プログラミング言語 13－2
（１９８７・９・１６）

入力インターフェース用日本語サブセットの構成法とその応用

萩原　正也　　徳田　雄洋

山梨大学　工学部　計算機科学科

本報告では、プログラミングシステムの立場から、日本語の文をＵＮＩＸシェルプログラムへ変換する翻訳系の作成に用いる人工的日本語サブセットの構成方法として、後置演算子文法を提案する。

後置演算子文法は、一種類の非終端記号と三種類の終端記号（演算子・変数・区切り記号）の関係を文法規則として記述したものである。この後置演算子文法は、その構文解析が常に一意にできることを特徴としている。さらに、単純後置翻訳法などを組み合わせることによって比較的簡単に翻訳系を作成することができる。

A method of constructing subsets of Japanese Language using postfix operator grammars and its applications

Masaya Hagiwara and Takehiro Tokuda

Department of Computer Science, Faculty of Engineering,
Yamanashi University, 4-3-11 Takeda, Kofu 400, Japan

We propose a subclass of context-free grammars, which we call postfix operator grammars. Postfix operator grammars consist of productions of the following form, S → var, or S → S a1 … S an op, where S is a nonterminal, var and op are terminals, and a1, … ,an are strings of terminals of length zero or more. Strings a1, … , an work as delimiters when they are not empty. A terminal op belongs to a special class of terminals which work as postfix operators. Postfix operator grammars are unambiguous. Hence this grammar class is suitable for constructing subsets of Japanese language which can be used as input languages of programming systems such as a translator of Japanese sentences into command procedures.

１．　はじめに

本論文の目的は、人工的日本語をａｗｋやｓｅｄを用いたＵＮＩＸのシェルプログラムに翻訳する際に使用する文法を簡易的に構成する方法を示すことである。

プログラミングシステムの分野では、構文解析手法としてＬＬ(k)系やＬＲ(k)系やEarleyのアルゴリズムなどがあげられる[1]。ＬＬ(k)系やＬＲ(k)系の構文解析手法はそれぞれ文脈自由文法の部分集合に対して効率がよい。これは、構文解析の準備として構文解析表を作成するからである。しかし、文法が与えられたときのＬＬ(k)条件やＬＲ(k)条件の判定は、構文解析表を作成するまでは一般的には判明しない。また、文法が大きくなると、それにともない構文解析表用の記憶領域は大きなものになる。さらに、文法を拡張するたびごとに構文解析表を生成し直す必要がある。Earleyのアルゴリズムは一般的な文脈自由文法に対して有効であり、曖昧さがあってもなくても解析は可能である。しかし、文脈自由文法の部分集合に対しての効率は、ＬＬ(k)系やＬＲ(k)系と比べてあまりよくない。

本論文で示す方法は、従来からプログラミングシステムの分野で用いられてきた抽象木や演算子・被演算子のモデルをもとにして、文脈自由文法の部分集合であり、曖昧さがなく構文解析法が簡単な後置演算子文法を提案し、それを日本語に対して導入することにより、人工的日本語サブセットを簡易的に構成するものである。この方法による人工的日本語サブセットは、

・構文解析が単純な方法で、しかも一意に行うことができる。
・自然な日本語表現に近いものが得られる。
・翻訳が比較的容易である。

といった特徴をもっている。

本論文の構成は次の通りである。第２章では、本論文の動機について述べる。第３章では、一般的な後置式を生成する文法について数学的証明も交えて説明する。第４章では、第２章・第３章の理論を基に、本論文で提案する後置演算子文法について述べる。第５章では、後置演算子文法の構文解析について述べる。第６章では、後置演算子文法による人工的日本語サブセットとその応用例として、

