as a Data Type of Programming Languages and Syntax Oriented Programming Language Fix

Kazuo OTAKE*, Hideyuki FUKUOKA*, Noriyuki KISHIKAWA*, Masashi NAKAGAWA**
*C&C Systems Interface Engineering Laboratory, NEC Corporation,
**NEC Scientific Infromation System Development Corporation
14-22, Shibaura 4-chome, Minato-ku, Tokyo 108, Japan

A new data type for programming languages is proposed to represent nest-structured data. This new data type is based on syntax description and processing techniques. This type is called syntax data type. A programming language 'fix' is developed for evaluating this syntax data type. Programming languages that introduce the syntax data type have some advantages. Structures of data appear explicitly as BNF descriptions in a source program. Sub-structures of data can be accessed using symbols in BNF description. Programming languages with the syntax data type can use automatic parser generation techniques and users of this programming languages need not to develop input routines for nested data.

Programming languages with syntax data type is effective for developing systems processing multi-media documents, systems processing data that are exchanged through computer networks, systems processing command languages of operating systems, and data base systems.

1．　はじめに

マルチメディア文書、メールなどネットワーク上で交換されるデータ、オペレーティングシステムのコマンド、データベースの情報等、複雑でかつ厳密に定義された構造を持つ情報を処理するソフトウェアのニーズがきわめて高くなっている。それらの情報の構造は実は構文として厳密に規定できる場合が多い[1]。一方構文解析器ジェネレータの研究が進み[8]性能も向上し実用化が進んでいる。構文解析器の自動生成はもともとコンパイラの自動生成を目的として研究されてきたものだが、上で述べたようなデータの構造はプログラミング言語の構文よりもはるかにバラエティに富んでおり、この領域への構文解析器の自動生成の適用の意義は大きい。

ところで、現在利用できるツールとしての構文解析器ジェネレータはコンパイラコンパイラの形態をとっていることが多いが、目的とするところが異なるので、コンパイラ生成以外の用途に対しては最適の形態ではない。本稿では、構文によるデータ構造の定義とこれからの構文解析器の自動生成を汎用言語の提供する機能として実現することを提案する。

以下では、構文を汎用言語に組み込む場合の方式について考察し、これの一部を実現したやや特殊な問題向き言語であるｆｉｘに関して報告する。

2．　データ型としての構文

構文に対する処理で重要なものは構文解析であり、得られた解析木にたいする処理はこれに付随するものとして表現されることが多い。しかし本稿では構文をデータ型のひとつとして捉え、この型の値である解析木に対する処理と構文解析を同等なものとして扱うことを提案する。このデータ型を構文型と呼ぶことにする。

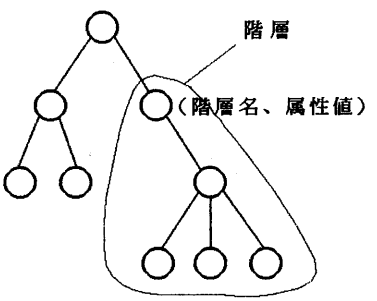


図１．階層木

定義　構文型とは次のような階層木（図１）を値として持つ型である。

階層化されていて、各階層が名前を持ち、とりうる構造が構文によって規定される。各階層名は任意の型の属性値を持つことができる。

階層木は構文解析の言葉でいう解析木に他ならない。階層の名前はＢＮＦ（Backus Naur Form）による記述に現われる文法記号に対応する。解析木では終端記号（最下位の文法記号）のみが属性値を伴うことが多いが、構文型では属性文法[7]と同様にすべての文法記号が属性値をもつことができるものとする。

文法にはさまざまなクラスがあるが本稿では文脈自由文法（あるいはそれより狭いクラス）のみを考える。構文の記述法としてはＢＮＦを用いる。

階層木をデータ型とする利点を以下にあげる。

(1)　構文型の定義は文脈自由文法（ＢＮＦ）によって行なう。ＢＮＦによるデータの記述がプログラムに陽に現われるのでソースプログラムの文書性が良くなる。また後で述べるように階層構造に沿ったデータへのアクセスがＢＮＦによる記述中の階層名によって行えるのでこの意味でもプログラムの文書性の向上が達成される。

