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†††

本論文では，DCG の形式をベースとした構文記述法である DCGo 記法と，その記法で書かれた構文規則から内部構造変換機能付きパーザとアンパーザを自動生成する方式を提案する．我々は一般ユーザを対象としているので構文規則に，記述力，記述の容易さ，記述量の少なさ，という評価基準を設け，それを満足する構文規則からパーザとアンパーザの自動生成することを目的とした．DCG 形式は上記の評価基準を満たしながらパーザとアンパーザを自動生成することはできない．ユーザの負担を軽減するために，より簡単な構文記述法からの自動生成が望まれる．本方式では自動生成を行うために，DCG 形式に構成子定義を加えた構文記述法である，DCGo 記法を与える．DCGo 記法で記述された構文規則から内部構造変換機能付きパーザとアンパーザを自動生成する．この自動生成方式は，汎用の構文記述言語である DCG 形式をベースとしているので，扱う言語の定まっていないシステムにおける言語処理系の作成に有効である．

1. はじめに

今日，計算機科学や人工知能の分野において様々な問題領域の研究が行われている．これらの様々な問題を計算機上のシステムで処理するためには，何らかの言語体系によって記述する必要がある．各々の問題は適切な言語体系で表現されることが望ましいが，それは問題領域ごとに異なる．

様々な対象問題を処理する汎用のシステムでは，これらの様々な問題を一つのシステムで記述し分けなければならない．例えば，汎用の証明支援システムでは，様々な論理式を扱う必要がある．同様の問題は，数式や形式言語で問題を記述する場合にも生じる．汎用のシステムで一つの言語体系を定めて，その範囲内で問題を表現しようとすると，人間にとって書きやすく，読みやすい問題領域特有の慣用的表現を使えない．それが，自然な思考の妨げとなる．汎用システムの問題記述能力を向上させるためには，ユーザが言語を定義できなければならない．

システムとユーザのインタフェースに用いられる文字列は，通常，ある種の構造を持っている．そのような表現の例として，論理式やプログラミング言語が挙げられる．このことを考慮すると，システム内部では表現を構造体で処理するほうがよい．また，次の論理式の場合，

"$\forall x.A(x)$" と "$\forall x.(A(x))$"

は文字列としては異なっているが，同じ内容を表していると考えられるので，これらを等価なものとして扱う必要がある．そのためには，インタフェースに用いる表現（以後，外部表現と呼ぶ）のほかに，システム内部での処理に用いられる構造表現（以後，内部構造と呼ぶ）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に**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つ．もし，ユーザがパーザやアンパーザのプログラムを書くとすれば，以下の二つの条件を満たさなければならない．

① パーザの条件

言語の定義に合った外部表現だけを受理し，その外部表現が表す内容に対応した内部構造に変換する．

② アンパーザの条件

内部構造を，それが表している内容を表現し，かつ言語の定義に合った外部表現に変換する．そのとき，外部表現を再びパーザによって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が保証されなければならない．ただし，外部表現を内部構造に変換し，再び

† An Automatic Generation of a Parser and an Unparser in the Definite Clause Grammar by KYOKO OHASHI, KAORU YOKOTA (Fujitsu Laboratories Ltd.), TOSHIRO MINAMI, HAJIME SAWAMURA and TAKESHI OHTANI (Internatinal Institute for Advanced Study of Social Information Science, Fujitsu Ltd.).

†† (株)富士通研究所
††† 富士通(株)国際情報社会科学研究所

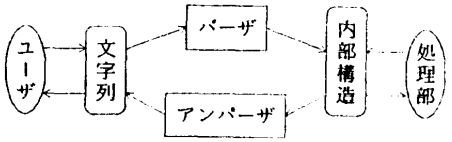

図 1 パーザとアンパーザの関係
Fig. 1 Relation between parser and unparser.