日本語によるＣの型宣言を英語に翻訳する日本語サブセット
ＵＮＩＸのシェルプログラムへの翻訳を行う簡単な日本語サブセット

を示す。第７章では、後置演算子文法の拡張について述べる。第８章では、現状と今後について述べる。

ＵＮＩＸは、ＡＴ＆Ｔ　ベル研究所の登録商標である

２．　動機

プログラミングシステムのコンパイラ生成系や構造エディタ生成系で取り扱っている基本モデルは、図１のようなモデルである。

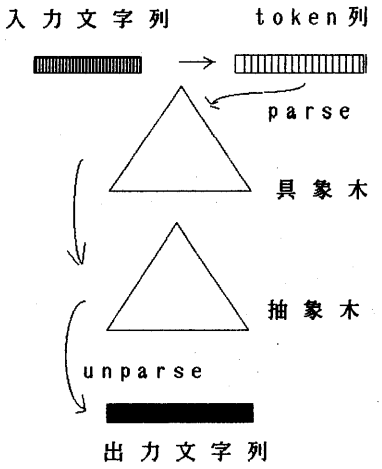


図１　基本モデル

例えば、カーネギーメロン大学のGandalf ソフトウェア開発環境で使用されている構造エディタＡＬＯＥでは、次のように抽象木を利用している[13]。

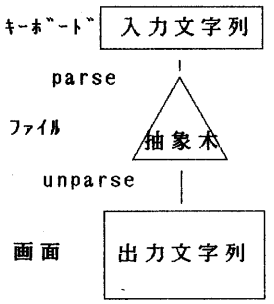


図２　ＡＬＯＥにおける抽象木の利用

ＡＬＯＥにおけるデータ構造は、単なる文字の集まりではなく、内部表現として木構造になっている。この木構造は、解析木から冗長な情報を消去した抽象的な木になっている。この抽象木は、編集の対象となるプログラミング言語に依存している。

しかし、ファイルの抽象木構造をそのまま端末でみることはできず、抽象木に予約語やセミコロン・かっこ・タブ・スペースなどを追加して端末上で見やすい形にする操作を行う。これをアンパースと言う。

このような抽象木の考え方は、プログラミングシステムの他の分野や自然言語処理の分野でも一般的である。例えば、ウィーン定義言語[19]や、変形生成文法[5]や格文法[6]、概念依存文法[14]などである。

もう一つの基本モデルの例として、中置式から後置式を翻訳するプロセスがある。そのプロセスは次のようになる。

《中置式から後置式を生成するプロセス》

G) 中置式の文法

E→E+T　　E→T

T→T*F　　T→F

F→i

U) 後置式を生成するアンパース手続き

unparse(x)…ｘを親ノードとする二分木のアンパースを行う手続き

ｘ：具象木のノードを表す

output(c)…文字ｃを出力する手続き

unparse(+)→unparse(左の子);unparse(右の子);
output('+');

unparse(*)→unparse(左の子);unparse(右の子);
output('*');

unparse(i)→output(ｉに付いているラベル);

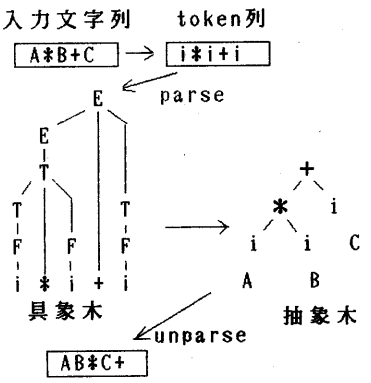


図３　unparse手続きの例

算術式はまず token列に変換され、中置式の文法による構文解析で具象木を生成する。具象木から不必要な情報を消去し抽象木を生成する。抽象木に対し後置式を生成するアンパース手続きを行うことによって後置式が生成される。

日本語の構文構造や意味構造も、プログラミングシステムで扱うように演算子・被演算子関係の抽象木にあてはめて取り扱い、抽象木にアンパース手続きを用いることによって翻訳を行いたい、というのが本論文の動機である。。

日本語の簡単な抽象木をつくるために、日本語の語句のかかり受けの関係を演算子と被演算子の関係に考えると、次のようになる。

1. 前置演算子：
   例　形容詞と名詞　演算子　形容詞
   　　被演算子　名詞
   　　赤い　本
2. 中置演算子：
   例　助詞と　演算子　と
   　　被演算子　名詞
   　　本　と　ノート
3. 後置演算子：
   例　目的語と動詞　演算子　動詞
   　　被演算子　目的語
   　　学校へ　行く
4. ０項演算子：
   例　名詞　演算子　名詞
   　　被演算子　なし
   　　学校

これらの４種類の演算子の組合せは、一般には曖昧な文を生成する。

1)中置演算子の存在は単独で曖昧となる。

例　文法　S → S + S

　　S → a

　文　a＋a＋a

2)前置演算子と後置演算子の共存は曖昧となる

例　文法　S → f S

　　S → S g

　　S → a

　文　f a g

3)同一の前置演算子（または後置演算子）が異なる個数の被演算子を持つ場合は曖昧となる

例　文法　S → S －　（負の符号）

　　S → S S －　（減算）

　　S → a

　文　a a－－

以上の考察から、一意に構文解析できる文法を得るためには、後置演算子と０項演算子の組合せ、または、前置演算子と０項演算子の組合せでなければならないことになる。

日本語　ＡとＢの積とＣの和

後置式　ＡＢ＊Ｃ＋

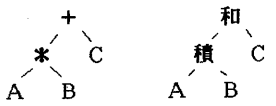


図４　後置式と日本語の類似

一方、図４に示すように日本語の表現と後置式の表現には、強い類似性がある。この点に注目し、後置演算子と０項演算子の組合せを用いることにより、一意的に構文解析できる人工的日本語文法を得ることができると考えられる。

