ICOT Technical Memorandum: TM-0867

TM-0867

確定節文法のための
内部構造変換機能付きパーザと
アンパーザの自動生成方式

大橋恭子，横田かおる，南俊朗，
沢村一，大谷武(富士通)

March, 1990

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

確定節文法のための
内部構造変換機能付きパーザと
アンパーザの自動生成方式

大橋　恭子，横田　かおる　　㈱富士通研究所
南　俊朗，沢村　一，大谷　武　　富士通㈱国際情報社会科学研究所

1. はじめに

今日，計算機科学や人工知能の分野で様々な論理やその応用の研究が盛んに行われている[1]．論理に基づいた問題解決には，論理系の形式化，問題の記述，証明を経て定理を導く過程が含まれている．これらの過程を計算機で支援することによって問題解決や理論の構築を正確かつ容易に行うことが期待される．

我々は，計算機による支援を行う際に，ただ一つの問題領域だけを扱うのではなく，様々な問題領域を扱える汎用の論証支援システムEUODHILOS[2]を開発した．従来の証明チェッカーや証明コンストラクタ（LCF[3]，EKL[4]等）の多くは，論理系に依存した固有の証明チェック機能，証明構成機能の充実を目指している反面，固定した論理系の下での証明だけしか扱えない．EUODHILOSの大きな特徴は，広範な領域における論理を扱うこと，すなわち，汎用性にある．

EUODHILOSでは汎用性を実現するために，ユーザが扱う問題に適した論理系を定義する機能を用意している．論理系の定義をするときには，論理系で扱う言語（以後，論理構文と呼ぶ）を表現することが必要である．論理構文を表現するには，記述力があり，書きやすく，記述量が少ない記法が望まれる．この条件を満たすものとして，本論文ではDCG形式[5]による構文記述を用いる．

構文だけを表したDCG形式で表現されるのは，ユーザへの入力あるいは出力となる文字列，すなわち，外部表現である．しかし，外部表現を記述するだけでは不十分なことがある．それは，外部表現が異なっていても，実は，同じ内容を表していることが起きるからである．例えば，次の二つの論理式は，文字列としては異なるが，論理式としては同じことを表している．

“∀x．A（x）”と“∀x．（A（x））”

ユーザとしては外部表現の方が扱いやすいが，システム側では同じ論理式をただ一つの形式で扱うために，論理式の意味することを表現する内部構造が必要である．

外部表現と内部構造という二つの表現を扱うため，図1に示すような，外部表現を内部構造に変換する内部構造変換機能の付いたパーザ（以後，単にパーザと呼ぶ）と，内部構造から外部表現に変換するアンパーザが必要である．図1からもわかるように，パーザとアンパーザは，ユーザとシステムの間をつなぐ役割を持っているので，パーザとアンパーザのプログラムには，以下の二つの条件が要求される．

図1　パーザとアンパーザの関係

① 外部表現を正しく受理し，正しい内部表現に変換する．そして，内部構造を構文的に正しい外部表現に変換する．

② 内部構造を外部表現に変換し，その外部表現を再び内部構造に変換したとき，元の内部構造と変換後の内部構造が等しくなること．ただし，外部表現を内部構造に変換し，再び外部表現に変換したときには，元と変換後が等しくなくてもよい．これは，論理式の表している意味が内部構造では，一意に表されるのに対し，外部表現では，一意に定

まらないからである．

この要求を満たすようにユーザがプログラムを記述することにすると，プログラムの作成に多大の注意を払わなければならず，かつ，記述量も多くなる．その結果として，ユーザの負担が大きくなる．

ユーザへの負担を減少させるためには，論理構文を表現する DCG形式の構文規則からパーザとアンパーザを自動生成することが有効である． DCG形式は，論理構文の外部表現を記述するだけでなく内部構造についても記述できる記法であるが，よりユーザの負担を軽減するために，外部表現のみを記述した DCG形式の構文規則からの自動生成が望まれる．しかし，それだけでは自動生成することができない．

これまでにも構文規則からプログラムを生成するパーザ・ジェネレータやコンパイラ・コンパイラの研究がされてきた．既に，UNIX上で動くYacc[6),7)] という実用レベルのものが開発されている．しかし，これらは，構文規則から構文解析機能だけのパーザを生成することを主眼とし，システム内部で扱う内部構造から文字列へ変換する方式はほとんど議論されていない．また，出力文字列を考慮したシステムとして，Cornell 大学で開発されたSynthesizer Generator[8)] もあるが入力用と出力用の構文を別々に記述するため，両者の整合性にも注意を払わなければならない．これらのジェネレータやコンパイラ・コンパイラは，我々の目的である，表現力があり記述量が少なく，かつ，パーザとアンパーザの自動生成ができることを満足するものではない．

