Definite clause set grammars for free word-order languages

国立国語研究所報告94　研究報告集 9 (1988)

集合型言語の確定節文法

田 中 卓 史

TANAKA Takushi : Definite Clause Set Grammars for Free Word-Order Languages

要旨：日本語のように語順のゆるい言語を形式的に取り扱うための第一段階として，語順を全く持たない言語（集合型言語）を定義し，その言語を計算機上で生成・解析することのできる確定節文法 DCSG を提案する。DCSG を用いると論理プログラミングにおいて陥るある種のループの問題を構文解析の問題に帰着して容易に解決することができる。次に DCSG を集合の変換規則としてとらえ，逆変換のためのオペレータを導入する。このオペレータは確定節文法の下降解析の過程において部分的な上昇解析を可能にする。DCSG はデータ集合の中に構造を見出す種類の問題や事象に従って状態が変化するような問題を一般化された構文解析の問題に帰着して効果的に取り扱うことができる。
キーワード：確定節文法，DCG，構文解析，集合型言語，語順を持たない言語

Abstract: This paper presents definite clause set grammars (DCSG; DCG like formalism) for free word-order languages, as first step to deal with languages such as Japanese, which do not have fixed word-order. A cirtain type of looping problem in logic programming can be solved by using DCSG as generalized parsing problem. DCSG can be extended further by viewing grammar rules as rules for set convesions and by introducing an inverse operator for set conversion into DCSG syntax. A bottom-up mechanism in the top-down parsing process of DCSG can be implemented by using this operator. DCSG is a simple but powerful tool for generalized parsing problems which involve finding structures in a given data set.
Key words: definite clause grammars, DCG, parsing, free word-order language

1. はじめに

句構造文法は，日本語文の述部の構造のみに着目すると，比較的素直に適用することができる。例えば，「食べさせられたくないようだった」に現れる「食べ」，「させ」，「られ」，「たく」，「ない」，…… のような動詞や助動詞，助詞の接続関係はほぼ半順序関係をなし，そのまま有限オートマトンの状態遷移図として見ることができるので，正規言語の生成規則により表すことができる(16)。次に適用できそうな部分は語構成の問題である。単位となる形態素を定めて終端記号とし，意味分類を行って，構造を整理すれば，文脈自由文法の範囲内で多くの複合語を記述できる可能性がある。

一方，句構造文法を用いて文節（連文節）以上の単位を扱う場合には，同一内容の文に対して幾つもの規則が必要となり能率が悪い。例えば，「太郎が学校へ自転車で行く」と「学校へ自転車で太郎が行く」，「自転車で太郎が学校へ行く」，「太郎が自転車で学校へ行く」，「自転車で学校へ太郎が行く」，「学校へ太郎が自転車で行く」はそれぞれ別の規則を用いて生成・解析しなければならない。すなわち，構成要素の順序が重要になる文節よりも短い単位では，句構造文法の取り扱いが有効であるのに対して，構成要素の順序が比較的ゆるやかな文節よりも長い単位では，語順が意味を持つ句構造文法の取り扱いは能率が悪くなる。

そこで，文節以上の構造をより適切に扱うための準備として，語順を持たない言語を仮定し，その性質や取り扱い方法を明らかにしておくことが必要になる。この論文では文脈自由言語から語順を取り去った集合型の言語を定義する。さらに集合型言語を生成・解析するための確定節文法（DCSG）を提案する。集合型言語の確定節文法は直ちに日本語の生成や解析に用いることはできない。しかし，データ集合の中に構造を見出すことの必要な多くの問題を，一般化された構文解析の問題に帰着して，効果的に取り扱うことができる(17)。

日本語確定節文法を構築するには，ここで開発した集合型言語の確定節文法を基礎として，終端記号や各レベルの非終端記号に対して語順や意味など

の拘束条件を付加して行けば良いと考えている。一般化句構造文法（GPSG）の ID/LP 規則はこの考えに近いものである[3]。

確定節文法（DCG）は文脈自由文法の生成規則を述語論理の確定節を用いて表している[14]。生成規則を表す確定節を公理として仮定すると，文法に適合する文の生成と解析は定理証明の過程として機械的に実行することができる。述語論理において，定理の証明を一般的に効率よく行うことは難しい問題であるが，近年，公理を表すのに確定節を用い，推論規則としてユニバーサル・インスタンシエーション（全称例化）とモーダス・ポーネンス（帰結の法則）のみを用いて導かれる定理の証明は機械的な計算の過程として効率よく計算機上において実行できることが明らかになり，この定理証明のメカニズムは論理型の計算機言語として実現されるようになった。確定節文法はそれ自身が計算機上で文の生成・解析を行うことのできる論理プログラムを構成しており，外に何ら特別のプログラムを必要としない。

確定節文法の方法は論理プログラミングに固有の性質を効果的に用いているために，論理型言語の基礎知識なしに文の生成・解析の過程を正しく理解することは困難である。そこで，この論文を自己完結なものにするため，初めに論理型言語の基礎概念を紹介する。

2. 論理型言語の基礎

2. 1 表記法

2. 1. 1 項

一階述語論理において，議論の対象となる個体は項で表される。項は次のいずれかである。

（i） 定数記号

定数記号は特定の個体を表すもので，英小文字で始まる文字列である。

（ii） 変数記号

変数記号は任意の個体を表すもので，英大文字で始まる文字列である。変数記号は常に量記号で束縛されて用いられる（2．1．3）。名前を与える必

要のない無名の変数は1個のアンダーライン“_”で表す。

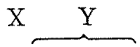
(iii) 複合項

複合項は次の形の表現である。

$f(t_1, t_2, \ldots, t_n)$ $(n \geqq 1)$

ここで，“f”は n 引数の関数記号，“$t_1, \ldots, t_n$”は項である。2引数の関数記号は引数の間に書くことができる。例えば，関数記号“+”はプレフィクス表記 “+(X, Y)”のほかインフィクス表記“X+Y”で書くことができる。

複合項の重要なものにリストがある。リストは項の順序づけられた並びである。最初の要素がXで残りのリストがYとなるリストを複合項“cons(X, Y)”を用いて表す。



空リストを定数記号“nil”で表すと次の項はリスト
a, b, c を表している。

cons (a, cons (b, cons (c, nil)))

通常，リストはより便利な表記法を用いる。上のリストは次のように書くことができる。

[a | [b | [c | []]]] または [a, b, c]

ここで，空リストは“[ ]”で表される。リストの要素はリストを含む任意の項でよい。

2. 1. 2 素論理式

素論理式は次の形の表現である。

$p(t_1, t_2, \ldots, t_n)$ $(n \geqq 0)$

ここで，“p”は n 引数の述語記号，“$t_1, \ldots, t_n$”は項である。2引数の述語記号は“=(X, Y)”のようなプレフィクス表記のほか，“X = Y”のようなインフィクス表記を用いることができる。

2. 1. 3 確定節

確定節は次の形の表現である。

$A :- B_1, B_2, \ldots, B_n.$ $\quad (n \geqq 0)$

ここでA及び $B_1, \ldots, B_n$ は素論理式である。Aを確定節の頭部と呼び，$B_1, \ldots, B_n$ を確定節の本体と呼ぶ。素論理式の間の区切り記号 "," は論理結合子 "and" を略記したものである。記号 ":-" は右辺を条件，左辺を結論とする論理的含意記号である。すなわち，確定節は連言の条件$B_1, \ldots, B_n$ と単一の結論Aを持つ論理的含意である。確定節に現れるすべての変数は全称記号で束縛されているものとして解釈する。変数 $x_1, x_2, \ldots, x_k$ を含んでおれば通常の表記法では次のようになる。

$(\forall x_1)(\forall x_2)\ldots(\forall x_k)(B_1 \wedge B_2 \wedge \ldots \wedge B_n \rightarrow A)$

条件を持たない結論だけの確定節は次のように表し，

A.