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．

この条件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な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．ユーザの負担を軽減するためには，ユーザが構文規則を用いて言語を定義し，その定義からパーザ，アンパーザを自動生成することが有効である．

我々は，構文規則に対して，記述力，書きやすさ，記述量の少なさ，という評価基準を設けた．なぜならば，対象とするユーザとして構文記述の専門家ではなく，一般ユーザを仮定しているからである．この評価基準を満足した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望ましい．構文記述法の一つである DCG（確定節文法）形式[1]は文脈自由文法を基にした記法で構文規則を書きやすい，という利点がある．DCG形式は，パーザの生成方法も既に知られており[2),3)]，それに様々な機能を追加することも可能であるが，その場合プログラミングに近い詳細な記述を行わなければならない．その結果，

- プログラミングや内部構造に関する知識が必要とされるため，書きやすさが減少する．
- 記述量が増える．

という問題が生じ，構文規則の評価基準を満足しなくなるので，DCG形式は本自動生成方式の構文記述方式としては不適当である．

我々は本論文で，DCG形式の欠点を克服するための記法である DCGo 記法と，その記法からパーザ，アンパーザを自動生成する方式を提案する．DCGo 記法は DCG形式に構成子定義という簡単な定義を加えたものである．DCG形式による構文記述と構成子定義から内部構造変換機能を付けた構文規則を作成し，それを基にパーザを生成する．アンパーザの生成に関しては，これまでに知られた方法がないので，構文規則に依存しないアンパーズのアルゴリズムを考案した．アンパーザは，そのアルゴリズム，および，構文規則と構成子定義から抽出したデータから作成する．

以下，2章で DCGo 記法，3章で DCGo 記法で書か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与え，4章で本自動生成方式の適用例を示す．

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形式を 2.1 節で示し，本論文中で用いる内部構造の形式を 2.2 節で与える．2.3 節では，自動生成を実現するためには必要だが，DCG形式では表現できない点を明らかにする．その後 2.4 節で 2.3 節に挙げた点を表現することを可能にした DCGo 記法による構文規則の書き方を与える．

2.1 DCG 形式

構文規則の記述に用いる 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，構文規則間での（文脈に依存した情報も含めた）データの受け渡しや，それに対する処理が可能である．

文脈自由文法では，命題論理のように構文規則間でのデータの受け渡しが不要なものは定義できるが，内包論理における型のようにデータの受け渡しによるチェックを必要とするものは定義できない．しかし，DCG形式を用いれば，**図 2** に示すようにデータの受け渡しを必要とするものも定義可能である．例えば，

```
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_syml --> "f" | "g";
```

図 2 内包論理の構文規則（一部分）
Fig. 2 Syntax rule of intensional logic (part).

図2の中の構文規則

term5((T1, T2))--〉

lambda, variable (T1), ".", term5(T2);

は，非終端記号の引数 (T1, T2)，T1, T2 でデータの受け渡しを行い，左辺の term5 の引数の値と右辺の variable と term5 の引数の値の関係を示している．

2.2 内部構造の形式

構造表現には，部分的な表現を組み合わせて構成したもの（以下，構成表現と呼ぶ）と，それ以上の部分表現に分けられないもの（以下，基本表現と呼ぶ）がある．我々は，構成表現にはそれを構成してい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

$\forall x.A(x)$

は，「『x は A である』を全称限量した式」と認識される．そのとき，構成表現の内部構造を，構成子となる要素を先頭に置き，それ以外の要素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]

構成表現の内部構造における引数もまた内部構造であり，再帰的な構造となっている．また，基本表現は，その外部表現である文字列を用いて内部構造とする．構成子は，それ以上部分表現に分けられないので，基本表現である．

この内部構造に関する情報（以下，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 自動生成実現のための必要事項

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非終端記号や終端記号（以後，主記号と呼ぶ）のほかに，表現を見やすくすることを主な目的としている終端記号（以後，補助記号と呼ぶ）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，". " は補助記号と解釈できる．

パーザやアンパーザの自動生成をするには以下の①～③を記述しなければならない．DCG形式は，構文記述には十分な記法だが，以下の違いを記述できないため，自動生成のための記法としては不十分である．