本論文では， DCG形式をベースとし，論理構文を記述するのに十分なDCGo記法と，DCGo記法で記述された構文規則からパーザとアンパーザを自動生成する方式を提案する．なお，この自動生成方式は論理系を扱うために開発されたものであるが，汎用の構文記法である DCG形式ベースとしていることにより，論理に限らず，様々な分野に応用が可能である．以下で述べる方式は，逐次推論マシン PSI上で Prologをベースとしたプログラミング言語である ESP[9)]によって実現されている．

以下，２章でDCGo記法，３章でDCGo記法からパーザとアンパーザを自動生成する方式を与え，４章で本自動生成方式を EUODHILOSの言語系定義部に適用するにあたってより有効に用いるために工夫した点を示す．

2. DCGo記法

本章では，パーザとアンパーザの自動生成を可能とするDCGo記法を与える．まず準備として，本論文中で扱うDCGo記法の基礎となっている DCG形式[5)]を2.1節で，内部構造の形式を2.2節で与える．その後2.3節で，自動生成を実現するために必要な DCG形式の改善点を挙げ，2.4節で2.3節に挙げた点を改善したDCGo記法による構文規則の書き方を与える．

2.1 DCG形式

論理構文の記述に用いる DCG形式は文脈自由文法をベースとしているので，構文規則の書きやすさや読みやすさという特徴を受け継いでいる．一方，文脈自由文法からの拡張として，非終端記号への引数の付加と手続き呼び出しが挙げられる．非終端記号の引数と手続き呼び出しを組み合わせることによって，文脈に依存した情報の処理が可能である．

文脈自由文法では，命題論理や第一階述語論理は定義できるが，内包論理のように文脈に依存した型付きの論理は定義できない．しかし， DCG形式を用いれば，図２に示すように内包論理のような型付きの論理も定義できる．

```
term1(t) --> term1(t),imply,temr2(t);
term1(T) --> term2(T);
imply --> "⊃";
term2(t) --> term2(t),and,term3(t);
term2(T) --> term3(T);
and --> "∧";
  . . . . . .
term5((T1,T2)) --> lambda,variable(T1), ".",term5(T2);
  . . . . . .
variable(T) --> var__sym,":",type(T);
lambda --> "λ";
type(e) --> "e";
type(t) --> "t";
type((T1,T2)) --> "(",type(T1),",",type(T2),")";
var__sym --> "x"|"y"|"p";
p__sym1 --> "f"|"g";
```

図2　内包論理の構文規則（一部分）

2.2　内部構造の形式

論理式には，部分式や部分項などの部分的な表現を組み合わせて構成した式（以下，構成式と呼ぶ）と，それ以上の部分表現に分けられない式（以下，基本式と呼ぶ）がある．我々は，構成式にはそれを構成する部分表現の関係を表す構成子が存在する，という立場をとっている．例えば，構成子∀を使った論理式

∀x．A（x）

は，「A（x）を全称限量した式」と認識される．そのとき，構成式の内部構造を，構成子となる要素を先頭に置き，残りを引数として以下のように表現する．

["∀", "x", ["A", "x"]]

この内部構造を一般的に表すと次のようになる．

［構成子，引数1，・・・・，引数n］

内部構造は，構成子は文字列，引数は内部構造という再帰的な構造となっている．また，基本式は，それを表す文字列を内部構造とする．

この内部構造に関する情報（以後，構造情報と呼ぶ）は，DCG形式の非終端記号に引数を加えることによって表現し，かつ構文規則の間で受け渡すことができる．例えば，型"t"を持つ項の合成を表す，

```
term2(t) --> term2(t),and,term3(t);
```

という構文規則が与えられた場合，その引数の部分に構造情報を加えると，

```
term2( [AND,T2,T3] ,t)  >
        term2(T2,t),and(AND),term3(T3,t);
```

という構文規則となる．この構文規則では，構文規則右辺の非終端記号 term2,and,term3 の構造情報を使って，左辺の非終端記号 term2の内部構造を作成している．

2.3　DCG形式の改善点

ここで，以後の説明に必要な用語を定義する．構文規則を記述するときには，構成子と引数のように，内部構造を構成するのに必要な記号の他に，主に論理式を見やすくするために用いられている記号がある．例えば，構成子lambdaを使った次の構文規則では，

```
term5((T1,T2)) -->
        lambda,variable(T1),".",term5(T2);
```

lambdaが構成子，variable,term5が内部構造の引数，"."は論理式を見やすくするために用いる記号と解釈できる．本論文では，内部構造を構成するのに必須の要素を主記号と呼び，それ以外の記号を補助記号と呼ぶ．また，終端記号一つだけが構文規則の右辺にあら

われる構文規則を辞書型の構文規則，そうでない構文規則を規則型の構文規則という．

DCG形式は，論理系で用いる表現を定義するには十分な記法だが，パーザやアンパーザの自動生成をするには不十分である．パーザやアンパーザの自動生成を考えた場合には，以下の点を満足するように改善しなければならない．