３．　後置式を生成する文法

構文解析が一意的にできる人工的日本語文法を得るために、まず後置式を生成する文法について述べる。

３．１　後置式文法

【定義】　後置式の定義

後置式を次のように定義する。

(1) 変数１個の式は後置式である。
(2) 後置式を１個以上ｋ個（１≦ｋ）被演算子として並べ、最後にｋ項演算子を１個書いた式は後置式である。（演算子と変数は異なるものである。）
(3) (1),(2) で得られた式だけを後置式と呼ぶ。

【定義】　後置式文法の定義

次の形をした文脈自由文法を、後置式文法という。但し、文法記号で、Ｓは特定の非終端記号であり、var，opは、終端記号である。

非終端記号　Ｓ
終端記号　　ＶＡＲＵＯＰ
　の集合　　VAR:変数の集合
　　OP:演算子の集合
　　（ＶＡＲ∩ＯＰ＝Φ）
文法規則　Ｓ　→　var
　　Ｓ　→　Ｓ　Ｓ　…　Ｓ　op
　　var∈ＶＡＲ
　　op∈ＯＰ

このとき、右辺のそれぞれのＳを被演算子と呼び、Ｓの個数は１以上である。opは演算子である。

例１　解析が一意な後置式文法

非終端記号　Ｓ
終端記号　　ＶＡＲＵＯＰ
　の集合　　ＶＡＲ＝｛var｝
　　ＯＰ＝｛＋，＊｝
文法規則　　Ｓ→var
　　Ｓ→ＳＳ＋
　　Ｓ→ＳＳ＊
生成文の例　var var var ＋　＊

例２　解析が曖昧な後置式文法

非終端記号　Ｓ
終端記号　　ＶＡＲＵＯＰ
　の集合　　ＶＡＲ＝｛var｝
　　ＯＰ＝｛－｝
文法規則　　Ｓ→var
　　Ｓ→Ｓ－　　（負の符号）
　　Ｓ→ＳＳ－　（減算）
生成文の例　var var －　－

このとき例１は曖昧な文法ではないが、例２は曖昧な文法である。

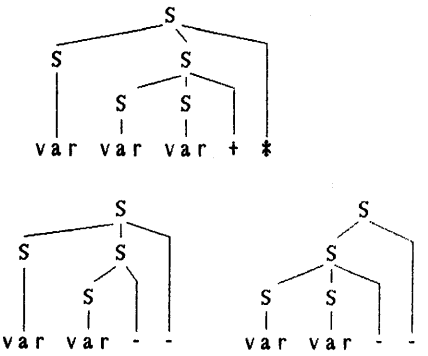


図５　例１と例２の解析木

後置式文法のうち、特に文法規則中の右辺の右端のopがすべて相異なる演算子であるならば、この場合の文法を限定後置式文法と呼ぶことにする。限定後置式文法では、文法規則の右辺に含まれる被演算子の個数が演算子記号に対して一意に決まるので、この被演算子の個数を、演算子の項数と呼ぶことにする。

３．２　限定後置式文法の無曖昧性

（定理１）　限定後置式文法で生成されるような文は、２通りの異なる構文解析木を持つことはない。つまり限定後置式文法は、曖昧ではない。

《定理の証明》

まず次の補助定理１を証明する

（補助定理１）　限定後置式文法で使用する終端記号ｘに対して、関数ｇ（ｘ）を次のように定義する。
g(var)=+1
g(op)=-(j-1)　但しｊは演算子ｏｐの項数である。
（１≦ｊ）
また、限定後置式文法から生成された終端記号列Ｘ1Ｘ2…Ｘnに対して、関数ｆ(i)(1≦i≦n)を、次のように定義する。
f(i)=g(x1)+ … +g(xi)
このとき次のことがいえる。
f(i)≧1 (1≦i≦n-1)
f(n)=1　　(1)

つまり１つの後置式の関数ｆの値は１となり、１つの後置式の長さｎ－１以下の前部分式の関数ｆの値は１以上となる。

《補助定理１の証明》

限定後置式文法で生成された文に含まれている演算子ｔに関する数学的帰納法を用いて証明する。

ｔ＝０のとき、限定演算子文法で生成される文はvarであり、

f(1)=g(var)=1

よって（1)式が成立する。

ｔ＜Ｎのとき(1)式が成立したとする。

ｔ＝Ｎのとき､次のことがいえる。

限定後置式文法から生成された終端記号列Ｘ１Ｘ２…Ｘｐ…Ｘｉ…Ｘｓ、(1≦s,1≦p,i≦s)について、終端記号Ｘｓはℓ個の後置式をもつ演算子であるとする（１≦ℓ）。ℓ個の後置式に含まれている演算子の個数は最大Ｎ－１（＜Ｎ）であるので、帰納法の仮定より各後置式の関数ｆの値は１である。終端記号Ｘｉの左側で最も近いところにある演算子をｏｐ’（＝Ｘｐ）そこまでにある後置式の個数をＬ個とすると、（ｏｐ’が存在しない場合はＬ＝０、ｏｐ’＝Ｘｉのときはℓ＝Ｌ）

$$f(i)=g(x_1)+\cdots+g(x_p)+g(x_{p+1})\cdots+g(x_i)$$
$$=1*L+1*(i-p)\geqq 1$$
$$f(s-1)=1*\ell\geqq 1$$
$$f(s)=f(s-1)+g(xs)$$
$$=\ell-(\ell-1)$$
$$=1$$