(2)　ひとつの構文の定義をデータ型の定義として扱うので、ひとつの処理中で複数の構文を扱うことが自然に行える。

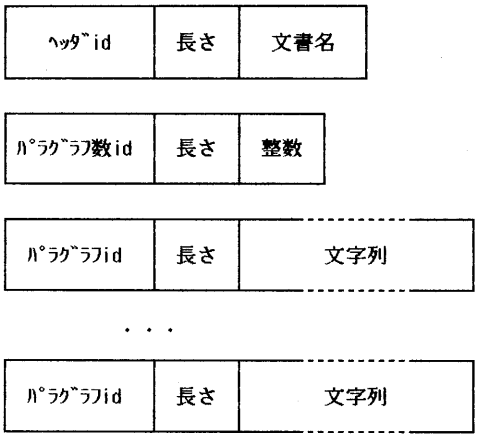
(3)　構文解析動作以外でも任意の計算の結果として階層木を作成していくことができる。一度作成した階層木の構造を変更することもできる。

(4)　構文解析動作を言語上でread操作として捉え、構文解析の実行のタイミングをコントロールする事ができる。

コンパイラコンパイラの形態ではなく、構文解析と独立して階層木を扱うことの有用性を例を用いて説明する。

フォーマットⅠ（図２）に従う文書をフォーマットⅡに変換する場合を考える。フォーマットⅠの文書はパラグラフの列であり、フォーマットⅡは例えばページの列である。入力した文書の構造の解析は構文解析によって解析されるので、コンパイラコンパイラの利用が有効である。この場合文書の変換は得られた階層木をフォーマットⅡの構文にしたがう階層木に変形する事によって達成できる。しかし、コンパイラコンパイラでは構文解析以外の方法では階層木を作成しないので、階層木をもとに新しい階層木を作成したいこの場合には利用できない。

(a) 文書フォーマットの構造



(b) ＢＮＦによる構造の記述

```
Document: HEADER NofParagraph paragraphs ;
paragraphs: PARAGRAPH paragraphs
          | PARAGRAPH
          ;
```

図２．　文書フォーマットＩの構造とＢＮＦ記述

新しい階層木を作成しないで直接フォーマットⅡの文書を生成することによる解決も考えられるが、いつでもそうできるとは限らない。例えばフォーマットⅡでは先頭に文書のページ数を付け、各ページにそのページの総字数を付加しなければならないとする。ページの字数がわかるのはページを作った後、ページ数がわかるのは文書全体を作った後である。この場合はフォーマットⅡの階層木を作成した後で木を探索しながら総ページ数、字数の情報を木に付加していく方法が自然である。

この例はプログラミング言語以外に構文型を適用する場合に典型的な特徴を持っている。この例の構文解析は難しいことではない。階層化した文書フォーマットＩはそのデータ内にその構造に関する情報を持っている（図２(a)）。構文解析のやらなければならないことはこの構造に関する情報から自明に導かれる階層木の各階層名をＢＮＦ記述の文法記号に対応して付加することだけである。

この例で重要なのは階層化したデータを処理するためにプログラミング言語のデータ構造として取り込む際に階層木という構造を用いたことである。これに伴ってＢＮＦによって階層データの構造を定義し、ＢＮＦの文法記号を用いて階層木にアクセスすることができる。

３．　構文型に対する言語要素

議論をより具体的にするために、以下では仮想的な言語Ｇを想定しその仕様を説明していく。

３．１　母胎とする言語

構文型を用いる言語に制限はない。手続き型言語、関数型言語，論理型言語、対象指向言語のどの言語においても用いることができる。後に述べる言語ｆｉｘでは処理の記述においてもＢＮＦを拡張した形式を用いる。また、強い型の言語（Pascal等）でも弱い型の言語（C等）でもそれぞれ利用できる。型付けが強い構文型の変数には構文で許される構造以外は代入できないが、弱い場合には構文からはずれる構造も代入できるという設計がたとえば考えられるであろう。