①主記号と補助記号の違いの記述

構文規則右辺の非終端記号や終端記号のうち，何が内部構造の要素として使われるものかを明確にする．

②構成子と引数の違いの記述

主記号の中で，構成子であるものを明確にし，残りを引数として，内部構造を構成する．

③ 演算子と述語関数記号の違いの記述と，演算子の 優先順位の違いを記述できること．

以後，優先順位のため括弧の有無で曖昧さが生じる構成子を演算子と呼び，優先順位による問題が生じない構成子を述語関数記号と呼ぶ．例えば，次の内部構造では，

[“∧”,A,B]

構成子“∧”を演算子として定義すると，

“A∧B”

という文字列になる．また，内部構造

[“∧”, [“⊃”,A,B] ,C]

は，

“A⊃B∧C”

という文字列になる．この文字列で表している論理式は，演算子“∧”と“⊃”の優先順位の解釈によって曖昧になる．演算子の優先順位の違いを記述することによって，適当な括弧を補い，

“(A⊃B)∧C”

と曖昧さのない文字列を生成できるようにする． また，構成子“∧”を述語関数記号として定義すると，次の文字列となる．

“∧”(A,B)

このときには，生成された文字列に，曖昧さが生じない．

2.4 DCGo記法

2.3節では，パーザとアンパーザを自動生成するために， DCG形式を改善すべき点を挙げた．それらの点を改善した記法としてDCGo記法を提案する．DCGo記法は， DCG形式の記法に構成子定義を加えたものである．構成子定義とは，非終端記号や終端記号のうち，構成子として扱う要素を定義するものである．

まず，構成子定義の記法を示す．構成子定義は，次の形式で行う．

with_priority

$O_{11}, O_{12}, \ldots, O_{1n(1)}$;

...

$O_{m1}, O_{m2}, \ldots, O_{mn(m)}$.

without_priority

$P_1, P_2, \ldots, P_t$.

(ただし， O_{ij}は演算子，P_k は述語関数記号)

構成子定義では，構成子を演算子と述語関数記号に分けて記述する．演算子として用いる構成子を予約語 with_priorityの後に優先順位の高い順にセミ・コロンで区切って並べる．優先順位は高い順に1,2,...,m とする．ただし，同じ優先順位を持つ演算子はカンマで区切り，最後の演算子の後にはピリオドを置く．また，述語関数記号を予約語without_priorityの後にカンマで区切って並べる．演算子の場合と同様に，最後の述語関数記号の後にはピリオドを置く．演算子や述語関数記号は，非終端記号と終端記号のどちらで記述してもよい．実例として，内包論理による構成子定義の一部を図3に示す．また，DCGo記法の BNFによる定義を付録に示す．

```
with_priority
        :,:,:
        ( not,lambda,bind_op );
        and,
    without  priority
        p_syml,
```

図3　内包論理の構成子定義（一部分）

以上の記法に従って論理構文が記述できる．ただし，内部構造を正しく構成するための条件がある．その条件とは，

「規則型の構文規則のうち，右辺に主記号が二つ以上上ある場合には，そのうちの一つが構成子であること」である．この制限は，内部構造の構成から明らかなように，構成子がただ一つであるからである．

このようにDCGo記法には記述の際の若干の制限はあるが，論理系の記述には十分であり，かつ，DCG形式の記述力を損なうものではない．

また，DCGo記法は，主記号を非終端記号と構成子，補助記号を構成子でない終端記号と定義することにより，2.3節で挙げた改善点の①を満足し，主記号のうち構成子でないものを引数とすることによって，改善点の②を満足する．また，構成子定義で，演算子と述語関数記号の違いと，演算子の優先順位を記述した．これにより，改善点の③が満足される．以上，構成子定義を導入したことにより，2.3節で挙げた点①～③が改善され，パーザとアンパーザの自動生成が可能となる．

また，DCGo記法には，上記に挙げた本自動生成方式に固有な記述上の制限の他に，一般的な制約がある[9),10)]．次にその制約条件を二つ示す．しかし，これら二つの制約は本質的なものではない．

(a)　サイクル・フリーであること．

これは，構文規則を記述するときに一般的な制限である．サイクル・フリーとは，

S　→　S

や，

S　→　S1

S1　→　S

というような，循環して自分自身を定義している規則がないことである．

(b)　ε規則がないこと．

ε規則とは右辺に終端記号や非終端記号が一つもない規則で，右辺がnull文字列の辞書型の構文規則と等価である．3.2節で議論するように，我々はパーザとしてボトム・アップ・パーザを用いている．このパーザでは，構文解析のときに，文のいたるところで適用可能となり，制限をしないとε規則があることによって膨大な量の処理をしなければなら