で、成立する。　　補助定理１　証明終了

定理１の証明は、この補助定理１を用いて背理法で証明する。

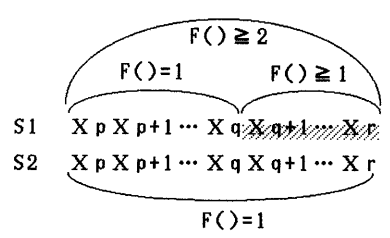
限定後置式文法で生成された終端記号列をＸ１Ｘ２…ＸｎとしＸｎがℓ個の後置式をもつ演算子であるとする。このとき終端記号列Ｘ１Ｘ２…Ｘｎがℓ個の被演算子に一意に区切ることができないと仮定する。つまり、左端が一致して右端が一致しない被演算子

Ｓ１＝ＸｐＸｐ+1…Ｘｑ

Ｓ２＝ＸｐＸｐ+1…ＸｑＸｑ+1…Ｘｒ

が存在すると仮定する。

Ｓ２に注目すると、Ｓ２は演算子Ｘｒに関する後置式だから関数ｆ(r)の値は１である。



一方Ｓ１に注目すると、Ｓ１は演算子Ｘｑに関する後置式だから関数ｆ(q)の値は１になる。このときＸｐ+1…ＸｑはＳ１の次の後置式の前部分列にあたるので関数ｆの値は１以上となる。よってＳ１に注目するとＸｐ…Ｘｒの関数ｆ(r)の値は２以上となる。

同じ終端記号列Ｘ１Ｘ２…Ｘｎに対して一意な解を持つはずの関数ｆが２つの異なる値を持ってしまう。これは矛盾である。

よって、左端が一致して右端が一致しない被演算子は存在しない。被演算子の区切りは一意である、つまり２通りの構文解析木を持つことはないことがわかる。　　証明終了

このように、限定後置式文法は曖昧でなく構文解析が一意に行えることがわかる。次の第４章では、限定後置式文法の拡張を考える。

４．　文法の構成方法

４．１　後置演算子文法

本論文で示す文法の構成方法は、第３章で述べた限定後置式文法を・文法規則中に区切り記号を加える・演算子は、被演算子の個数を一意に決めることによって２つ以上の文法規則に含まれてもよい。の２点について拡張したものである。

構成方法はつぎのとおりである。

【構成方法】

条件１）文法規則の形式は、次の形式のいずれかである。ただし、以下の文法規則のスキームにおいて、非終端記号は１つの特定の非終端記号Ｓであり、終端記号は、var_i, op_iである。また、a_{i1}, a_{i2}, …, a_{ik}は、終端記号列である。(1≦i,j,k)

非終端記号　Ｓ

終端記号の集合　ＶＡＲＵＤＥＬＵＯＰ

VAR:変数の集合

DEL:区切り記号の集合

OP:演算子の集合

（ＶＡＲ∩ＤＥＬ∩ＯＰ＝Φ）

文法規則

$S \rightarrow var_i$

$S \rightarrow S\ a_{i1}\ S\ a_{i2}\ \cdots\ S\ a_{ik}\ op_i$

$var_i \in VAR$

$a_{i1}, a_{i2}, \cdots, a_{ik} \in DEL*$

$op_i \in OP$

すなわち、被演算子はa_{i1}, a_{i2}, …, a_{ik}という終端記号列で区切られ、被演算子と区切り記号列の並びの最後には演算子op_iが存在する。

条件２）各終端記号op_iは、その被演算子の個数が一意的に定まる。すなわち、同一の終端記号op_iに対し、その被演算子の個数はどの文法規則でも同一である。

このとき３つの集合ＶＡＲ，ＤＥＬ，ＯＰは互いに素であるならば、区切り記号の列 a_{i1}，a_{i2}，…，a_{ik} に特に制約はない。文法全体ですべて同一の終端記号でもよいし、また空列でもよい。