仮想言語Ｇの母胎となる言語としてはその親しみやすさからＣ等の手続き型言語を想定する。非手続き型言語に埋め込む場合は相当異なったsyntaxになるであろう。

３．２　仮想言語仕様Ｇ

(1) 定義（宣言）

構文型の定義はＢＮＦによって行なう。

```
grammarＩ{
Document: HEADER NumberofP paragraphs ;
paragraphs: PARAGRAPH paragraphs
          | PARAGRAPH
          ;
}
```

(2) 入力

構文解析をread動作として言語に組み込むことにすると任意のタイミングで構文解析を行なうことができる。

```
gv = parse(tstream, startsymbol)
```

parseの返す値を構文型のgvに代入するという使い方が一般的であろう。startsymbolは解析するデータの持っているべきパターンを示すＢＮＦのスタートシンボルである。parseの返す値は最上位の階層にstartsymbolの名前のついた構文値か解析失敗を示す値である。tstreamは終端記号の一次元列を生成するストリーム名（すなわち呼ばれる度に次の終端記号と属性値を返す字句解析ルーチン名）である。

(3) 出力

```
unparse(tstream, gv)
```

write動作は２種類考えられる。ひとつはread動作と完全に対称的に終端記号の列をstreamにおくりこむ。すなわち、構文値gvを最下層の階層名（終端記号）とそれぞれに伴う属性値のペアをtstreamに送り込む。しかしより一般的にはgvの持つ情報を最大限送り込むために階層木の全ての階層名をリスト表現に相当する記号列を送り込む。例えば以下の列をアンダーラインごとにtstreamに送り込む。

layer1(layer2(T1("a b c") T2(15)) layer3(T3('ff'x)))

(4) 構文値に対する操作（演算）

構文型の値に対する操作として特有なものとしては階層の下位の要素にアクセスする操作、複数の構文型の値を集めてひとつの階層とする操作が考えられる。また階層名の持つ属性値を取り出し、また代入する操作がある。構文型の階層は名前を持つと定めたので、階層を作る場合には名前をつけ、下位の要素にアクセスする場合には下位の階層が持つ名前によって望む対象を指定する。名前によって指定するので、ある階層の直接の下の階層の名前はすべて異なっていなければならない。これは一般のＢＮＦにはない制限である。この制限をなくすには名前でなくて位置によるアクセスを導入する必要がある。

(4-1) （上位のシンボルに第入するために）シンボルをまとめる

vs = makelayer(s, vc1, vc2, vc3)

ここでsはこの階層に付けられる名前である。階層の末尾に要素を増やす操作も有用であろう。

vs = add(vs, vc)

(vc,vc1,vc2,vc3は階層木を表わす)

(4-2) 下位の階層をアクセスする。

gv.paragraphs.PARAGRAPH

gvは構文変数（または構文値）でありその他は階層名である。gvの下のparagraphsの下のPARAGRAPHを参照する。

構文型は一種の木構造である。一般的な木構造はLISP言語やprolog言語などの主要なデータ構造であるリスト構造で表現できる。しかし、構文型は構文解析を別にしてもリスト構造とは別のものである。リストは構造的には任意の木を表現できるが、階層に名前が無い。位置（car、cdr)があるだけである。prologのデータは名前でアクセスする（ユニフィケーション）が、データを扱う述語が一時的に名前を決めるのであってデータの階層に名前が張り付いているわけではない。

構文型では各階層が固有の名前を持っている。この特徴を活かして階層を順にたどらずに一気に望む位置を指定することができれば便利である。例えば、

gv2..X

という記述で、gv2の階層内のXという階層名を参照する。この場合に問題となるのはgv2の内部の複数の階層がXという名前持っていた場合である。この問題のひとつの解を４．３章で議論している。

(4-3) 階層の持つ属性値をとりだす／代入する。

51 + T~val

T~val = "abc"

(5) 変数に対する操作

(5-1)値参照

構文値が期待される文脈で構文変数が現われた場合にはこれに束縛されている値と解釈する。

(5-2) 代入（束縛）

構文変数への値の束縛は構文解析に対応するread操作が最も重要であるが、一般的な代入も考えられる。その場合はＢＮＦで規定された構造を持つ構文値のみを代入することが許されるという実現方法も考えられる。