ない[10]．対処する手段もあるが，ε規則を使わないほうが望ましい．

以後，本論文では，図2と図3に示した2つの定義を用いながら例を説明する．

3. DCGo記法からのパーザとアンパーザの自動生成方式

本章で与えるDCGo記法からパーザとアンパーザを自動生成する方式の概略を図4に示す．

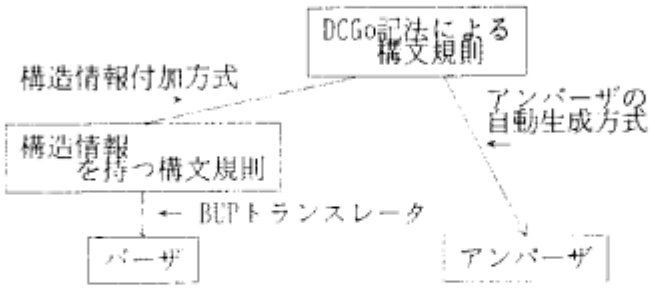


図4　構文規則からのパーザ，アンパーザの生成

この図4に示すように，パーザはDCGo記法で書かれた構文規則から，構造構造情報の付加した構文規則に変換し，それを　BUPトランスレータ[10)-12)]を用いて作成する．3.1節では，構文規則に構造情報を付加する方式を与える．パーザは，与えられた文字列が構文的に正しいか否かを判定するのと同時に，対応する内部構造も生成する．BUPパーザを採用した理由は3.2節で明らかにする．アンパーザは，DCGo記法の構文規則からアンパーズ用データを取り出し，そのデータに基づいて作成する．このアンパーザは，内部構造を構文的に正しい文字列に変換する．アンパーズの方式とその実現方式を3.3節で述べる．

3.1　構文規則への構造情報付加方式

構成子定義を参照しながら，論理式の文字列表現だけを表す構文規則に構造情報を付加する．パーザは，構造情報の付いた構文規則から生成する．ここでは，構文規則をいくつかに分類し，それぞれの構文規則に対して構造情報を付加する方式を述べる．

まず，論理式と構文規則の対応を示す．我々は，論理式を基本式と構成式に分けている．基本式は，辞書型の構文規則で表すことができ，構成式は，右辺に主記号が二つ以上ある規則型の構文規則で表すことができる．また，規則型の構文規則には，次に示すように右辺に非終端記号が一つだけのものもあるが，

```
term1 --> term2;
```

これは，右辺の非終端記号であらわされる構成式を，そのまま左辺の構成式にする場合を示している．

構文規則に構造情報を付加するには，非終端記号に引数を加え，その引数に変数を用いたメタな形式で内部構造を表現する．この構文規則から生成されたパーザで，入力文字列の構文解析が成功すると，変数と文字列がユニファイして内部構造が生成される．ここでは，論理式と構文規則の対応に従い，構文規則を辞書型と規則型に，規則型の構文規則は，主記号の数によりさらに分類して，次に述べるように構造情報付き構文規則へ変換する．

① 辞書型の構文規則

辞書型の構文規則では，基本式を表している．この型の規則では，右辺の終端記号をそのまま内部構造とする．例えば，

```
type(e) --> "e";
```

という構文規則に対しては，右辺の終端記号 "e"を内部構造として左辺の第1引数に加え，

```
type("e",e)    > "e";
```

という構文規則に変換する．

② 規則型の構文規則

規則型の構文規則を，右辺の主記号，すなわち内部構造の作成に必要な構成要素，の個数によって，さらに分類する．

(a)主記号が1個のとき

この形式の構文規則の場合には，構文規則右辺が持つ内部構造を，そのまま左辺の内部構造とする．

```
term2 --> term3;
```

という構文規則ならば，term3の内部構造をそのまま term2の内部構造とするので，

```
term2(A) --> term3(A);
```

という構文規則に変換する．

(b)主記号が2個以上のとき

この場合には2.4節で述べた本自動生成方式に固有の制限により，主記号の中の一つが内部構造の構成子となる．主記号が構成子であるか否かは，構成子定義から得ることができる．それ以外の主記号は，出現順に内部構造の引数とする．次の構文規則では，":"が構成子である（図3の構成子定義より）．

```
variable(T)  -> var_sym,":",type(T);
```

VARで var_sym の内部構造，TYPEでtypeの内部構造を表わすと，

```
variable(["":",VAR,TYPE],T) -->
        var_sym(VAR),":",type(TYPE,T);
```

という構文規則に変換される．

補助記号 "."を含む，

```
term5((T1,T2)) --> lambda,variable(T1),
                   ".",term5(T2);
```

という構文規則の場合には，補助記号を除いて内部構造を作成し，構文規則を変換する．この構文規則での構成子はlambdaである．LAMBDAで構成子lambdaの内部構造，VARで variable，Term5で term5の内部構造を表すと，

```
term5(["LAMBDA,VAR,Term5",(T1,T2)) -->
      lambda(LAMBDA),variable(VAR,T1),
                  ".",term5(Term5,T2);
```