この文法構成法で作成された文法を後置演算子文法という

４．２　後置演算子文法の無曖昧性

（定理２）　後置演算子文法で生成されるような文は、２通りの構文解析木を持つことはない。つまり後置演算子文法は、曖昧ではない。

《定理２の証明》

次の補助定理２を証明する

（補助定理２）　後置演算子文法で使用される終端記号ｘに対して、関数ｇ（ｘ）を次のように定義する。

$g(x)=+1$　　$x\in VAR$
$g(x)=0$　　$x\in DEL$
$g(x)=-(j-1)$　$x\in OP$
但しｊは演算子ｏｐの項数である。
（$1\leqq j$）

また、後置演算子文法で生成された終端記号列Ｘ１Ｘ２…Ｘｎに対して、関数ｆ(ｉ)($1\leqq i\leqq n$)を、次のように定義する。

$f(i)=g(x1)+\cdots+g(xi)$

このとき次のことがいえる。

$f(i)\geqq 1$　($1\leqq i\leqq n-1$)
$f(n)=1$　　(1)

つまり１つの後置式の関数ｆの値は１となり、１つの後置式の長さｎ－１以下の前部分式の関数ｆの値は１以上となる。

《補助定理２の証明》

後置演算子文法で生成された文に含まれている演算子ｔに関する数学的帰納法を用いて証明する。

ｔ＝０のとき、文法Ｇで生成される終端記号列はvarであり、

$f(1)=g(var)=1$

よって（1)式が成立する。

ｔ＜Ｎのとき(1)式が成立したとする。

ｔ＝Ｎのとき、次のことがいえる。

終端記号列Ｘ１Ｘ２…Ｘｐ…Ｘｉ…Ｘｓ、($1\leqq s$，$1\leqq p$，$i\leqq s$)について、終端記号Ｘｓはℓ個の後置式をもつ演算子であるとする（$1\leqq\ell$）。ℓ個の後置式に含まれている演算子の個数は最大Ｎ－１（＜Ｎ）であるので、帰納法の仮定より各後置式の関数ｆの値は１である。終端記号Ｘｉの左側で最も近いところにある演算子をｏｐ’（＝Ｘｐ）そこまでにある後置式の個数をＬ個とすると、（ｏｐ’が存在しない場合はＬ＝０、ｏｐ’＝Ｘｉのときは$L=\ell$）

$f(i)=g(x_1)+\cdots+g(x_p)+g(x_{p+1})\cdots+g(x_i)$
$=1*L+1*(i-p)\geqq 1$
$f(s-1)=1*\ell\geqq 1$
$f(s)=f(s-1)+g(xs)$
$=\ell-(\ell-1)$
$=1$

で、成立する。　　補助定理２　証明終了

定理２の証明は、後置式文法の定理１の証明と同様であるので省略する。　　定理２　証明終了

《文法例》　図４の日本語の抽象木に対する具象木を生成する文法

非終端記号　Ｓ
終端記号の集合　ＶＡＲＵＤＥＬＵＯＰ
　ＶＡＲ＝｛Ａ，Ｂ，Ｃ｝
　ＤＥＬ＝｛と，の｝
　ＯＰ＝｛和，積｝

文法規則
Ｓ → Ｓ と Ｓ の 和
Ｓ → Ｓ と Ｓ の 積
Ｓ → Ａ
Ｓ → Ｂ
Ｓ → Ｃ

例文　Ａ と Ｂ の 積 と Ｃ の 和

５．　構文解析方法

第４章で定義した後置演算子文法の構文解析方法は次の通りである。

【構文解析方法】

環境
入力文字列　Ｘ１ … Ｘｎ　（$1\leqq n$）
構文解析用スタック　Ｐ
文法規則照会用配列　Ｍ

解析手順

(1)初期化
スタックＰ、配列Ｍを空にする。
文字Ｘ１をスタックＰにプッシュする。
(2)解析
スタックＰの先頭文字Ｙが
① Ｙ∈ＶＡＲ：Ｓ→Ｙで還元する。
② Ｙ∈ＤＥＬ：何もしない
③ Ｙ∈ＯＰ　：
1)演算子Ｙが持つ被演算子の項数ｍ（$1\leqq m$）を調べ、スタックＰの先頭からｍ番目のＳまでの内容を配列Ｍにコピーする。
2)配列Ｍの内容と右辺の右端の演算子がＹであるような文法規則のパターンマッチングを行う。