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，しばしば使われる構成子で前置，中置，後置等の使い方をする構成子を演算子と呼ぶ．演算子が二つ以上ある表現では，括弧の有無や解釈によって意味に曖昧さが生じるため，演算子には優先順位が必要である．例えば，構成子"∧"，"⊃"を演算子として定義したとき，内部構造

["∧", A, B]

は構文規則に従い，次の文字列となる．

"$A \wedge B$"

二つ以上の演算子が含まれる

["∧", ["⊃", A, B], C] ……(*)

のような内部構造（*）は，同様に

"$A \supset B \wedge C$"

となるが，この文字列で表現されていることは，演算子"∧"と"⊃"の優先順位の解釈によって曖昧になる．そのため，演算子の優先順位を明らかにし，適当な括弧を補い，

"$(A \supset B) \wedge C$"

として曖昧さのない文字列を生成できるようにしなければならない．

逆に，一般の述語記号や関数記号を表すときのように"$p(x)$"といった使い方をする構成子を述語関数記号と呼ぶ．さきほど用いた構成子"∧"

と“コ”を述語関数記号として定義すると，内部構造（＊）は

“∧(コ(A, B))C”

という文字列となり，括弧を付けなくとも曖昧さが生じない．

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は必要だが，DCG形式には表現できない点を挙げた．それらの点を表現できるようにした記法として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
  O11, O12, ..., O1n(1) ;
    ...
  Om1, Om2, ..., Omn(m).
without_priority
  P1, P2, ..., Pl.
```

（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．先ほど図2に示したDCG形式の構文規則とこの図3に示す構成子定義によって，DCGo 記法による内包論理の定義となる．また，DCGo 記法の BNF による定義を付録に示す．

```
with__priority
      ":", "," ;
      ( not, lambda );
      and.
    without__priority
      p__syml.
```

図3 内包論理の構成子定義（一部分）
Fig. 3 Constructor definition of intensional logic (part).

以上の記法に従って記述された構文規則には，終端記号一つだけが右辺にあらわれる構文規則と，そうでない構文規則がある．本論文では，前者を辞書型の構文規則，後者を規則型の構文規則と呼ぶ．DCGo 記法では，これら二つの構文規則のうち規則型の構文規則に，内部構造を正しく構成するための条件を設けた．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，右辺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，構造表現の構文記述には十分であり，かつ，DCG形式の記述力を損うものではない．

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 節で挙げた必要事項の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，必要事項の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，必要事項の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が満足され，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ほかに，一般的な制約がある[2),3)]．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．

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，右辺が空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[2)]．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に示した二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．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，構造情報を付加した構文規則に変換し，それを BUPトランスレータ[3),4)]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，外部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，構造表現と構文規則の対応を示す．我々は，構造表現を基本表現と構成表現に分けている．基本表現は，辞書型の構文規則で表すことができ，構成表現は，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表される構成表現を，そのまま左辺の構成表現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，構造表現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．

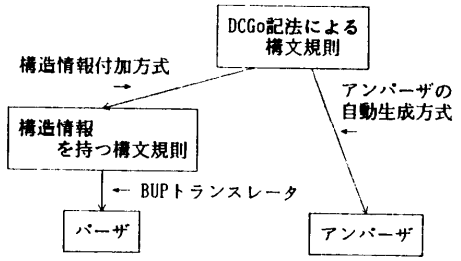

図4 構文規則からのパーザ，アンパーザの生成
Fig. 4 Parser and unparser generation from syntax rule.

① 辞書型の構文規則

辞書型の構文規則では，基本表現を表している．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．例えば，

```
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．例えば，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, T 2)) --> 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[1),2)] とボトム・アップ・パーザの BUP[3),4)] 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[2),4)]. 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[3),4)] 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.