という構文規則になる．内部構造に補助記号を含めないこと以外は，補助記号を含まない構文規則と同じように変換する．

ここで示した変換法を用いることによって，DCGo記法で記述された構文規則は構造情報を持つ通常の DCG形式の構文規則に変換される．

3.2 パーザの自動生成

3.1節で示した変換によって作られた構造情報付き構文規則は，パーザ・ジェネレータによってパーザに変換される．

Prologベースの代表的なパーザとしてトップ・ダウン・パーザの DCG[9),10)]とボトム・

アップ・パーザのBUP[10)-12)]が良く知られている．トップ・ダウン・パーザでは，左再帰的な構文を扱う場合に，無限ループに陥る可能性がある[9),13)]．そこで，パーザの生成にあたっては，左再帰的な構文も扱うことができる BUPを採用した． DCG形式を効率的にBUPパーザへ変換する手法はすでに知られている[10)-12)] ので，この既存の方法を用いることによって，3.1節の方式で変換した構造情報を持つ構文規則から，パーザが生成できる．

図2，3から生成した構造情報付き構文規則を BUPトランスレータによって変換したBUPパーザのプログラムを，図5に示す．

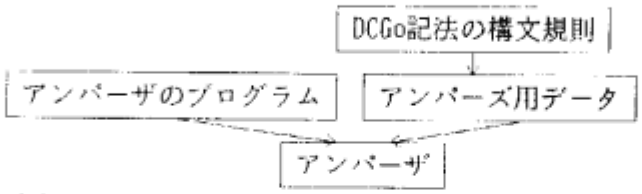
```
parse(G,A,S) :- goal(G,A,S,[]).
goal(G,A,S1,Sn):-
  (dict(N,X,S1,Sn,C),link(N,G),call(C);
   e_rule(N,G,X),S1=S2),
  P=..[N,G,X,A,S2,Sn],call(P).

term1(term1,A,A,S,S).
term1(G,[T1,t],X,S1,Sn):-
  link(term2,G),
  goal(or,[OR],S1,S2),
  goal(term2,[T2,t],S2,S3),
  term2(G,[[OR,T1,T2]],X,S3,Sn).
term2(term2,A,A,S,S).
term2(G,[T2,t],X,S1,Sn):-
  link(term2,G),
  goal(and,[AND],S1,S2),
  goal(term3,[T3,t],S2,S3),
  term2(G,[[AND,T2,T3]],X,S3,Sn).
term3(term3,A,A,S,S).
term3(G,[A,T],X,S1,Sn):-
  link(term2,G),
  term2(G,[A,T],X,S1,Sn).
  . . . . . .
dict(and,["∧"],["∧"|S1],Sn,true).
  . . . . . .
```

図5　内包論理の BUPパーザ（一部分）

3.3　アンパーザの自動生成

パーザで作成した内部構造は，論理式をシステム内部で処理するときの構造である．補助記号は，内部で処理されるときは不要であるため内部構造には含まれていない．しかし，アンパーザによって，文字列表現に変換する際には，補助記号を補わなければならない．

図6には，DCGo記法の構文規則からアンパーザを自動生成する方式の概略を示す．本論文で与えるアンパーザは，どの論理にも共通な方式を用い，アンパーズに必要なデータ（アンパーズ用データ）を参照して文字列を生成する．アンパーズ用データは，DCGo記法の構文規則から作成する．



第6図　DCGo記法の構文規則から，アンパーザを生成する過程

(1)　アンパーズ用データの要素

アンパーズ用データは，構成子を含む規則型のものを使って作成される．

アンパーズ用データは，構成子が優先順位を持つ演算子，すなわち，with_priorityで定義されたものならば，

- 文字列生成用リスト
- 構成子の位置

└優先順位

という3つの要素を持ち，構成子が優先順位を持たない述語関数記号，すなわち，without_priorityで定義されたものであれば，

┌文字列生成用リスト
└構成子の位置

という2つの要素を持つ．

文字列生成用リストは，文字列を形式的に表したもので，構文規則の右辺と同じ数の要素を持つリストで表現される．リストの各要素は構文規則右辺の同じ位置の構文要素と対応する．各要素には対応する構文要素が非終端記号ならば変数，終端記号ならばその文字列を割り当てる．構成子の位置は，構文規則右辺の中での構成子の位置を示すものである．また，優先順位は構成子定義から得ることができる．

次に構文規則からアンパーズ用データを作成する方式を具体的な例を用いて説明する．次の構文規則，

```
term5((T1,T2)) --> lambda,variable(T1),
                                    ".",term5(T2);
```

は，右辺の構文要素が4個で，1，2，4番目の構文要素が非終端記号，3番目の構文要素が終端記号（文字列）である．このような構文規則の場合の文字列生成用リストは，長さ4で1，2，4番目の要素が変数，3番目の要素が文字列 "."となる．図3の構成子定義より，lambdaは優先順位2の構成子である．この構文規則では，lambdaが構文規則右辺の中の1番目にあるので，構成子の位置は1である．これらを組にした，構成子lambdaに対するアンパーズ用データは次のようになる．

構成子lambdaのアンパーズ用データ

┌文字列生成用リスト：「LAMBDA,VAR,".",T5」
├構成子の位置：1
└優先順位：2

(2) アンパーズの方式

本節では，内部構造が基本式の場合と，再帰的な構造の構成式の場合に分けて説明する．構成式の場合には，(1)の方式で作成したアンパーズ用データを用いる．アンパーズ用データは，構成子をキーにして取り出す．① 基本式の場合