マッチするような文法規則があればS → S a_{i1} … S a_{ik} Yで還元を行う。なければ、異常終了である。

④それ以外の場合：異常終了である。

(3)判定

入力文字列がまだ残っていれば、先頭文字を1つスタックPにプッシュし、再びステップ（2）へ行く。

入力文字列がなくなった場合、スタックPの内容がSだけならば正常終了である。それ以外であれば異常終了である。

6． 文法例と翻訳例

後置演算子文法を用いて構成された日本語サブセットの2つの文法例を示す。文法構成の際、演算子はキーワードとなる動詞や名詞など、区切り記号は助詞などが相当している。

G1． 言語Cの型宣言の日本語表現の文法例

非終端記号　S
終端記号　VARUDELUOP
　の集合　VAR＝{整数，文字}
　　DEL＝{へ，の，を，に，要素，持つ，返す}
　　OP＝{ポインタ，配列，関数}
文法規則
　S → S へ の ポインタ
　S → S を 要素 に 持つ 配列
　S → S を 返す 関数
　S → 整数
　S → 文字

この文法から生成される文の例には、次のようなものがある。

　文字 へ の ポインタ を 要素 に 持つ 配列
　整数 へ の ポインタ を 返す 関数

G2． UNIXコマンドawk用日本語文法

非終端記号　S
終端記号　VARUDELUOP
　の集合　VAR＝{1，2，myfile，yourfile}
　　DEL＝{を，と，の}
　　OP＝{出力する，和，積，番目，フィールド}
文法規則
　S → S を 出力する
　S → S と S の 和
　S → S と S の 積
　S → S の S の フィールド
　S → S 番目
　S → 1
　S → 2
　S → myfile
　S → yourfile

この文法から生成される文の例には、次のようなものがある。

myfile の 1 番目 の フィールド と yourfile の 2 番目 の フィールド の 積 を 出力する。

myfile の 1番目 の フィールド と yourfile の 1 番目 の フィールド の 和 と yourfile の 2 番目 の フィールド の 積 を 出力する。

2つの文法例を翻訳する例を示す。T1.は、日本語の型宣言を英語の型宣言になおしている。T2.は、日本語のawkコマンド動作指定をUNIXコマンドに翻訳している。翻訳の定義は、属性文法[9]を用いている。

T1． G1.を用いたCの型宣言の日本語から英語への翻訳

```
S0 → S1 へ の ポインタ
   code(s0) = "a pointer to" + code(s1)
S0 → S1 を 要素 に 持つ 配列
   code(s0) = "an array of" + code(s1)
S0 → S1 を 返す関数
   code(s0) = "a function
                returning" + code(s1)
S → 整数
    code(s) = "integer"
S → 文字
    code(s) = "character"
```

翻訳例

```
文字 へ の ポインタ を 要素 に 持つ 配列
  → an array of a pointer to character
整数 へ の ポインタ を 返す 関数
  → a function returning a pointer
      to integer
```

T2． G2.を用いた日本語からシェルプログラムへの翻訳　ただしcatsは、横方向に空白をいれてファイルを結合するコマンドである。

```
S1 → S2 を 出力する
   code(s1) =
     code(s2) + "lpr place(s2) ¥n"
S1 → S2 と S3 の 和
  place(s1) = newname()
   code(s1) = code(s2) + code(s3) +
     "cats place(S2) place(s3) |
     awk '{print $1+$2}' > place(s1) ¥n"
```

```
S1 → S2 の S3 の 積
  place(s1) = newname()
   code(s1) = code(s2) + code(s3) +
     "cats place(S2) place(s3) |
      awk '{print $1*$2}' > place(s1) ¥n"
S1 → S2 の S3 の フィールド
  place(s1) = newname()
   code(s1) = code(s2) + "cat place(s2) |
      awk '{print part(s3)}' > place(s1)
      ¥n"
S1 → S2 番目
   part(s1) = "$num(s2)"
S → 1
    num(s) = "1"
S → 2
    num(s) = "2"
S → myfile
   place(s) = "myfile"
    code(s) = ""
S → yourfile
   place(s) = "yourfile"
    code(s) = ""
```

翻訳例

myfile の 1 番目 の ﾌｨｰﾙﾄﾞ と yourfile の 2 番目 の ﾌｨｰﾙﾄﾞ の 積 を 出力する。

```
→ cat myfile | awk '{print $1}' > temp1
  cat yourfile | awk '{print $2}' > temp2
  cats temp1 temp2 |
              awk '{print $1*$2}' > temp3
  lpr temp3
```

myfile の 1番目 の ﾌｨｰﾙﾄﾞ と yourfile の 1 番目 の ﾌｨｰﾙﾄﾞ の 和 と yourfile の 2 番目 の ﾌｨｰﾙﾄﾞ の 積 を 出力する。

```
→ cat myfile | awk '{print $1}' > temp1
  cat yourfile | awk '{print $1}' > temp2
  cats temp1 temp2 |
          awk '{print $1+$2}' > temp3
  cat yourfile | awk '{print $2}' > temp4
  cats temp4 temp5 |
          awk '{print $1*$2}' > temp5
  lpr temp5
```

ＵＮＩＸシェルプログラムへの翻訳では、一つのコマンドの終了ごとにその結果を一時的な仮ファイルを生成している。これを、ＵＮＩＸのパイプ（｜）を使うことによって処理を行うことは、今後の課題である。