(6) 定数

型の完全性からみて構文型の定数も必要である。階層名と付随する属性値のペアのネストしたリストの形式で記述することが便利であろう。

構文

```
s: b c D ;
b: B ;
c: C ;
```

に対して例えば、

```
(<s>
 (<b,5> <B, "abc">)
 (<c,0> <C, "">)
 <D, "xyz">
 )
```

(7) 字句解析ルーチンの指定

read操作で字句解析ルーチンの関数名を指定するとした。これに関してもUNIXのlexのような正規表現から関数を生成する機能を言語に取り込むことが理想である。

しかしlexは入力として文字列のみを仮定している。構文型は文字列のみでなくマルチメディア、データベース、ネットワーク情報等の階層化されたデーター般に有効である。より一般的な字句解析ルーチン生成の方法が求められる。

4．　ｆｉｘ言語

ｆｉｘ[4,5,6]は構文とこれに基づく入力を受けて行なう処理の記述を組み合わせた言語である。別の言い方をすれば、コンパイラコンパイラのコード生成記述を汎用の処理記述に拡張したものとも考えられる。筆者らの所属するセクションで開発しているターゲットマシンリンク（ＴＭＬ）システム[11,12]の開発などで用いている。このＴＭＬシステムはワークステーション上のUNIX環境でメインフレーム上のソフトウェアを開発するためのシステムである。

ｆｉｘはひとつの構文の記述とこの構文を持つ階層木をひとつ入力してそれに対して行なわれる処理を記述する。複数の構文を同時に扱う処理は記述できない。これはコンパイラコンパイラを汎用に用いようとした場合にも生じる問題である。構文はＢＮＦで記述し、その処理もＢＮＦを拡張した記法で記述する。構文と処理記述は対等の関係にあり、基本的には構文解析が終了した後で処理が１度だけ行なわれる。コンパイラコンパイラは構文解析の進行に連れて処理を行なうが、４．１章④の例でわかるように、入力がすべてそろってから処理を行なうというパラダイムの方が汎用性が高い。階層木はひとつしかないので名前もなく、ただ各階層の名前だけでアクセスできる。階層のアクセスとしては階層を順にたどる方法のほかに直接指定する方法を提供している。名前による直接的なアクセス方法は人間にとって自然な記述を提供する。

コンパイラコンパイラでは解析木は入力時に決まりその後変形することはない。ｆｉｘでも原則的には同様だが、適用の経験から木の変形操作の必要性が明らかになりライブラリとして追加した。変形操作の可読性は悪い。仮想言語Ｇでは構文値はread操作によって決まるだけでなく計算の対象であるので自由に変形できる。

ｆｉｘコンパイラはｆｉｘソースからｃプログラムを生成する。構文解析の処理にyaccを用いており、ソースプログラムの構造、字句解析ルーチンのインターフェースもyaccに似たものになっている。

4．1．　ｆｉｘの動作

本節ではＴＭＬのｆｉｘによる記述の一部を簡単にしたもの用いて言語を説明していく。用いる例はFortran 77のコンパイルコマンドをUNIX上のものからメインフレームのコマンドへ変換する部分である。

コンパイルコマンドを変換するためにはUNIXとメインフレームのファイルシステムの対応規則を決めておかなくてはならない。今回は簡単に以下のように対応させる。

メインフレームとUNIXのファイルシステムの対応

メインフレーム上ではソースユニット、コンパイルユニット、ロードモジュールはそれぞれ固定されたファイルに格納されるものとする。すなわち、メインフレーム上ではソースユニットはすべてＦという名前の待機結合編成ファイルに格納されていなければならない。そしてそれらをコンパイルして得られるコンパイルユニットはソースユニットと同じ名前で待機結合編成ファイルＯに格納される。同様にロードモジュールはＯＵＴという名前の待機結合編成ファイルに格納される。