無条件に次式が成り立つことを述べている。

$(\forall x_1)(\forall x_2)\ldots(\forall x_k)\ A$

結論を持たない確定節はゴール節と呼ばれる。ゴール節は次のように書かれ，

$?- B_1, B_2, \ldots, B_n.$ $\quad (n \geqq 1)$

各々の原子式 $B_1, B_2, \ldots, B_n$ はゴールと呼ばれる。ゴール節は次のように解釈される。

$(\forall x_1)(\forall x_2)\ldots(\forall xk)\ \neg(B_1 \wedge B_2 \wedge \ldots \wedge B_n)$

ゴール節は背理法（次節 refutation）により次の命題を定理として導くのに使われる。

$(\exists x_1)(\exists x_2)\ldots(\exists x_k)(B_1 \wedge B_2 \wedge \ldots \wedge B_n)$

すなわち，ゴール節が成功する（次節）とその否定が証明されたことになるので変数は存在記号で束縛されたものとして解釈される。

2. 2 論理プログラム

論理プログラムPとはゴール節を除く確定節の有限集合である。この集合は公理に相当する。計算とはその公理から背理法で定理を証明する過程に相当する。計算は次のようにゴール節を加えることでスタートする。

?- $A_1, A_2, \ldots, A_n.$ $\quad (n \geqq 1)$

計算は成功か失敗かのどちらかの結果となる。計算が成功すると，最初のゴール節に含まれる変数の値が計算結果として出力される。与えられたゴール節は幾つかの異なった計算が可能である。計算は非決定的なゴール・リダクションにより進行する。あるリダクションの段階のゴールが $A_1, A_2, \ldots, A_m$ であるとき，A_1 と代入 θ により頭部がユニファイ（後述）できるP中の確定節 "A' :- $B_1, B_2, \ldots, B_k$." が非決定的に選ばれ，新しくリダクションされたゴールは次のようになる。

$(B_1, B_2, \ldots, B_k, A_2, \ldots, A_m)\theta$

計算は現存のゴールが空になったとき（refutation），成功し終了する。リダクションできないときは失敗する。ここで，代入 θ とは "T/X" で表される項の組の有限集合である。Tは任意の項，Xは変数を表す。代入

$\theta = \{T_1/X_1, T_2/X_2, \ldots, T_n/X_n\}$

と式Eがあるとき，$E\theta$ は変数 X_i を項 T_i $(1 \leqq i \leqq n)$ で置き換えた結果の式を表している。$E\theta$ はEのインスタンスと呼ばれる。二つの式EとFを同一にできる代入 θ，すなわち $E\theta = F\theta$，は二つの式のユニファイヤと呼ばれる。EとFの別の任意のユニファイヤ θ_1 による代入の結果 $E\theta_1$ が $E\theta$ のインスタンスとなるとき，θ はEとFの最も一般的なユニファイヤと呼ばれる。二つの式がユニファイできるとき，最も一般的なユニファイヤが唯一存在し，各々のゴール・リダクションに用いられる。

Prolog[1] や Duck[10] などの論理型プログラミング言語はこの背理法による証明の過程を計算機上に実現したものである。これらの計算メカニズムは非決定的なゴール・リダクションをシーケンシャルにシュミレートする。非決定的にリダクションの節を選ぶ代わりに，バックトラックのメカニズムを用いて，すべてのユニファイ可能な節の組合わせを実行する。

このように，証明したい式からスタートする計算の過程は，通常，後向き推論と呼ばれる。Duck のように一部のシステムはモーダス・ポーネンスをそのまま実行する機能も備えている。これは，前提となる式から計算が出発するので，前向き推論と呼ばれる。

Prolog や Duck は確定節の本体に論理結合子 or が現れることを許している。ここでは or を ";" 記号で書くことにする。"(P ; Q)" は次のように定義される。

(P ; Q) :- P.

(P ; Q) :- Q.

例えば，次の節

A :- B, (C ; D).

は次の二つの節と同じ効果をもつが

A :- B, C.

A :- B, D.

Bを2回計算しないので効率がよい。

Prolog や Duck などのプログラミング・システムは幾つかの重要な作り付けの述語を持っている。特に，"not" (Prolog) / "thnot" (Duck) はこの研究に重要なものである。ゴール "not X" / "(thnot X)" はゴールXが失敗した時に成功する。

3. 確定節文法 DCG

3. 1 簡単な日本語文法

確定節文法 DCG (Definite Clause Grammars)[14] による文の生成と解析の動作を理解するために，簡単な日本語句構造文法の生成規則を考える。

s ⟶ caseGa, caseWo, v. (1)

caseGa ⟶ np, [ga]. (2)

caseWo ⟶ np, [wo]. (3)

np ⟶ n. (4)

np ⟶ np, [no], n. (5)

np ⟶ caseGa, v, n. (6)

np ⟶ caseWo, v, n. (7)

n ⟶ [boku]; [ringo]; [nezumi]; [neko]; [shippo]; [inu]; [bou]; [nokogiri]. (8)

v ⟶ [kajitta]; [tsukamaeta]; [hunzuketa]; [kuwaetekita]; [kitta]. (9)

通常の言語理論の表記法と異なり，確定節文法 DCG では終端記号も非終端記号も英小文字で始まる文字列で表す。終端記号は非終端記号と区別するために "[" と "]" とで囲む。区切り記号 "," は終端記号や非終端記号が連なることを表し，区切り記号 ";" は左辺の等しい文法規則を略記するために用いる。(1)は文を格助詞「が」をとる文節と格助詞「を」をとる文節との後に動詞が連なったものと定義している。(2)と(3)は「が」をとる文節と「を」をとる文節を，それぞれ，名詞句に「が」または「を」を付加したものとして定義している。(4)は名詞が単独で名詞句になれることを定義している。(5)は「名詞句＋の＋名詞」を名詞句と定義している。(6)と(7)は「ねずみが噛ったりんご」や「りんごを噛ったねずみ」などの連体修飾された名詞を名詞句と定義している。(8)は名詞に「僕」，「りんご」，「ねずみ」などがあることを定義している。(9)は動詞に「噛った」，「捕まえた」，「踏んずけた」などがあることを定義している。

ここで，(6)，(7)を(2)，(3)を用いて書き換えると次のようになる。

np ⟶ np, [ga], v, n. (6)'

np ⟶ np, [wo], v, n. (7)'

(5)，(6)'，(7)' は生成規則の右辺の左端に左辺と同じ記号が現れる左回帰の文法規則と呼ばれるもので，下降型に構文解析を行うと一般に無限ループに陥る危険性を持っている。下降解析は，対象となる語列に名詞句を同定する際に，(4)が適用できない場合に，(5)を適用するのであるが，(5)は最初に名詞句が来ることを要求し，この要求に対して(4)が適用できず，

再び(5)を適用してループに陥る。下降解析を成功させるには，これらの生成規則を右回帰の形に書き換えることが必要になる。

(5)，(6)，(7)を右回帰の規則に書き換えるには次のように補助的な非終端記号xを導入すれば良い（グライバッハの標準形）[6]。

np ⟶ n, x. (10)

x ⟶ [no], n. (11)

x ⟶ [no], n, x. (12)

x ⟶ caseMarker, v, n. (13)

x ⟶ caseMarker, v, x. (14)

caseMarker ⟶ [ga]; [wo]. (15)