７．　議論

後置演算子文法は、ＵＮＩＸシェルプログラム翻訳の日本語インターフェース用の日本語サブセットを構成するに際して有効である。後置演算子文法に対しては、さらに次の点に関しての拡張を考えている。

1．被演算子の置換

後置演算子文法は、日本語で語順が比較的自由であるのと同様に、被演算子の順番に対して比較的自由である。被演算子の置換に対しては、構文解析方法を次のように行う。

【構文解析方法】
(2)解析
スタックＰの先頭文字Ｙが
① Ｙ∈ＶＡＲ：Ｓ→Ｙで還元する。
② Ｙ∈ＤＥＬ：何もしない
③ Ｙ∈ＯＰ　：
　　1)演算子Ｙが持つ項数ｍを調べ、スタックＰの先頭からｍ番目のＳまでの内容を配列Ｍにコピーする。
　　2)配列Ｍの内容が、右辺の右端の演算子がＹであるような文法規則の置換列になっていないかを調べる。置換列になっている文法規則があればＳ → Ｓ … Ｓ a_{ik} Ｙで還元を行う。なければ、異常終了である。
④それ以外の場合：異常終了である。

2．被演算子の省略

後置演算子文法は、日本語で語の省略ができるのと同様に、被演算子を省略することも部分的には可能である。例えば、文法構成方法の条件に次の条件が加わる。
条件３）省略できる部分は前部分列だけとする。
条件４）各文法規則の区切り記号列の集合は、互いに素である。また、意味的に自分の文法規則の右辺の中に自分を埋め込むことはない。

このとき構文解析方法は次のように行う。

【構文解析方法】
(2)解析
スタックＰの先頭文字Ｙが
① Ｙ∈ＶＡＲ：Ｓ→Ｙで還元する。
② Ｙ∈ＤＥＬ：何もしない
③ Ｙ∈ＯＰ　：
　　1)演算子Ｙが持つ項数ｍを調べ、スタックＰの先頭からｍ番目のＳまでの内容を配列Ｍにコピーする。
　　2)配列Ｍの内容と右辺の右端の演算子がＹであるような文法規則の後部分列とパターンマッチ

ングを行う。マッチするような文法であればS → S a_{11} … S a_{ik} Yで還元を行う。そうでなければ被演算子と区切り記号の組を1組スタックからポップして、2)を行う。

④それ以外の場合：異常終了である。

8. おわりに

現在、本論文の後置演算子文法をもとにシェルプログラムへの翻訳系に使用する人工的日本語サブセットを構成中である。今後は、付録にあるようなユーザ登録手続きや、その他採点集計プログラムのようなコマンドプログラムを翻訳可能な人工的日本語文法を構成していく。

《参考文献》

[1] Aho,A.V. and Ullman,J.D. : The Theory of Parsing,Translation,and Compiling,Vol.1 and 2, Prentice-Hall (1972 and 1973).

[2] Aho,A.V. and Ullman,J.D. : Principles of Compiler Design,Addision-Wesley (1977).

[3] Aho,A.V. and Sethi,Ullman,J.D.: Compilers principles,Techniques,and Tools,Addison-Wesley Publishing Company (1986).

[4] Aho,A.V., Kernighan,B.W. and Weinberger, P.J. : Awk,-a pattern scanning and processing language,Unix User's Reference Manual, 4.3 BSD University of California Berkeley

[5] Chomsky,N.: Aspects of the Theory of Syntax,MIT press (1957).

[6] Fillmore,C.: The Case for Case,in Bach,E. and Harms,R. (eds) Universals in linguistic Theory,Holt,Reinhart and Winston (1968).

[7] 淵一博 監修 古川康一・溝口文雄 共編 自然言語の基礎理論，共立出版，(1986)

[8] Knuth,D.E. : On the translation of Languages from Left to Right, Inf.Cont.,Vol.8,No.6, pp.607-639 (1965).

[9] Knuth,D.E. : Semantics of context-free languages, Math. Systems Theory, Vol.2, No.2 pp.127-145 (1968).

[10] Lewis,P.M.II and Stearns,R.E. : Syntax directed transduction,J.ACM,Vol.15,No.3,pp.465 -488 (1968).

[11] Montegue,R: The Proper Treatment of Quantification in Ordinary English, Approaches to Natural Language. D.Reidel Pub. (1973).

[12] 長尾 真：言語工学、昭晃堂 (1983)

[13] Notkin,D. : The GANDALF Project,The Journal of Systems and Software, Vol.5, no.2, pp.91-106 (1985).