UNIXのコマンドと対応するメインフレームのコマンドへの変換の実行例を以下に示す。小文字がUNIXのコマンドで、続く大文字がメインフレームのコマンドである。

① コンパイル後リンク

```
f77 main.f sub.f lib.o

FORT77 F(MAIN,SUB) CULIB=0
LINK OUT(MAIN) INLIB=(O)
```

UNIXのf77コマンドはひとつでコンパイルとリンクまで行うのでメインフレームでは二つのコマンドに対応させる。リンクコマンドで作成するロードモジュール名としては、f77コマンドで最初に現われた名前（この例ではMAIN）を用いる。

② ロードモジュール名の指定

```
f77 -o prog.out main.f

FORT77 F(MAIN) CULIB=0
LINK OUT(PROG) INLIB=(O) ENTRY=MAIN
```

f77の-oオプションはメインフレームのLINKコマンドのロードモジュール名に反映される。この場合、ロードモ

ジュール名とメインプログラム名が一致しなくなるので、ENTRY指定を追加する。
そのほかさまざまなオプションをメインフレームのコンパイルコマンドかリンクコマンドの対応するオプションにつくり直す。

③ コンパイルのみ。

```
f77 -c sub.f aux.f

FORT77 F(SUB, AUX) CULIB=0
```

-cオプションがある場合はリンクを行なわない。

④ リンクのみ行う場合

```
f77 main.o sub.o lib.o

LINK OUT(MAIN) INLIB=(0)
```

このf77コマンドにはソースファイル（サフィックス.fのついたファイル）がひとつも現われない。コンパイルを行なわずリンクするためだけにf77コマンドを使うのも正しい使い方である。しかし、

```
FORT77 F() CULIB=0
```

などというコマンドはメインフレームではエラーになるのでFORT77コマンドの生成を抑制しなければならない。入力にソースファイルが現われるか否かはコマンドを最後までみなければわからない。従って、入力の解析が終了して初めて出力を始めることができる。

4．2 ｆｉｘによるコマンド変換の記述

図３はUNIXのf77コマンドの（やや簡略化した）フォーマット定義を記述したもので原構文と呼ぶ。コマンド名（F77）に続いてフラグとファイル名がいくつか並ぶ。このように扱うデータの構造が陽にプログラム中に現われることがひとつの利点である。これによってプログラムの文書性が上がる。

原構文は入力のフォーマットを定め入力ごとに階層木を作る。例えば次の入力に対しては図４で示すような階層木が作られる。

```
f77 -C -o prog.out main.f sub.f lib.o
```

```
%token F77 SFILE OFILE NAME
%token MINUS_c MINUS_O MINUS_C MINUS_g MINUS_U MI
NUS_o MINUS_lm
%token OTHEROPTION
%token EOS  /* carriage return */
%token FIRSTFILE
%%
commands :  commands f77 | /* empty */ ;
f77 :     F77 args EOS { af77(f77); } ;

args : args arg | arg ;
arg : flag | file { memtopfile(file); } ;
file : SFILE | OFILE ;
flag :    MINUS_c       /* "-c" */
        | MINUS_O       /* "-O" */
        | MINUS_g       /* "-g" */
        | MINUS_U       /* "-U" */
        | MINUS_C       /* "-C" */
        | MINUS_o NAME  /* "-o file" */
        | MINUS_lm      /* "-lm" */
        | OTHERO        ;
```

図３．　コマンド変換の原構文

4．3 入力の各要素へのアクセス

この階層木を参照しながら行うべき動作の記述が目的構文である。このときｆｉｘでは階層木上の階層を指定する方法を２通り提供している。ひとつはすでに何らかの方法で特定されている階層から出発して下へ順にたどっていき目的の階層にたどり着く指定法と、もうひとつは階層木の階層につけられた名前で直接指定する方法である。

直接指定法では階層木中には同一の名前のついた階層が複数あり得るので曖昧さがある。そこで極大値解釈と呼ぶ方法を採用する。同じ名前を持つ階層の中で極大な階層とは、階層木内でそれより上にたどっていって同じ名前が現れないような階層を指す。

これは次のように言い替えることができる。階層木を階層間に親子関係があるとき親のほうを大きいとする半順序集合とみなす。このとき同じ名前を持つ要素の中で極大な要素を採用する。そのような極大な階層もまだ複数あり得るが、その場合はどれが採られるかは規定しな