3. 2 文法・確定節変換

確定節文法 DCG の文法・確定節変換プロシージャは（1）を次の確定節(1)' に変換する。

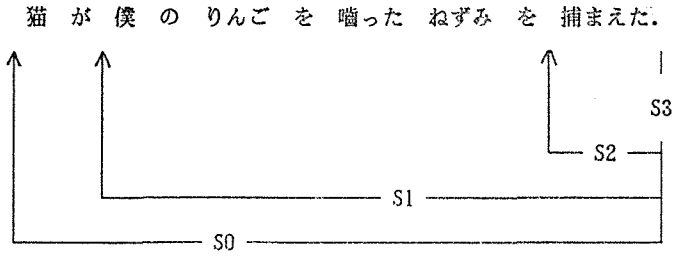
s(S0, S3) :- caseGa(S0, S1), caseWo(S1, S2), v(S2, S3). (1)'

ここで，全称変数 S0, S1, S2, S3 は図1に示すように対象となる語列の特定の位置を示すポインターである。この確定節は通常の述語論理の表記法を用いると次のように書くことができる。

$$(\forall S0)(\forall S1)(\forall S2)(\forall S3)\,(\mathrm{caseGa}(S0, S1) \wedge \mathrm{caseWo}(S1, S2) \wedge \mathrm{v}(S2, S3) \rightarrow \mathrm{s}(S0, S3))$$

すなわち，この命題は対象となる語列において特定の位置 S0 から S1 までが「が」をとる文節で，位置 S1 から S2 までが「を」をとる文節で，位置 S2 から S3 までが動詞であるならば，位置 S0 から S3 までが文となることを述べている。確定節文法 DCG の巧妙な点は，このポインターを表すのにポイント点から文末までの語のリストを用いたことである(図1)。論理プログラムの節として，(1)' を手続き的に見ると「二つのポインター S0 と S3 で挟まれた語列の範囲を文 s として同定するために，S0 から S1 までの範囲を「が」をとる文節として同定し，さらに S1 から S2 までの範囲を「を」をとる文節として同定し，さらに，S2 から S3 までの範囲を1個

の動詞として同定せよ」と読むことができる。



S0 = [neko,ga,boku,no,ringo,wo,kajitta,nezumi,wo,tsukamaeta]

S1 = [boku,no,ringo,wo,kajitta,nezumi,wo,tsukamaeta]

S2 = [tsukamaeta]

S3 = []

図1　語列とポインター

一方，非終端記号から終端記号を生成する規則(8)は次の確定節に変換される。

```
n(S0, S1) :- connect(boku, S0, S1);
             connect(ringo, S0, S1);
             connect(nezumi, S0, S1);
               ・・・
             connect (nokogiri, S0 S1).          (8)'
```

ここで述語 connect は次のように定義されている。

connect (X, [X | Y], Y).　　(16)

すなわち，述語 connect の第2引数にリスト [X | Y] を代入すると，先頭の要素Xと残りのリストYがそれぞれ第1引数と第3引数から得られる。逆に第3引数のリストの先頭に第1引数の要素を加え，第2引数から得ることもできる。

3. 3　DCG による文の解析

入力文の解析を行うには二つのポインターで文の初めと終わりの位置を指

して，その間を出発記号sとして同定すればよい。すなわち，図1の文を解析するには次のゴール節を実行すればよい。

?- s([neko, ga, boku, no, rjngo, wo, kajitta, nezumi wo, tsukamaeta], []).

このゴールは確定節 (1)' により次のサブゴールに展開される。

caseGa ([neko, ga, boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta]. S1),

caseWo (S1, S2),

v (S2, []).

この第一のサブゴールはさらに，生成規則(2)に基づく確定節により，次のサブゴールに展開される。

np([neko, ga, boku, no, ringo, wo, kajitta, nezumi,wo, tsukamaeta], S4).

connect(ga, S4, S1)

この第一のサブゴールは生成規則(4)に基づく確定節により，次のサブゴールに展開される。

n ([neko, ga, boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta], S4)

このゴールは(8)' により展開され，

connect (neko,
[neko, ga, boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta],
[ga, boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta]).

となり，

S4 = [ga, boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta]

となって成功する。ここで一つ前のゴールに戻り，

connect (ga, S4, S1)

が実行され，これも成功して次のように変数の値が得られる。

S1 = [boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta]

これで，最初の「が」の文節が同定できたので，次に「を」の文節を同定するため次のゴールを実行する。

caseWo ([boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta], S2)

これも同様なプロセスで解析が進み，「を」の文節として「僕のりんごを」が仮定され，

S2 = [kajitta, nezumi, wo, tsukamaeta]

で成功する。「を」の文節の後に動詞「噛った」が来るので，最初のゴールの第2引数を存存変数 S3 にしておけば，「猫が僕のりんごを噛った」を一つの文として同定し，S3 の値は次のようになる。

S3 = [nezumi, wo, tsukamaeta]

しかし，実際には第2引数は空リストとなっているので，この形では成功せずバックトラックが起こり，「を」の文節の別解として「僕のりんごを噛ったねずみを」が仮定され，

S2 = [tsukamaeta]

となり，文を同定するための最終のサブゴール

v (S2, [])

が成功する。

3. 4　構文解析木の生成

構文解析木を生成するには解析の過程で成功したゴールの軌跡を残せばよい。そこで次のように文法規則の中に構造を表す項を導入する。

s (s (G, W, v)) ⟶ caseGa (G), caseWo (W), v.

caseGa (cg (NP, ga)) ⟶ np (NP), [ga].

caseWo (cw (NP, wo)) ⟶ np (NP). [wo].

np (np (n)) ⟶ n.

np (np (n, X)) ⟶ n, x (X).

x (x (no, n)) ⟶ [no], n.

x (x (no, n, X)) ⟶ [no], n, x (X).

x (x (m, v, n)) ⟶ caseMarker, v, n.

x (x (m, v, n, X)) ⟶ caseMarker, v, n, x(X).

これらの規則と，(8)，(9)，(15)を用いて，次のゴールを実行すると

?- s (S, [neko, ga, boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta], R).