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, 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, どの表現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.

```
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)
Fig. 5 BUP parser on intensional logic (part).

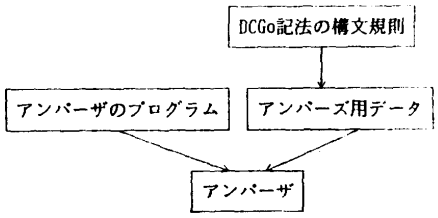

図 6 DCGo 記法の構文規則から, アンパーザを生成する過程
Fig. 6 Unparser generation process of syntax rule on DCGo-notation.

(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

という三つの要素を持ち, 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

という二つの要素を持つ.

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．例えば，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，内部構造が基本表現の場合と，再帰的な構造の構成表現の場合に分けて説明する．構成表現の場合には，（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 基本表現の場合

基本表現の内部構造は，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 構成表現の場合

構成表現の内部構造は引数が再帰的な構造をしている．構成表現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.

引数が構成表現の場合には，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，引数の構成表現の演算子（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

例えば，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の構成子定義により，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，引数に括弧を付けない.

（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", "$x:t$", "$x:t \wedge y:t$"]

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

"$\lambda x:t.\ x:t$"

が生成される．また，(b)で例に挙げた内部構造

["λ", "$x:t$", "$x:t \wedge y:t$"]

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

"$\lambda x:t.(x:t\wedge y:t)$"

が生成される．

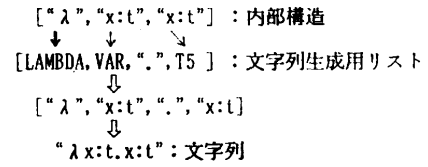

図7 ["λ", "$x:t$", "$x:t$"] をアンパーズする例
Fig. 7 Example of unparsing ["λ", "$x:t$", "$x:t$"].

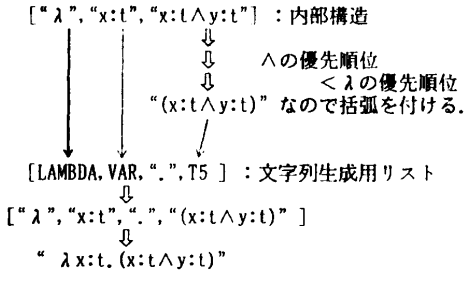

図8 ["λ", "$x:t$", "$x:t\wedge y:t$"] をアンパーズする例
Fig. 8 Example of unparsing ["λ", "$x:t$", "$x:t\wedge y:t$"].

4. 自動生成方式適用例

我々は汎用の論証支援システム EUODHILOS[5] を開発中である．従来の証明チェッカや証明コンストラクタ(LCF[6], EKL[7] 等)の多くは扱う論理系を固定したり，あるいは，推論機能や表現形式を固定して各々のチェック機能，証明構成機能の充実を目指している反面，与えられた論理系や推論機能，表現の下での証明しか扱えない．EUODHILOSは，様々な問題領域を対象とした論証支援システムであるだけでなく，推論機能や問題の表現もユーザによる定義を可能としたことを特徴としている．ユーザによる表現の定義を特徴の一つとしているEUODHILOSに本論文で述べた記法と自動生成方式を適用した．