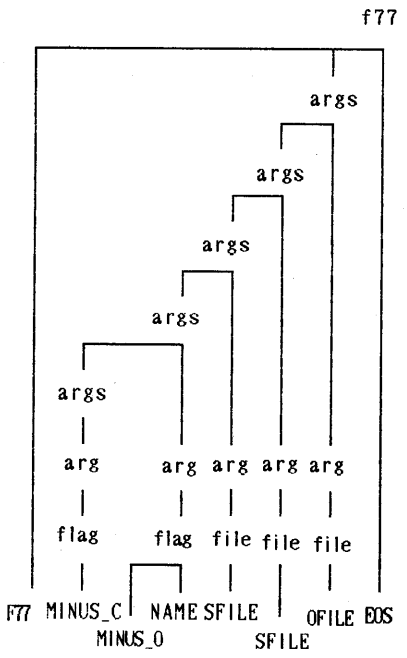


図４．階層木の例

い（非決定的）。インプリメントによって異なってかまわない。

要素の指定に曖昧性が残る点は次に挙げるような使い方をする場合は障害とはならない。

＜ケース１＞　もともと探す候補が一つの場合。

＜ケース２＞　複数あっても存在するか否かだけ分かればよい場合。

＜ケース３＞　複数の階層名が一つのリストの中に現われる場合。この場合極大値解釈はリスト全体を表わす階層を参照する。

どうしても曖昧さが問題で、かつ階層に沿ったアクセスを使いたくない場合は原構文において他と区別できる階層をひとつふやすことによって解消できる。曖昧さを嫌って、選ばれる要素を特定する複雑な規則を定めると、それに依存した記述ができるためにきわめて読みにくいプログラムが書かれるおそれがある。

続く目的構文の説明でその記述能力を見ていく。

階層木に対する処理の記述を目的構文と呼び、文法記号にパラメータの付いたＢＮＦで記述する。図５で示す目的構文のトップレベルではコンパイルコマンド、リンクコマンドそれぞれを生成するか否かをまず振り分けている。この目的のためにガードの中でトークンMINUS_cとSFILEに対して直接参照を用いている（図５の①、②）。"-c"オプションが指定されていればコンパイルコマンドのみを作成する。"-c"オプションは存在のみを問題にしているので＜ケース２＞に相当する。同時に入力中に高々ひとつなので＜ケース１＞でもある。　またSFILEの存在をたしかめて、ソースファイルが指定されている場合にのみコンパイルコマンドとリンクコマンドを作成する。これは＜ケース２＞に相当する（図５の②）。

この例で明らかなようにコマンド全体を読み込んでから初めて最初の出力を作ることができる。yaccのように構文解析と同時に処理を行ない解析木を作らない方法は一般のプログラミングのパラダイムとしては狭い。ｆｉｘでは解析を終了してから処理を行なうことを基本とし、構文解析中に処理を行なうことは例外的な機能とみなしている。

ｆｉｘでは仮想言語Ｇとは異なり、ＢＮＦ記述のスタートシンボルからの極大値探索のみが提供されている。そこで実装上は、階層木を作る段階で入力に現れる全ての名前に対して各々の極大値解釈を計算しておくことが可能である。極大値のうちのどれが選ばれるかは規定しないので実装上もっとも効率的な方法を選ぶことができる。現在開発中のシステムではボトムアップパージングを用いていて、階層を作成する度に階層と同じ名前の変数に作成した階層の位置を代入（上書き）していくという単純な方法で極大値解釈を効率的に実現している。

図６にコンパイルコマンドを作成する目的構文をしめす。この中の

```
printf("FORT77")
```

等はｃ言語のprintf文である。ｆｉｘではこのようにｃ言語の呼び出しを行えるようになっている。

```
af77(f77)
        : (MINUS_c) fort77(.args)              ①
        | (SFILE) fort77(.args) link(.args)    ②
        | link(.args)                          ③
        ;
```