S = s (cg (np (n), ga), cw (np (n, x (no, n)), wo), v)

R = [nezumi, wo, tsukamaeta];

S = s (cg (np (n), ga), cw (np (n, x (no, n, x (m, v, n))), wo), v)

R = [];

no

バックトラックの機能により二通りの解析結果が得られる。変数Sから構造解析木が得られる（図2 a，b）。変数Rには解析の対象とならなかった入力語列の残りの部分が代入される。

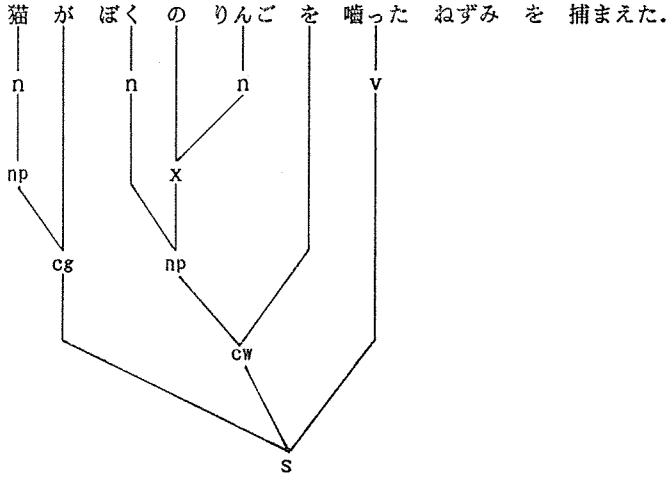


図2—a　入力の一部を文として解析した構文解析木

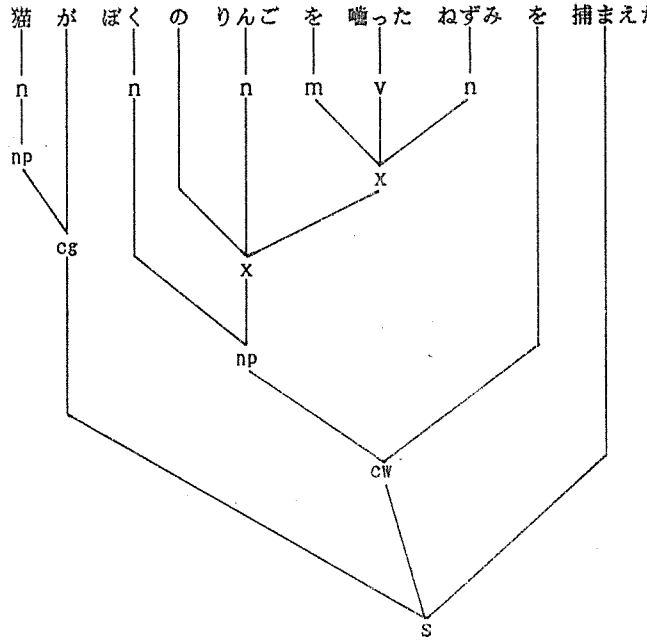


図2—*b*　入力の全体を文として解析した構文解析木

文法規則が左回帰となることを避けるために，3．3節において補助的な非終端記号xを導入した。そのため，名詞句 np より下位の構造は意味的なまとまりを示す構造とは異なっている。3．1節に示した左回帰の規則を直接的に扱うには上昇解析を行うことが必要になる。与えられた文法規則から上昇解析を行う論理プログラムを作る研究も行われている(9)。後述の集合型言語の確定節文法は，文法規則でプロダクション・システムを構築でき，全く異なった方法で上昇解析を行うことができる（詳細は文献(18)参照)。

3．5　DCG による文の生成

一般に論理プログラムにおいて述語に対するデータの入出力の関係は双方向的である。すなわち，文の解析に用いた論理プログラムはそのまま文の生成に用いることができる。しかし，前述の名詞句のように，生成規則が二通りの回帰的定義を含む場合は二通りの名詞句の無限集合が定義されるため，

二つの無限集合から交互に要素を取り出すような工夫をしなければ，文法規則に適合するあらゆる名詞句を生成することはできない。そこで，名詞句の定義(10)—(14)を生成に用いるため，再び左回帰に変更する。

np ⟶ np, z

z ⟶ [no], n.

z ⟶ caseMarker, v, n.

この規則を名詞句の生成に用いると，すべての名詞が名詞句として生成された後に，「n+z」のあらゆる名詞句が生成される。ついで「n+z+z」のあらゆる名詞句が生成される。以下同様に短いものから長いものへと文法規則に適合するすべての名詞句を枚挙することができる。実際には，名詞句の無限集合の枚挙は次のゴールにより実行する。文法規則に意味的な拘束条件を導入していないので，意味的におかしな名詞句が生成される。

```
?- np(X, []).
X = [boku];
X = [ringo];
  ・・・
X = [boku, no, ringo];
X = [boku, no, nezumi];
  ・・・
X = [boku, ga, kajitta, ringo];
X = [boku, ga, kajitta, nezumi];
  ・・・
X = [ringo, wo, tsukamaeta, neko];
X = [ringo, wo, tsukamaeta, shippo];
  ・・・
X = [boku, ga, kajitta, ringo, wo, kuwaetekita, inu];
X = [boku, ga, kajitta, rjngo, wo, kuwaetekita, bou];
  ・・・
```

$\dot{X}$ = [boku, no, ringo, wo, kajitta, nezumi, wo, tsukamaeta, neko, no, shippo, wo, hunzuketa, inu, ga, kuwaetekita, bou, wo, kitta, nokogiri];

・・・

4. 集合型言語の確定節文法 DCSG

4. 1 集合型言語

文脈自由文法Gは形式的に次の四つ組で定義される。

$G = \langle V_N, V_T, P, S \rangle$

ここで V_N は非終端記号の有限集合である。V_T は終端記号の有限集合である。自然言語では終端記号はほぼ単語に相当し，非終端記号は品詞や句，文などの文法的な単位に相当する。Pは次の形をした生成規則の有限集合である。

$A \longrightarrow B_1, B_2, \ldots, B_n.$ $\quad (n \geqq 1)$

ここでAは集合 V_N の要素である。B_i $(i=1, \ldots, n)$ は集合 V_T と V_N の和集合の要素である。生成規則は記号Aをその出現環境とは無関係に記号列 $B_1, B_2, \ldots, B_n$ で書き換えることを意味している。Sは出発記号と呼ばれ文を意味する非終端記号である。文は出発記号Sに生成規則Pを繰り返し適用して導かれる終端記号の列である。生成規則の非決定的な適用は多くの異なった文を導くことになる。この文の集合を文法Gにより生成された言語L(G)と呼ぶ。

文脈自由文法の定義をモディファイして語順を取り去った文脈自由型の集合型言語を定義することができる。生成規則 "$A \longrightarrow B_1, B_2, \ldots, B_n$" の右辺を記号列ではなく，記号の集合を表すものとして解釈すると，生成規則は記号Aを記号の集合 $\{B_1, B_2, \ldots, B_n\}$ で書き換えることを意味する。このモディファイされた文法 G' で定義される集合型言語L(G')の文は終端記号のマルチセット（同一記号を複数個含んでよい記号の集合）[2] となる。マルチセットの導出に用いられた非終端記号はそのマルチセットの部分集合に

与えた名前に相当する。この論文では，生成規則は言語の生成よりも解析に用いることが多いので単に文法規則と呼ぶことにする。

文脈依存型の集合型言語を考えることもできる。通常の文脈依存型の文法規則は次の形をしている。

α, A, $\beta \longrightarrow \alpha$, B_1, ..., B_n, β.　　$(n \geqq 0)$

ここでαやβは終端記号や非終端記号の列（空の記号列を含む）を表し，規則 "A $\longrightarrow$ B_1, ..., B_n" が適用できるための記号Aの文脈上の環境条件となっている。集合型の言語では語順を考えないので文脈の項は一つになる。それまでに生成されている記号の集合の中に，特定の部分集合が含まれるかどうかということが規則の適用を定める環境条件となる。集合型言語の確定節文法（後述）は基本的には文脈自由型の集合型言語に対して定義を行うが，文脈に依存する性質もある程度，扱えるように幾つかの拡張を行う。

4. 2　形式化の基礎

集合型言語の一つの文は出発記号から生成された終端記号のマルチセットである。生成の過程で用いられた非終端記号はこのマルチセットの部分集合に相当する。そこで，部分集合を表す述語 "subset" と集合の要素を表す述語 "member" を用いて，集合型言語の確定節文法 DCSG (Definite Clause Set Grammar) を定義する。ここでは文法規則中に現れる終端記号も非終端記号も英小文字で始まる文字列で表す。文法規則中の終端記号は非終端記号と区別できるように，"[" と "]" とで囲む。文法規則は確定節文法の文法・確定節変換プロシージャにより，確定節に変換される。