基本式の内部構造は，それを表す文字列自身であった．アンパーズの場合も，内部構造の文字列を生成文字列とする．

② 構成式の場合

構成式の内部構造は引数が再帰的な構造をしている．構成式のアンパーズを行うときには，あらかじめ再帰的な処理により引数を文字列に変換する．

引数が構成式の場合には，生成される文字列にあいまいさをなくすために，括弧を付けなければならない場合がある．構成子が述語関数記号のときは，括弧付けの処理は必要ないが，構成子が演算子のときは，引数の構成式の演算子（子演算子）の優先順位と，元の内部構造の演算子（親演算子）の優先順位によって引数を表す文字列に括弧を付ける．括弧を付けるか否かは，親演算子の優先順位が高い場合と，子演算子の優先順位の高い場合で分けられる．

(a) 親演算子の優先順位＜子演算子の優先順位の場合　次の内部構造の引数は，

［“λ”,“x:t”,“x:t”］

再帰的な処理を用いて“x:t”という文字列にアンパーズされている．この内部構造では引数の演算子“:”（図3の構成子定義より，優先順位1）が子演算子となる．また，親演算子はlambdaで優先順位が2である．親演算子のアンパーズ用データは，(1)で与えたとおりである．この二つの演算子のように，

親演算子の優先順位＜子演算子の優先順位

の場合には，引数に括弧をつけない．

(b) 子演算子の優先順位≦親演算子の優先順位の場合　次の内部構造の第二引数の構成子（子演算子）は，　［“λ”,“x:t”,“x:t∧y:t”］

優先順位3の andで，親演算子は優先順位2のlambdaである．このように，

andの優先順位≦lambdaの優先順位

すなわち，

子演算子の優先順位≦親演算子の優先順位

の場合には，引数に括弧を付ける．

最後に，内部構造と文字列生成用リストの各要素をユニファイさせる．内部構造の最初の要素である構成子は，アンパーズ用データの構成子の位置を参照し，文字列生成用リストの該当位置にある要素とユニファイさせる．引数は文字列生成用リストの中の対応する変数とユニファイさせる．(a)で例に挙げた内部構造

［“λ”,“x:t”,“x:t”］

は，図7に示すように文字列生成用リストとユニファイし，文字列

“λx:t.x:t”

が生成される．また，(b)で例に挙げた内部構造

［“λ”,“x:t”,“x:t∧y:t”］

は，図8に示すように文字列生成用リストとユニファイし，文字列

“λx:t.(x:t∧y:t)”

が生成される．

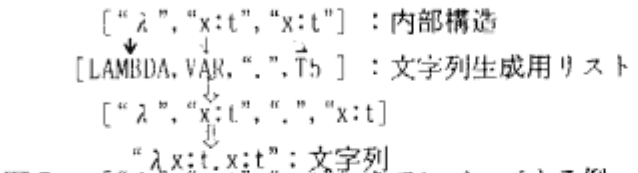
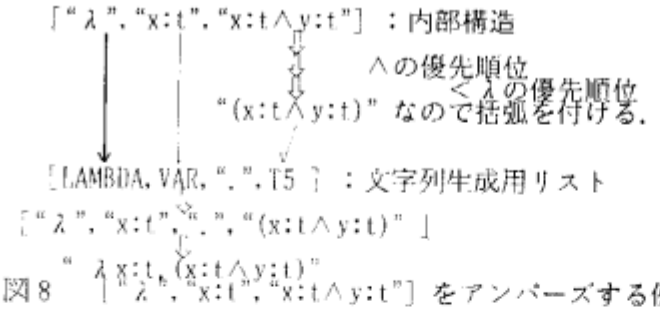


図7　［“λ”,“x:t”,“x:t”］をアンパーズする例



4. 自動生成方式適用上の工夫

本論文に述べた自動生成方式を EUODHILOSの言語系定義部に適用した．そこでは，ユーザがエディタからDCGo記法で記述される論理構文の入力を行う．本自動生成方式をEUODHILOSで有効に用いるためには，単に，システムの中に加えるだけでなく，それを有効に使うための工夫が必要である． EUODHILOSの場合は，構文規則のテスト機能を加えた．一般に，一度だけで意図した構文規則を記述するのは非常に難しいが，テスト機能により誤った構文規則を直ちに訂正することができる．その結果，論理構文の正しい記述を素早く完成させることができる． EUODHILOSの場合には，テスト機能として，構文上のテストの他に，ユーザの意図した内部構造が作成できるかどうか，すなわち，意味上のテストも必要である．