EUODHILOS ではユーザがエディタを用い，DCGo 記法で記述される構文規則を入力する．本自動生成方式によって生成したパーザとアンパーザを用いた例を**図9, 10**に示す．

図9はパーザによって変換された内部構造を利用した構造エディタの論理式エディタである．指定した外部表現をパーザを用いて内部構造に変換し，その構造に従った表示を行っている．

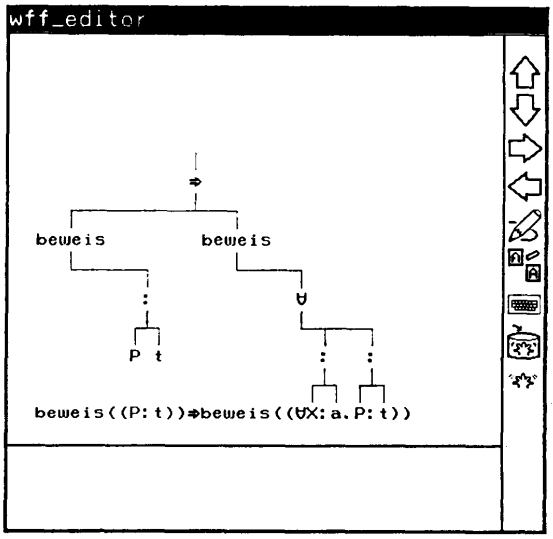

図9 論理式エディタ
Fig. 9 Wff-editor.

図10は証明を行うためのツールの思考シートである．ここでは指定された外部表現をパーザによって内部構造に変換し，それを処理して新たに内部構造を作成する．この新たな内部構造をアンパーザで外部表現に変換し，ユーザへの表示を行っている．

本自動生成方式により，一般ユーザにとってパーザやアンパーザの生成を行う負担は大幅に軽減された．しかし実際に使ってみると，一般に，ユーザの意図する構文規則を一度で書くことは難しいことがわかった．難しい理由として次の2点が挙げられる．

① 構文規則が誤っているため，使えるはずの外部表現が使えない．

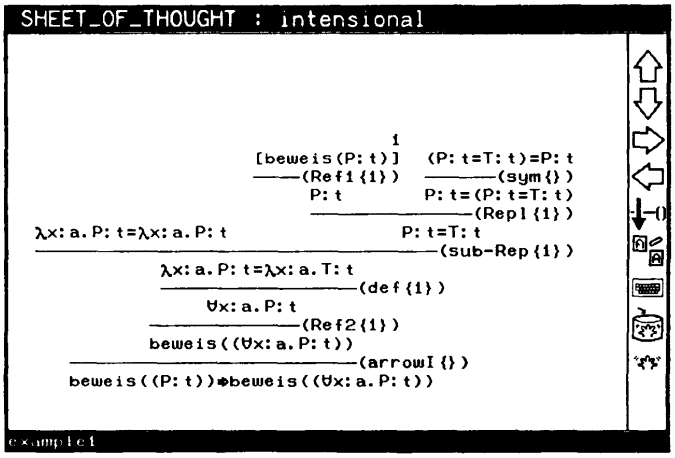

図10 思考シート
Fig. 10 Sheet of thought.

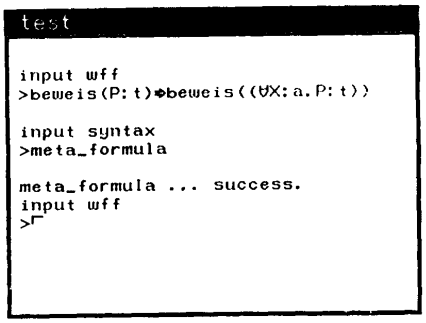

図 11 構文チェッカ
Fig. 11 Syntax checker.

② 構成子定義や構成子の使い方が誤っているため，ユーザが意図していた構造と実際に作られた内部構造が対応していない．

構文記述をより簡単にするために，①の対策として，外部表現を確認するための構文チェッカを作成した．これを**図 11** に示す．このチェッカの中では，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．また，②の対策としては論理式エディタを用いた．このエディタの中で，内部構造に従った構造表示を見ることによってチェックを行う．

構文チェッカと論理式エディタを用いることにより，正しい構文規則を素早く完成させることができる．これらを用いることにより，これ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，第二階述語論理，Hoare論理等を定義したが，どの論理系でも，DCGo 記法により構文を記述する労力が著しく軽減した．

以上で述べた方式は，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[8] によって実現されている．

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る．このことはユーザにとって負担が軽減され，本来の作業 (EUODHILOS で言えば，証明) に集中できるという点で非常に有効であった．ただし，4章にも示したようにより有効に用いるためには，構文的妥当性と意味的妥当性をチェックする機能が必要であろう．

従来，実用化レベルのパーザ・ジェネレータとして Yacc[9),10)]，出力用文字列まで考慮したシステムとして Synthesizer Generator[11)] が開発されてきたが，Yacc には，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，Synthesizer Generator では入力用と出力用の構文を別々に記述するので，両者の整合性にも注意を払わなければならない，という問題点があるため我々の目的を満足するものではない．

本論文で述べた DCGo 記法と自動生成方式を用いれば，一般ユーザでもパーザ，アンパーザを手軽に生成することが可能であるため，システムとユーザが文字列で対話するシステムにおいて必要なパーザ，アンパーザ生成ツールとして有効である．なかでも，構文が確定しておらず，構文規則が頻繁に変更されるような場合には，特に有効である．

2章，3章では，DCGo 記法と自動生成方式について論理式を例にとって説明したが，これは構造表現一般にあてはまるものである．具体的な応用例の一つとして，プログラミング言語の開発支援環境の一つである構造エディタの作成を考えている．

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いきたい．

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) Pereira, F. C. N. and Warren, D. H. D.: Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artif. Intell.*, Vol. 13, pp. 231-278 (1980).