次の非終端記号から別の非終端記号を生成する文法規則

s $\longrightarrow$ np, vp.　　(17)

は次の確定節に変換される。

```
subset(s, S0, S2) :- subset(np, S0, S1),
                     subset(vp, S1, S2).        (17)'
```

ここで，S0, S1, S2 などの変数には終端記号 V_T のマルチセットが代入される。マルチセット自体は要素の並びとしてリストを用いて表現する。述語

"subset" は第1引数が第2引数に代入されたマルチセットの部分集合に与えた名前であることを述べている。第3引数はその部分集合を差し引いた残りの集合（補集合）である。例えば，"subset(s, S1, S2)" は "s"が S0 の部分集合で S2 がその補集合であることを述べている。確定節(17)'は，計算上の手続きとして「集合 S0 の部分集合が s で補集合が S2 であることを示すために，集合 S0 の部分集合が np でその補集合 S1 中に部分集合 vp が存在し，S1 から vp を除いた補集合が S2 であることを示せ」と読むことができる。

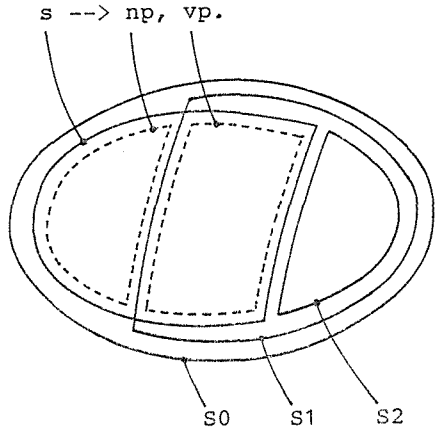


図3　文法規則と部分集合の関係

一方，非終端記号から終端記号を生成する規則

noun ⟶ [book].　　(18)

は次の確定節に変換される。

subset(noun, S0, S1) :- member(book, S0, S1),　　(18)'

ここで，集合の要素を言及する述語 "member" は次の確定節により定義しておく。

member (M, [M | X], X).

member (M, [A | X], [A | Y]) :- member (M, X, Y).　　(19)

述語 member は3個の引数を持っている。第1引数は第2引数に代入され

る集合の要素である。第3引数はこの要素を除いた補集合が代入される。すなわち，集合型言語の確定節文法では終端記号も非終端記号も二つの集合の差（補集合）により表される。

文法規則

$A \longrightarrow B_1, B_2, \ldots, B_n.$

から確定節への変換の一般形は次のようになる。

subset (A, S_0, S_n) :- subset (B_1, S_0, S_1).
subset $(B_2, S_1. S_2)$,
...
subset (B_n, S_{n-1}, S_n).

ここで，文法規則の右辺に含まれる記号は全て非終端記号として仮定されている。もし，"$[B_i]$" $(1 \leqq i \leqq n)$ の形の記号が現れると，"B_i" は終端記号と仮定され，文法・確定節変換の際に "subset$(B_i\ S_{i-1}, S_i)$" の代わりに "member (B_i, S_{i-1}, S_i)" が用いられる。

4. 3 文脈依存の特性

文脈自由型の集合型言語に対して定義した確定節文法の機能を拡張し，ある程度，文脈に依存した構造も扱えるようにしている。通常の文脈依存規則と異なって，右辺に環境条件をテストするための項を導入する。例えば，右辺の記号にオペレータ "test" を作用させた非終端記号 "test C" 及び終端記号 "test [C]"

$A \longrightarrow B_1, \ldots, B_i,$ test C, $B_{i+1}, \ldots, B_n.$

$A \longrightarrow B_1, \ldots, B_i,$ test [C], $B_{i+1}, \ldots, B_n.$

は次のように変換される。

subset $(C, S_i, \_)$

member $(C, S_i, \_)$

右辺に現れるこれらの記号Cは集合の生成の際も解析の際にも規則を適用させる過程において環境条件のテストを行う。生成の際には $B_1, \ldots, B_i$ までの記号を左端から順に終端記号になるまで書き換え，次に，その中に部分集

合Cあるいは要素Cが含まれているかの条件テストを行い，成功すればさらに $B_{i+1}, \ldots, B_n$ の書き換えを行う。失敗するとバックトラックが起こり，最後に行った選択（左辺が同じ規則がある場合，どれか一つを適用する）の別の選択を行い，再び条件テストを行う。

解析の場合には対象となる集合中に非終端記号Aを同定しようとして，初めに部分集合 $B_1, \ldots, B_i$ を同定する。これらの同定された部分集合は順次対象集合から除去されている。この時点で残りの集合（補集合）の中に部分集合Cあるいは要素Cが存存するかの条件テストを行う。テストが成功すれば $B_{i+1}, \ldots, B_n$ を同定に進む。失敗すれば，バックトラックが起こり，最後に行った選択の別の選択を実行する。

オペレータ“test”が環境中に特定の部分集合や要素の存在の条件テストを行ったのに対して，次のようにオペレータ“not”の付加された非終端記号“not C”及び終端記号“not [C]”

$A \longrightarrow B_1, \ldots, B_i,$ not C, $B_{i+1}, \ldots, B_n.$

$A \longrightarrow B_1, \ldots, B_i,$ not [C], $B_{i+1}, \ldots, B_n.$

はそれぞれ次のように変換され，

not subset (C, Si, _)

not member (C, Si, _)

それまでに生成された集合や，これから解析される集合中に特定の部分集合Cや要素Cが存存しないことを条件として要求する。

5. 一般化された構文解析

5. 1 無限ループの問題

完全に語順を持たない言語に対しては，直ちに集合型言語の確定節文法の方法を用いて，文の生成や解析を行うことができる。しかし，文節以上の単位を考えても，日本語文は全く語順がないことはないので，直ちに集合型言語の確定節文法の方法を適用することはできない。語順が全く自由な自然言語が地球上に存存するかどうかは不明であるが，自然言語以外では集合型言

語として見ることのできる多くのデータ集合が存在し，一般化された構文解析の問題として効果的に情報の構造を解析することができる。

集合型言語の確定節文法の長所を明らかにするため，後向き推論におけるループの問題を考える。ある回路の電圧のデータが図2に示すように与えられていたとする。これを次の命題（本体のない確定節）の集合で表し，公理として仮定する。

```
voltage ($1, $3, 20).
voltage ($2, $3, 15).                                    (20)
voltage ($2, $4, 8).
```

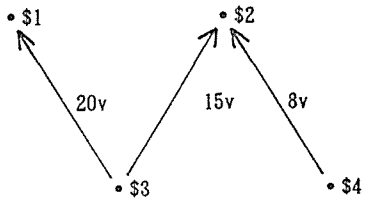


図4　ある回路の電圧

命題 "voltage($1, $3, 20)" は節点$1と$3の間の電圧が20(volt)であることを述べている。節点の順序に関係なく電圧のデータを求めるために，次の含意命題（本体のある確定節）を公理として仮定する。

```
volt(A, B, V) :- voltage(A, B, V).
volt(A, B, -V) :- voltage(B, A, V).                      (21)
```

これは，電圧は逆向きに測るときは負の符号をつければよいことを規則にしたもので，節点 $3, $1 間の電圧は定理 "(∃X)volt($3, $1, X)" を証明すること，すなわち，以上の確定節を論理プログラムとして，次のゴール節

```
?- volt ($3, $1, X).
```

を実行することで得られる。

さらに，任意の二つの節点A，C間の電圧を求めるために，次の二つの確定節を加える。

```
v (A, C, V) :- volt (A, C, V).                           (22)
```

v(A, C, V+W) :- volt(A, B, V), v(B, C, W). (23)

(22) は節点A，C間に電圧のデータが存存するときに適用される。(23) はデータがない場合に適用される。(23) は初めにデータのある節点A，B間の電圧Vを求め，さらに節点B，C間の電圧Wを求めて加え合わせれば良いことを述べている。

しかし，これらの定義は意図したようには働かない。節点 $1, $4 間の電圧を求めるために，次のゴール節を実行したとする。

?- ($1, $4, X).