構文上のテスト機能として，図9に示したような構文チェッカーを用意した．そのチェッカーの中では，文字列を入力すると，その文字列が構文規則に当てはまるか否かを示す．

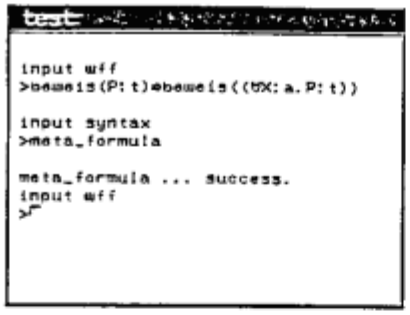


図9 テスト機能（構文チェッカー）

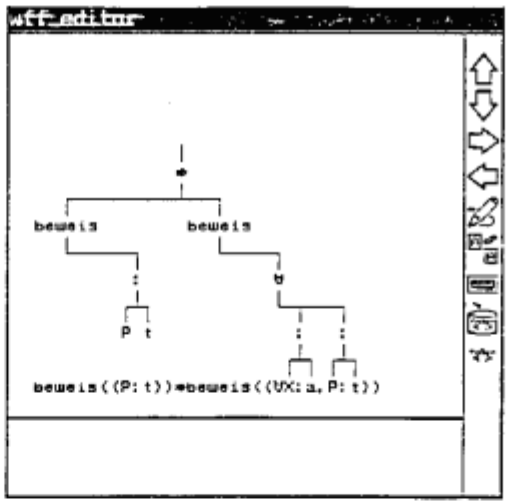


図10 テスト機能（論理式エディタ）

論理式の内部構造は，論理式エディタによってテストする．ここで使われる構造は，DCGo記法の構文規則に基づいているので，構文の記述が，自分の意図したものになっているかを確かめられる．図10に論理式エディタによる論理式の表示を示す．

ユーザは，構文チェッカーにより構文規則の構文的な妥当性，論理式エディタにより構文規則の意味的な妥当性を確認することができる．これらのユーティリティを用いることにより，どの論理系でも，DCGo記法により論理構文を記述する労力が著しく軽減した．

5. まとめ

本論文ではDCGo記法，すなわち，DCG形式に基づいた構文規則と構成子定義から成る論理構文の記述より，パーザ，アンパーザを自動生成する方式を与えた．これらの自動生成により，誤ったプログラムの作成，およびパーザとアンパーザ間の不整合が防止でき，ユーザの負担を軽減することができる．

また，本論文で与えた自動生成方式と3章で示したツールを用いて，論理系の定義を行った．今までに，第一階述語論理，直観主義論理，内包論理，様相命題論理，第2階述語論理，Hoare論理等を定義した．第一階述語論理の論理構文と内包論理の論理構文を定義したものを図11と図12に示す．これらの様々な論理系の定義が可能であることにより，本論文で与えた自動生成方式の一般性を示すことができる．

SYNTAX = predicate

```
formula --> formula,equivalence,formula1;
formula --> formula1;

formula1 --> formula1,imply,formula2;
formula1 --> formula2;

formula2 --> formula2,or,formula3;
formula2 --> formula3;

formula3 --> formula3,and,formula4;
formula3 --> formula4;

formula4 --> "(",formula,")";
formula4 --> not,formula4;
formula4 --> bind_op,variable,formula4;
formula4 --> basic_formula;

basic_formula --> predicate_symbol1,"(",term,")";
basic_formula --> predicate_symbol2,"(",term,",",!
                  term,")";
basic_formula --> predicate_symbol3,"(",term,",",!
                  term,",",term,")";
```

(A)

SYNTAX = predicate

```
with_priority
  be_fond_of;
  equal;
  ( not,bind_op );
  and;
  or;
  imply;
  equivalence;
without_priority
  predicate_symbol1,predicate_symbol2,
  predicate_symbol3.
```

(B)

図11 第一階述語論理の論理構文

SYNTAX = int_wt_type

```
meta_formula --> pred_const,"(",term(_),")";
pred_const --> "beweis";
meta_formula --> meta_formula,meta_imply,
                 meta_formula;
meta_imply --> "e";
meta_formula --> meta_term(_);
variable(T) --> var_sym,colon,type(T)
              ! meta_variable,colon,type(T);
constant(t) --> truth_value,colon,type(t);
constant(T) --> const_sym,colon,type(T);
type((T1,T2)) --> type(T1),comma,type(T2);
type((s,T)) --> s,comma,type(T);
term(t) --> term(t),imply,term1(t);
term(T) --> term1(T);
term1(t) --> term1(t),or,term2(t);
term1(T) --> term2(T);
term2(t) --> term2(t),and,term3(t);
term2(T) --> term3(T);
term3(t) --> term3(T),equality,term7(T);
term3(T) --> term7(T);
term7(T2) --> term7((s,(T1,T2))),left_brace,term(T1),!
              right_brace;
term7(T) --> term4(T);
term4(t) --> bind_op,variable(T),".",term5(t);
```

(A)

SYNTAX = int_wt_type

```
with_priority
    ( colon,comma );
    ( intension , extension , necessary , possible );
    ( apply , not );
    left_brace;
    ( lambda ,bind_op );
    equality;
    and;
    or;
    ( meta_imply , imply );
without_priority
    pred_const.
```

(B)

図12 内包論理の論理構文

今後は，演算子の優先順位の定義を構成子定義と構文規則で二重に行っている点を改良して行きたい．

なお，本研究は，第5世代コンピュータ開発の一環として，ICOTの委託によって行ったものである．

＜参考文献＞

1) Turner,R.:Logics for Artificial Intelligence, Ellis Horwood Limited(1984).