図５．　コマンド変換の目的構文（トップレベル）

```
fort77(args) :
    printf("FORT77")
    sourceunit(args)
    printf(" CULIB=0")
    options(args)
    printf("¥n")
    ;
options(args)    /* for each flag do option */
      : (.args) options(.args) option(.arg)
      | option(.arg)
      ;
option(arg)
      : (.flag.MINUS_O) printf(" OPT=4")
      | (.flag.MINUS_C) printf(" CHECK")
      | (.flag.MINUS_g) printf(" TEST")
      | /* other options are ignored */
      ;
```

図６．コマンド変換の目的構文（つづき）

ＴＭＬのようなシステムではunixコマンドをメインフレームのコマンドに変形するだけでなくメインフレーム上で実行した結果得られるメッセージをまたUNIX風に加工してユーザに示す機能も必要である。実行結果を的確に加工するには実行したコマンドの情報が必要である。ｆｉｘではひとつのプログラムではひとつの構文しか扱えないので両者を合わせて無理やりひとつの構文として扱かうか、それぞれの構文の処理を別のプログラムとして実現し、両者の間で情報を受け渡す他ない。自然に記述するにはやはり複数の構文をひとつの処理の中で扱える言語が望ましい。

５．　課題

仮想言語Ｇのパージングのアルゴリズムに対してはとくに議論しなかった。効率的な構文解析器の自動生成が可能な文法として最も広いクラスはbottom up parsingであるLR文法であるが、プログラミング言語以外のデータへの応用としてはtop down parsingのLL文法で十分であるとも考えられる。Ｇ言語の実現の際には考察が必要である。

６．　まとめ

構文を一般的なプログラミングのパラダイムとして用いようとする提案がいくつか見られる[7,9,10]。このパラダイムは、プログラムをできる限り「静的」に理解できるようにしようというプログラミング言語の長年の目標に対する手法として有効であると思われる。しかし構文のパラダイムで記述することが最も自然であるような問題ばかりではなく他のパラダイムがよく適合する問題も多いと考えられる。問題が大きい場合には、一部に構文として表現することが適当な部分を含んでいることがあるであろう。そのために構文のパラダイムを他のパラダイムに基づく言語に融合する必要がある。本稿ではデータ型として融合する方法を提案し、言語とその処理系を開発し有効性を確かめた。

参考文献

[1]　International Telegraph and Telephone Consultative Committee: Recommendations X.409 - Presentation transfer syntax and notation, June (1984)

[2]　Johnson, S.C.: Yacc- Yet Another Compiler Compiler, UNIX Programmer's Manual (1978)

[3]　Lesk, M.E.: Lex- A Lexical Analyzer Generator, UNIX Programmer's Manual (1987)

[4]　伊藤　敦之、大竹　和雄：　プロトコル処理技術、　ＮＥＣ技報　Vol.40 No.1 (1987), pp.58-64.

[5]　大竹　和雄：　プロトコルにおけるフォーマット変換の形式的記述、　情報処理学会第３３回全国大会論文集 (1986), pp.1113-1114.

[6]　大竹　和雄：　プロトコルにおけるフォーマット処理記述言語とそのプログラム生成系、　情報処理学会第３５回全国大会論文集 (1987), pp.1057-1058

[7]　片山　卓也：　属性文法型計算モデル、　情報処理　Vol.24 No.2 (1983)

[8]　佐々　政孝：　コンパイラの自動生成、情報処理(1987), pp.1359-1367

[9]　佐藤　豊：ＣＯＳＭＯＳ上の構文指向プログラミング、　情報処理学会第３６回全国大会論文集 (1983), pp.1085-1086

[10]　中田　育男、山下　義行：　ＣＣＦＧプログラムにおけるプログラム変換、コンピュータソフトウェア、Vol.5, No.2(1988), pp.41-50.

[11]　三上　理、他：　マイクロメインフレームリンクによるソフトウェア開発環境の実現方式、　情報処理学会第３４回全国大会論文集(1987), pp.1171-1172

[12]　三上　理、他：　複数のオペレーティングシステム環境とターゲットマシンリンクシステムの開発、情報処理学会オペレーティングシステム研究会(1987)、報告３７－１