2) Clocksin, C. F. and Merish, C. S.: *Programming in Prolog*, Springer-Verlag (1981).
3) 松本，田中，平川，三吉，安川，向井，横井：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*, pp. 1-9, ICOT (1983).
4) Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H.: BUP: A Bottom-Up Parser Embedded in Prolog, *New Generation Computing*, Vol. 1, No. 2, pp. 145-158, Ohmsha (1983).
5) 南，沢村，佐藤，土屋，袈：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*, pp. 93-102, ICOT (1988).
6) Gordon, M. J., Milner, A. J. and Wadsworth, C. P.: *Edinburgh LCF, LNCS 78*, Springer-

Verlag (1978).
7) Ketonen, J. and Weening, J. S.: *EKL—An Interactive Proof Checker, User's Reference Manual*, Department of Computer Science, Stanford University (1984).
8) Chikayama, T.: *ESP Reference Manual*, ICOT Technical Report, TR-044, ICOT (1984).
9) Johnson, S.: *Yacc: Yet Another Compiler-Compiler*, Computing Science Technical Report, No. 32, AT&T Bell Laboratories (1975).
10) Lesk, M. E.: *Lex—A Lexical Analysis Generator*, Computing Science Technical Report, No. 39, AT&T Bell Laboratories (1975).
11) Reps, T.: *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University (1985).

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．

構文規則と構成子定義の BNF による定義

(1) 構文定義

<構文規則・構成子定義>::=<構文規則定義>
<構成子定義>

(2) 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>|<述語関数記号>
<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>

(3) 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[8] を参照してください．

(平成元年 12 月 14 日受付)
(平成 2 年 9 月 11 日採録)

大橋　恭子 (正会員)

昭和 39 年生．昭和 61 年津田塾大学学芸学部数学科卒業．同年富士通(株)に入社．現在富士通研究所ソフトウェア開発部勤務．以来，論証支援システムの開発，主に言語処理系の開発に従事．

横田かおる (正会員)

昭和 36 年生．昭和 60 年慶應義塾大学文学部人間関係学科卒業．同年富士通(株)入社．現在，富士通研究所ソフトウェア開発部勤務．論証支援システムの開発に従事．

南　　俊朗 (正会員)

1951 年生．九州大学理学部数学専攻修了．富士通(株)国際情報社会科学研究所に勤務．圏論，計算機科学，論理学に興味を持つ．日本数学会，LA，EATCS 各会員．

沢村　一（正会員）

1949 年生．1978 年北海道大学工学部情報工学専攻修了．富士通(株)国際情報社会科学研究所に勤務．論理学とその応用に興味をもつ．日本ソフトウェア科学会，日本科学哲学会などの会員．

大谷　武

1964 年生．1987 年東北大学理学部数学科卒業．1989 年東北大学大学院工学研究科（情報工学専攻）修士課程修了．同年，富士通(株)国際情報社会科学研究所入所．論証支援システムの研究に従事．ソフトウェア科学会会員．