2) 南，沢村，佐藤，上屋，菱：論証支援システム： 論理モデル構築のための支援ツール，Proceedings of Logic Programming Conference '88,ICOT(1988).

3) Gordon,M.J.,Milner,A.J. and Wadsworth,C.P.: Edinburgh LCF,LNCS,Vol.78,Springer(1978).

4) Ketonen,J. and Weening,J.S.:EKL-An Interactive Proof Checker,User's Reference Manual, Dept. of Computer Science,Stanford University(1984).

5) Pereira,F.C.N. and Warren,D.H.D. : Definite Clause grammars for language analysis -A survey of the formalism and a comparison with augumented transition networks, Artificial Intelligence, Vol.13,pp.231-278,(1980).

6) Johnson,S. : Yacc:Yet Another Compiler Compiler, Computing Science Technical Report, No.32, AT&T Bell Laboratories(1975).

7) Lesk,M.E. : Lex-A Lexical Analysis Generator, Computing Science Technical Report, No.39, AT&T Bell Laboratories(1975).

8) Reps,T : The Synthesizer Generator Reference Manual,Dept.of Computer Science, Cornell Univ. (1985).

9) Chikayama,Takashi : ESP Reference Manual,ICOT Technical Report, TR-044,ICOT(1984).

10) Clocksin,C.F. and Merish,C.S. :Programming in Prolog,Springer-Velag(1981).

11) 溝口文雄監修：Prologとその応用２，総研出版， (1985).

12) 松本裕治，田中穂積他：Prologに埋め込まれたボトム・アップ・パーザ:BUP, Proc. of The Logic Programming Conf.'83,3.1 (1983).

13) Matsumoto,Yuji and Tanaka,Hozumi et al. : BUP:A Bottom Up Parser Embedded in Prolog ,New Generation Computing Vol.1, No.2,pp145-158(1983).

＜付録＞

構文定義と構成子定義の定義を，以下の形式のＢＮＦで定義する．

ＢＮＦの形式の定義

・"::="の左側が，その右側で定義されていることを示す．ただし，"｜"で区切られたものは，そのうちのいずれか一つの選択をを示している．

・"X"は一つの終端記号Xを示す．ただし，ダブル・クウォート（"）が２個連続することにより，一つのダブル・クウォートを示す．すなわち，""""は一つのダブル・クウォートを意味する．

・｛X｝は，０回以上の任意の回数，Xが繰り返してよいことを示す．

・［X］は，Xが一つ存在するか，あるいは省略して空であることを示す．すなわち，随意選択を示す．

構文規則と記号規則のＢＮＦによる定義

＊＊＊＊　構文規則定義　＊＊＊＊

＜構文規則・構成子定義＞::= ＜構文規則定義＞＜構成子定義＞

＊＊＊＊　構文規則定義に関する定義　＊＊＊＊

＜構文規則定義＞::= ＜構文規則＞｛";"＜構文規則＞｝"."

＜構文規則＞::= ＜非終端記号＞"-->"＜右辺＞

＜右辺＞::= ＜節＞｛","＜構文要素＞｝｜＜右辺＞｛"｜"＜右辺＞｝

＜節＞::=＜非終端記号＞｜＜終端記号＞｜＜演算子＞｜＜述語記号＞

＜構文要素＞::= ＜節＞｜＜ＣＡＬＬ節＞

＜非終端記号＞::- ＜アトム＞｜＜複合項＞

＜終端記号＞::- ＜文字ストリング＞

＜ＣＡＬＬ節＞::= "call("＜ＥＳＰメソッド・リスト＞")"

＜ＥＳＰメソッド・リスト＞::= ＜ＥＳＰメソッド＞｛","＜ＥＳＰメソッド＞｝

＜ＥＳＰメソッド＞::= ＜複合項＞｜＜演算子適用項＞

＊＊＊＊　構成子定義　＊＊＊＊

＜構成子定義＞::= "with_ priority" ＜演算子定義＞

"without_ priority"＜述語関数記号定義＞"."

＜演算子定義＞::= ＜同一優先順位の演算子＞｛";"＜同一優先順位の演算子＞｝";"

＜同一優先順位の演算子＞::= "(" ＜演算子＞｛","＜演算子＞｝")"

＜演算子＞::= ＜非終端記号＞｜＜終端記号＞

＜述語関数記号定義＞::- ＜述語関数記号＞｛","＜述語関数記号＞｝

＜述語関数記号＞::- ＜非終端記号＞｜＜終端記号＞

注）　＜アトム＞，＜複合項＞，＜文字ストリング＞，＜演算子適用項＞の定義の詳細はＥＳＰの文法書[9]を参照して下さい．