節点 $1, $4 間の電圧が直接，与えられていないから，ゴールは (23) によりサブゴールに展開される。第1のサブゴールは節点Bに $3 を代入することで "volt ($1, $3, 20)" となり成功する。第2のサブゴール "v ($3, $4, W)" もまた (23) によりサブゴールに展開される。第1のサブゴールは前と同じデータを使って "volt ($3, $1, -20)" で成功する。第2のサブゴールは最初のゴールと同じになってループに陥る。

このような問題を避けるための一つの方法は，同じデータが再度と使われないように，既に使われたデータを消去することである。これは (21) を (21)' で置き換えることにより実現できる。

```
volt(A, B, V) :- voltage(A, B, V),
                 retract (voltage (A, B, V)).
volt(A, B, -V) :- voltage (B, A, V),          } (21)'
                  retract (voltage (B, A, V)).
```

しかし，これは定理の証明の過程で公理系が破壊されるため，消去されたデータがバックトラックの際に回復されることがない。

よく用いられる方法は使ったデータの軌跡を残す方法である。これは(22)と (23) をそれぞれ (22)' と (23)' で置き換えればよい。節点 $1, $4 間の電圧はゴール節 ?- v($1, $4, X, []). により求めることができる。

この方法の欠点は電圧導出の論理に不要なデータの軌跡を陽な形で扱わねばならない点である。

```
v(A, C, V) :- volt(A, C, V).                                    (22)'
v(A, C, V+W, T) :- volt(A, B, V),
                   not member(B, T, _),
                   v(B, C, W, [A | T]).                         (23)'
```

5. 2　構文解析に帰着した問題

前節の無限ループの問題は一般化された構文解析の問題に帰着すると簡単に解決することができる。構文解析の問題にするために，電圧のデータの表現を素論理式から複合項に替える。言語の言葉で言えばは，文で表していたものを名詞句に替えたことに相当する。図4の電圧のデータは複合項の集合として，リストを用いて次の命題で表す。

```
vData([voltage($1, $3, 20), voltage($2, $3, 15), voltage($2, $4, 8)]).
                                                                (24)
```

ここで，リストで表された複合項の集合を集合型言語の一つの文であると仮定し，各々の複合項を単語であると仮定する。任意の節点間の電圧を求める問題は適当な文法規則を定めることにより，構文解析の問題に帰着することができる。次の規則は確定節（21）に対応する。

```
volt(A, B, V) ⟶ [voltage(A, B, V)].
volt(A, B, -V) ⟶ [voltage(B, A, V)].                            (25)
```

"["と"]"とで囲まれた voltage(A, B, V) は終端記号である。一方，volt(A, B, V) は非終端記号である。ここで，通常の文法規則と異なり文法規則の中に全称変数(A, B, V)を導入している。これらの変数は対象となる文に規則が適用される場合に対応する文中の値が代入されるものとする。これらの文法規則は DCSG の文法・確定節変換プロシージャにより次の確定節に変換される。

```
subset(volt(A, B, V), S0, S1) :- member(voltage(A, B, V), S0, S1).
subset(volt(A, B, -V), S0, S1) :- member(voltage(B, A, V), S0, S1).
```

任意の二つの節点間の電圧を求めるために，(22)，(23)に対応して，次の文法規則（26)，(27）を与える。

v(A, C, V) ⟶ volt(A, C, V). (26)

v(A, C, V+W) ⟶ volt(A, B, V), v(B, C, W). (27)

これらの文法規則は次の確定節に変換される。

subset(v(A, C, V), S0, S1) :- subset(volt(A, C, V), S0, S1). (26)'

subset(v(A, C, V+W), S0, S2) :- subset(volt(A, B, V), S0, S1),
subset(v(B, C, W), S1, S2).(27)'

節点 $1 と $4 間の電圧Xを求める問題は次のように，文 (24) の中に非終端記号 v($1, $4, X) を同定する問題に帰着できる。

?- vData(VD),
subset(v($1, $4, X), VD, _).

X = 20 + (-15+8)

ここで，変数Xの値は終端記号や非終端記号の同定が行われた軌跡を保持しており，この値から構文解析木を得ることができる(5図)。

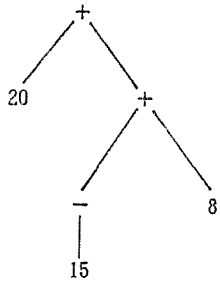


図5 構文解析木

5. 3 適用範囲

通常，論理プログラムはファクト {F_1, F_2, ..., F_n} とルール {R_1, R_2, ..., R_m} と呼ばれる二種類の確定節の有限集合からなっており，いずれも公理として見ることができる。プログラム上の計算はそれらの公理から後向き推論により定理を導く過程である。5．1節のループの問題は定理を導く上で同じファクトを多重に使用することに起因していた。なぜなら，定理を導く計算は同一公理の多重使用を認めていることによる。5．2節においてフ

クト $\{F_1, F_2, \ldots, F_n\}$ は集合型言語の一つの文として仮定された。一方，ルール $\{R_1, R_2, \ldots, R_m\}$ は文法規則に変換された。電圧導出の問題は一般化された構文解析の問題として解くことができた。文脈自由型の言語において，文中の各々の終端記号は非終端記号への還元に一度しか寄与することができない。それゆえに構文解析の問題としてとらえることで，同じデータの多重使用に基づくループの問題が必然的に解消されることになった。

与えられたデータ集合の中に構造を求めるような種類の問題，一筆書きの問題，道順を枚挙する問題など後向き推論ではループに陥り易い問題を DCSG を用いて一般化された構文解析の問題に帰着して扱うことにより，簡単に解決することができる。

6. DCSG の拡張

6. 1 集合の変換

文法規則中の終端記号 [voltage(A, B, V)] は DCSG の文法・確定節変換プロシージャにより member (voltage (A, B, V), S0, S1) に変換された。対象となる集合が S0 に代入されると，その集合から要素 voltage (A, B, V) が除去され，残りの部分が変数 S1 から得られた。そこで，文法規則中の終端記号は対象となる集合から特定の要素を除去し，新しい集合を作る関数として見ることができる(図 6)。

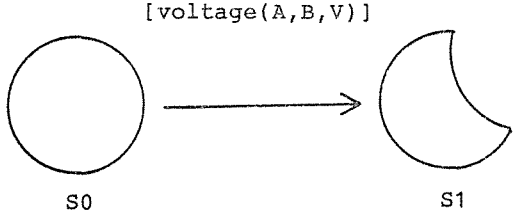


図 6　終端記号による集合の要素除去

一方，文法規則中の非終端記号は，最終的に終端記号の組み合わせとして構成されるから，対象集合から部分集合を除去して新しい集合を作る関数として見ることができる。それゆえに，文法規則は集合の変換を定義するもの

として見ることができる。

ここで，集合の変換の逆変換を考えよう。終端記号による要素除去変換の逆変換はその要素を対象集合に加え新しい集合を作ることである。非終端記号による部分集合の除去変換の逆変換はその非終端記号から導かれる終端記号の集合を対象集合に加えることである。逆変換を実現するための最も簡単な方法は述語 member と subset において入出力の関係を取り替えることである。この方法を終端記号 "m" に対して次のように適用すると，

?- member(m, OUT, [a, b, c]).