[14] Shank,R.C., Abelson,R.P. : Scripts, Plans and Knowledge, IJCAI 4 (1975)

[15] 田村直良、高倉伸、片山卓也：自然言語処理を目的とした属性文法評価システム、コンピュータソフトウェア、Vol.3,No.3 (1986).

[16] Teitelbaum,T. and Reps,T. : The Cornell Program Synthesizer, A syntax-directed Programming Environment,Comm,ACM,Vol.24,No.9,September ,pp.563-573 (1981).

[17] Teitelbaum,T, Reps,T. and Horwitz,S. : The Why and Wherefore of the Cornell Program Synthesizer,SIGPLAN Notices, Vol.16, No.6, pp8 -16, The Proceedings of the ACM SIGPLAN/SIGOA, Symposium on Text Manipulation (1981).

[18] Tomita, M.: An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications,Computer Science Department Carnegie-Mellon University (1985).

[19] Wegner,P.: The Vienna definition language Computing Surveys 4:1,5-63.

[20] Winograd,T.: Language as a Cognitive Process,Addison-Wesley (1983).

[21] Woods,W. : Augumented transition network grammar for natural language analysis,CACM, Vol.13,pp.589-602 (1970).

付録 ユーザー登録を行うシェルプログラム

ユーザー登録を行うメインプログラム

```
num="0"
count="1"
cp /etc/passwd p
cp /etc/group g
echo "How many people do you add?"
read num
echo "login_name UID group_name GID home_direc
      tory real_name"
while test $count -le $num
do
  adduser
  count= expr $count + 1
done
cp g /etc/group
cp p /etc/passwd
vipw
cat err
chmod +x own
chmod +x grp
own
grp
```

実際に登録ファイルに操作を行い登録を行う
プログラム

```
D="/usr/local/lib"
:
: echo "login_name UID group_name GID home_dir
        ectory real_name"
read nnn1 nnn2 nnn3 nnn4 nnn5 nnn0
:
echo "$nnn1::$nnn2:$nnn4:$nnn0:$nnn5:/bin/csh"
 > tmp
cat tmp
:
ttt=`
cat tmp p |
sed -e "s/::/:none:/g" |
sed -e "s/:/ /g" |
awk '
    BEGIN { c=0
            d=0 }
    {if(NR==1) {a1=$1
                a3=$3}}
    {if(NR!=1 && $1==a1) c=1}
    {if(NR!=1 && $3==a3) d=2}
    END{printf "%1d¥n",c+d}
' `
case $ttt in
0) ;;
1) echo "Use another user name;"$nnn1":"$nnn2"
:"$nnn3":"$nnn4 >> err
   exit ;;
2) echo "Use another user number;"$nnn1":"$nnn
2":"$nnn3":"$nnn4 >> err
   exit ;;
3) echo "Use another user name and user number
       ;"$nnn1":"$nnn2":"$nnn3":"$nnn4 >> err
   exit ;;
*) echo $ttt >> err
   exit ;;
esac
:
echo "$nnn3::$nnn4" > tmp1
cat tmp1
mmm=`
cat tmp1 g |
sed -e "s/::/:none:/g" |
sed -e "s/:/ /g" |
awk '
BEGIN {c=0
       d=0
       e=0}
{if(NR==1) {a1=$1
            a3=$3}}
{if(NR!=1 && $1==a1 && $3==a3) c=1}
{if(NR!=1 && $1==a1 && $3!=a3) d=2}
{if(NR!=1 && $1!=a1 && $3==a3) e=4}
END{printf "%1d¥n",c+d+e}
'
case $mmm in
0) echo "$nnn3::$nnn4:$nnn1" > err ;;
1) ;;
2) echo "Incorrect group number;"$nnn1":"$nnn2
":"$nnn3":"$nnn4 >> err
   exit ;;
4) echo "Incorrect group name;"$nnn1":"$nnn2":
        "$nnn3":"$nnn4 >> err
   exit ;;
*) echo $mmm >> err
   exit ;;
esac
:
if test -d $nnn5
then echo "Use another directory name($nnn5);"
         $nnn1":"$nnn2":"$nnn3":"$nnn4 >> err
     exit
fi
:
if mkdir $nnn5
then
  chmod 700 $nnn5
  echo $nnn1 >> name
  echo source $D/stdlogin > $nnn5/.login
  echo source $D/stdcshrc > $nnn5/.cshrc
  chmod 700 $nnn5/.login $nnn5/.cshrc
  echo chown -R $nnn1 $nnn5 >> own
  echo chgrp -R $nnn3 $nnn5 >> grp
  cat p tmp > p1
  cp p1 p
  sed -e "/¥^$nnn3:/s/¥$/,$nnn1/" g > g1
  cp g1 g
  exit
fi
echo "cannot mkdir $nnn5;"$nnn1":"$nnn2":"$nnn
      3":"$nnn4 >> err
```