集合の表現にリストを用いるために，4つのケース OUT = [m, a, b, c]; [a, m, b, c]; [a, b, m, c,]; [a, b, c, m] が生じる。

この方法を非終端記号に適用する際には別の問題点が生じる。これまで定義してきた文法規則は文の生成ではなく，解析に用いることを想定して定義したために，そのまま逆変換用の規則として用いると，(11) の中点Bのように値が定まらずに変数のまま残るようなことが起こる。

6. 2 add オペレータ

前節の逆変換の方法には問題点があるので，終端記号による集合変換の逆変換だけを独立に考える。次の文法規則のように，

$A \longrightarrow B_1, \ldots, B_{i-1}, \text{add } [B_i], B_{i+1}, \ldots, B_n.$

逆変換のためのオペレータ "add" の付加された終端記号 "add [B_i]" は式

$S_i = [B_i \mid S_{i-1}]$

に変換される。すなわち，オペレータ add はそれが付加された記号を対象集合 S_{i-1} の中に加え，新しい集合 S_i を作る。

オペレータ add を含む文法規則を用いて定義される非終端記号は，もはや対象集合の部分集合に与えた名前として見ることができない。そこで文法・確定節変換に述語 subset を使うことが不適当なので，拡張 DCSG では述語 subset の代わりに述語 convert を使うことにする。

7. 拡張 DCSG の応用

7. 1 事象・状態変化型の問題

集合型言語の確定節文法は文法規則を集合の変換規則として見ることにより，文法規則のシンタックスが拡張でき，集合の変換の逆変換が行えるようになった。すなわち，文法規則は集合の生成あるいは解析を定義するだけでなく，自由に集合の変換を定義することができる。したがって，単にデータ集合の中に構造を見出すような構文解析型の問題だけでなく，事象・状態変化型の問題にも効果的に適用することができる。拡張 DCSG の能力を示すために「人喰い土人と宣教師」の問題が集合変換の問題として，拡張 DCSG の文法規則を用いて形式化できることを示そう。

［人喰い土人と宣教師］

3人の人喰い土人と3人の宣教師が川を渡ろうとしている。川には2人乗りのボートが1艘ある。宣教師の数が土人よりも多いか等しい場合には土人はおとなしくしているが，土人の数が多くなると宣教師を食べてしまう。宣教師が喰われずに無事に6人が川を渡るにはどうすれば良いか。

7. 2 集合変換の文法規則

川を渡る前はこちら側の岸に3人の宣教師（missionaries）と3人の人喰い土人（cannibals）がいる。この状態を集合 {m, m, m, c, c, c} で表す。川を全員が渡り終え，こちら側の岸に一人もいなくなった状態を空集合 { } で表す。この問題は集合 {m, m, m, c, c, c} を，幾つかの拘束条件を満たしながら，空集合に変換する集合の変換が定義できれば良いのである。

ここで対象となる集合はこちら岸の状態を表している。この状態の変化はボートがこちら岸を離れるときと，こちら岸に到着するときにおこる。こちら岸を離れる事象に基づく状態の変化は次の3個の文法規則で表される。

f (mc) ⟶ [m], [c], fcd. (28)

f (mm) ⟶ [m], [m], fcd. (29)

f (m) ⟶ [m], fcd. (30)

非終端記号 f (mc) は宣教師 m と土人 c がボートに乗り，こちら岸を離

れる事象を表す。この事象は対象集合から要素 m と c を除去し，新しい集合を作る。同様に f (mm) は宣教師 m が 2 人ボートに乗り，こちら岸を離れる事象を表す。f (m) は宣教師 m が 1 人ボートに乗りこちら岸を離れる事象を表す。土人だけが単独で川を渡ることはないことにする。 fcd はこちら岸において宣教師が喰われない条件を表す非終端記号で，次の文法規則で定義される。

fcd ⟶ not gt(c, m);

not [m]. (31)

gt (X, Y) ⟶ [X], [Y], gt(X, Y). (32)

gt (X, Y) ⟶ [X], not [Y]. (33)

条件 fcd は二つの選言の条件より定義される。第 1 の条件 not gt(c, m) はこちら岸において土人の数が宣教師の数よりも多くないことを表す。第 2 の条件 not [m] はこちら岸に喰われる宣教師が一人もいないことを表している。

(32) と (33) で定義される非終端記号 gt (X, Y) は集合における 2 種類の要素 X, Y の数の比較を行うことができる。非終端記号 gt(X, Y) を対象集合中に同定することは，規則 (32) により，集合の要素 X と Y をそれぞれ 1 個だけ除去した集合の中に非終端記号 gt(X, Y) を同定することである。このプロセスを繰り返し，やがて，X か Y かのどちらか 1 個でも要素除去が行えなくなると規則 (33) が適用される。規則 (33) は集合中に要素 X が存在し，要素 Y が存在しないことを要求している。従って，対象集合中に非終端記号 gt (X, Y) が同定できるのは対象集合中の要素 X の数が要素 Y の数を越えたときである。

こちら岸にボートが到着したときの状態の変化はボートが出発したときと同様に，次の 3 個の規則で表すことができる。ここでも土人は単独でボートに乗ることはないと仮定している。

b (m) ⟶ add [m], bcd. (34)

b (mm) ⟶ add [m], add [m], bcd. (35)

b (mc) ⟶ add [m], add [c], bcd. (36)

ここで，bcd はボートがこちら岸に到着しているときに，向こう岸において宣教師が喰われない条件を表している。bcd は次のように定義される。

bcd ⟶ not gt (m, c);
test nm (3, m). (37)

nm (N, X) ⟶ [X],
nm(K, X),
"N is K+1". (38)

nm (0, X) ⟶ not [X]. (39)

条件 bcd は二つの選言の条件より定義される。条件 not gt (m, c) はこちら岸において宣教師の数が土人よりも多くないこと，すなわち，向こう岸において宣教師が喰われない条件となっている。条件 test nm (3, m) はこちら岸に宣教師が3人いること，すなわち，向こう岸に喰われる宣教師が一人もいないことを表している。(38) において引用符号で囲まれた部分は DCSG の文法・確定節変換プロシージャの操作を受けずにそのまま確定節の一部に組込まれる。

非終端記号 nm(N, X) を対象集合の中に同定すると，その集合中の特定の要素Xの数Nを調べることができる。規則 (38) は非終端記号 nm(N, X) を同定するために，対象集合に含まれる要素Xを除去し，残りの集合中に非終端記号 nm(K, X) を同定し，得られる要素Xの数Kに1を加えて，元の集合中の要素Xの数とすることを定義している。規則 (39) は集合中に要素Xが存在しないときに，要素の数を0にすることを定義している。

ボートが何度か往復する状態を，出発記号として次の規則で定義することができる。

s ([f(X)]) ⟵ f(X). (40)

s ([f(Z), b (Y) | X]) ⟶ s (X), b (Y),
"not X = [f (Y) | _]",
f (Z),

"not Y = Z". (41)

(40) はこちらの岸から向こう岸へ1回だけ行く事象 f(X) を定義している。(41)は向こう岸へ何度かいった後 s(X), こちらへ戻り b(Y), さらに向こう岸へ行った状態 f(Z) を定義している。条件 not X = [f(Y) | _] は何度か向こう岸に行った後，戻る場合に最後に行ったときと全く同じ構成の人員をボートに乗せてはならないことを表している。条件 not Y=Z は再度，向こう岸に行くときに今，戻って来たときと全く同じ構成の人員をボートに乗せてはならないことを表している。

これらの規則を用いて s(X) を出発記号として次のように集合の変換を実行すると，

?- convert (s(X), [m, m, m, c, c, c], []). (42)

ボートにどのように乗れば良いか，解が変数の値Xから得られる。しかし，これらの規則は計算の手続きとして見ると冗長な部分を多分に含んでおり，そのまま用いたのでは極めて能率が悪い。

能率を落としている原因はリストの要素を調べる際に，調べる順序を入れ替えるだけの無用なバックトラックを行う点である。そこで，カット記号[11]を挿入し無用なバックトラックを制限する。

ボートがこちら岸をはなれる事象を表す規則(28)—(30)は次のように規則を二段階に分けてカット記号を挿入する。

f(mc) ⟶ fmc. (28)'

f(mm) ⟶ fmm. (29)'

f(m) ⟶ fm. (30)'

fmc ⟶ [m], [c], "!", fcd. (28)''

fmm ⟶ [m], [m], "!". fcd. (29)''

fm ⟶ [m], "!", fcd. (30)''

集合中の特定の二種類の要素の数の比較を行う述語の定義 (32), (33) は (32)', (33)' に改める。

gt(X, Y) ⟶ [X], [Y], "!", gt(X, Y). (32)'

```
gt(X, Y)[X], "!", not [Y].                                    (33)'
```

集合中の特定の要素の数を求める述語の定義 (38) は (38)' に改める。

```
nm(N, X) ⟶ [X], "!",
                nm(K, X),
                "N is K+1".                                   (38)'
```

これらのカット記号を導入した規則を用いて，集合の変換 s(X) を実行すると，変数Xからどのように川を渡れば良いかの解を能率よく求めることができる。

```
?- T0 is cputime,
   convert(s(x), [m, m, m, c, c, c], []),
   T is cputime - T0.
X = [f(mc), b(m), g(mc), b(m), f(mm), b(mc), f(mm), b(m),
     f(mc), b(m), f(mc)]
T = 0.75 (sec)
```

解の先頭は最も最後に渡ったボートの状態を表す。すなわち，解の末尾の f(mc) が最初に宣教師 m と土人 c が渡ったことを示し，次に b(m) が宣教師 m が一人で戻ってきたことを示し，次いで，f(mc) が宣教師と土人が渡ったことを示し，ついで宣教師が戻り，次に宣教師が2人で渡り，土人を一人連れて戻り，…… という具合に渡ったことを示している。

8. おわりに

日本語文は文節以下の単位では語順が重要になり，一方，述部の格としての文節の順序は比較的ゆるやかである。この日本語文の性質をうまく取り扱うことを目指して確定節文法の拡張を行っている。ここで開発した集合型言語の確定節文法 DCSG は語順を全く持たない言語を想定しているので，ただちに日本語文法の記述に応用することはできない。集合型言語と見なせるような自然言語が存在するかどうかは不明であるが，一般にデータの集合が構造を持つものであれば，データ集合を集合型言語の一つの文として見るこ

とができる。DCSGはデータ集合の中に構造を見出す種類の問題を一般化された構文解析の問題に帰着して効果的に扱うことができる(17)。

文法規則を集合の変換規則としてとらえ，addオペレータを導入した。このオペレータを用いると，下降解析の過程の中で部分的に対象を書き換える上昇解析の操作が行える。解析の対象となる語列の中に特定の終端記号のパタンが現れたとき，addオペレータを用いて対応する非終端記号に書き替える。この操作を繰り返し，次々に上位の非終端記号へ書き替えて出発記号に到達させる（下降型に制御された上昇解析）(18)。確定節文法の下降型の構文解析のメカニズムは左回帰の文法規則に出会うと一般に無限ループに陥るが，addオペレータによる上昇解析を利用してこの問題を避けることができる。

addオペレータは確定節文法の性格を換えるため，非終端記号は部分集合に相当せず，単に集合の変換に与えた名前となった。集合の変換という見方は内容的に異なる種々の規則を統一的に扱うことを可能にしている。「人喰い土人と宣教師」の問題に用いた文法規則はプロダクション・ルールの適用が下降型に制御されたプロダクション・システムとして見ることができ，プロダクションの条件もプロダクション・ルールもそのルールの制御も同一のシンタックスの下に定義された。addオペレータの導入によりDCSGはバックトラックの行えるプロダクション・システムとしての機能を備えたのである。DCSGは構文解析型の問題だけでなく事象・状態変化型の問題にも応用することができるようになった。

参考文献

（1） Clocksin, W, F. and Mellish, C. S.: Programming in Prolog, Springer-Verlag, Berlin (1981).

（2） Dershowitz, N. and Manna, Z.: Proving Termination with Multiset Orderings, CACM. vol. 22, pp. 465-476 (1979).

（3） Dahl, V. and Abramson, H.: On Gapping Grammars, Proc. The 2nd International Logic Programming Conference (1984).

（4） Dahl, V.: More on Gapping Grammars, Proc. FGCS-84. ICOT (1984).

(5) Gazder, et al.: Generlized Phrase Structure Grammar, Basil Blackwell, Oxford (1985).
(6) Greibach, S. A.: A New Normal Form Theorem for Context-Free Phrase Structure Grammars, JACM. vol. 12, pp. 42-52 (1965).
(7) Hopcroft, J. E. and Ullman, J. D.: Formal Languages and their Relation to Automata, Addison-Wesley, Reading, Mass. (1969).
(8) Kowalski, R.: Logic for Problem Solving, North-Holland, Amsterdam (1979).
(9) 松本，田中，平川，三吉，安川，向井，横井：Prolog に埋め込まれボトムアップパーサ：BUP, proc. LPC7-83, ICOT (1983).
(10) McDermott, D.: Duck manual, Dept. of Computer Science, Yale Univ. (1983).
(11) 中島秀之：Prolog，産業図書 (1983).
(12) Nilsson, N. J.: Principle of Artficial Intelligence, Tioga Pub. Palo Alto, CA. (1980).
(13) 西田富士夫：言語情報処理，コロナ社 (1981).
(14) Pereira, F. and Warren, D.: Definite Clause Grammars for Language Analysis, Artificial Intelligence, vol. 13, pp. 231-278 (1980).
(15) Pereira, F., et al.: C-Prolog user's manual, SRI, CA. (nd.)
(16) 首藤公昭：文節構造モデルによる日本語の機械処理に関する研究，福岡大学研究所報，第45号 (1977).
(17) 田中卓史：論理型言語による電子回路の構造解析に関する研究，九州大学工学博士学位論文 (1987).
(18) 田中卓史：集合型言語の確定節文法 DCSG と応用，情報処理学会論文誌，vol. 28, no. 10, pp. 1013-1020 (1987).
(19) 田中穂積：Prolog による自然言語処理技術に関する基礎的研究 [1] 東京工業大学情報工学科，田中研究室研究成果報告 TR-1 (1986).