解　説

構造エディタ生成系†

渡辺喜道††　今泉貴史†††　徳田雄洋†††

1. はじめに

近年，ソフトウェアプロセス記述やソフトウェアデータベースなどのソフトウェア開発環境に関する研究が活発に行われている．ソフトウェアを開発するときの環境は，ソフトウェアの作成の効率，プロダクトの品質，保守の容易性に重大な影響を与えるという認識が広まっているためである．

ソフトウェア開発環境の分類の仕方として次の二つがある．

1)　特定の一つのプログラミング言語のための環境と複数のプログラミング言語のための環境

前者の例は Interlisp, Ceder, Smalltalk などの環境であり，後者の例はソフトウェアの生成系を用いて作る環境である．

2)　たくさんの汎用部品を人間が自発的に組み合わせる環境と専用の環境

前者の例は UNIX* 環境であり，後者の例は構造エディタを統一インタフェースとする専用環境である．

構造エディタを生成するシステムである構造エディタ生成系に基づく 1980 年代の代表的なソフトウェア開発環境の例としては，米国カーネギメロン大学計算機科学科の Gandalf プロジェクトの Gandalf システム[3),15)]と，米国コーネル大学の Cornell Synthesizer Generator (以下 C. S. G. と略す)[1),9)~14),16)]がある．

Gandalf システムは，ソースコード管理環境の SMILE，構文開発環境の ALOEGEN，標準ライブラリ付き意味の形式的記述 ARL，インタフェース記述生成系 DBGEN，共通なデータベースと I/O 機能をもつ ALOE カーネルの 5 つからなる．この構成により，複数のプログラマが開発を行うソフトウェアの開発環境を生成することを目標としている．

また，C. S. G. は PL/I のサブセットである PL/CS のプログラムを対象とした Cornell Program Synthesizer プロジェクトから発展したプログラム作成支援システムの生成系である．現在プログラムの教育用やソフトウェア開発用に使用されている．

両者のプロジェクトにはそれぞれ構造エディタ生成系が用意されている．Gandalf システムの場合は構造エディタ ALOE を生成するための構造エディタ生成系，ALOE 生成系であり，C. S. G. の場合は構造エディタ生成系 Synthesizer 生成系である．本稿では，この二つの構造エディタ生成系を中心に解説する．ほかの著名な構造エディタ生成系にはフランス INRIA 研究所の MENTOR[2)] などがある．

2. 基本用語

まず，いくつかの用語の簡単な定義をまとめておく．

テキストエディタ

テキストエディタ (text editor) では編集対象物を単なる文字の並びと考え編集作業を行う．このようなエディタは編集対象物に制限は少なく，プログラムやデータなどを編集することができる．その反面，プログラムを開発するときのような特定の対象物を処理する仕事のときには，プログラミング言語が固定されているのにもかかわらずエディタはこのことを理解していないため，知的処理を提供することができない．

構造エディタ

構造エディタ (structure editor) はテキストエディタとは異なり，ある特定の対象物を編集するこ

† Structure Editor Generators by Yoshimichi WATANABE (Department of Electrical Engineering and Computer Science, Yamanashi University), Takashi IMAIZUMI and Takehiro TOKUDA (Department of Computer Science, Tokyo Institute of Technology).

†† 山梨大学工学部電子情報工学科
††† 東京工業大学工学部情報工学科

* UNIX は AT&T のベル研究所で開発された OS の名称で，同社の登録商標である．

とを目的としたエディタである．構造エディタは編集対象が固定されているために，テキストエディタと比較すると，対象物の性質を知っている分だけ賢いエディタであるということができる．その反面，構造エディタの宿命ではあるが，対象物の汎用性に欠けるという欠点もある．この面を補うのが構造エディタ生成系であるといえる．

正規式

正規式 (regular expression) は，字句（またはトークン）を記述するのに適した表現記法である．正規式は文字に対する結合操作，選択操作，繰り返し操作と括弧操作からなる．これらにより一つの字句を表現する．クラスに属する文字列の集合を言語とみることにより字句を記述する．

文脈自由文法

字句の構造は正規式で指定することができるのに対して，構文の構造を指定する方法が文脈自由文法 (context-free grammar) である．変数としてふるまう非終端記号と定数としてふるまう終端記号の2種類の記号を使い，文法を非終端記号1個（左辺）を0個以上の記号列（右辺）に書き換える操作（生成規則）の集合として表す．この文脈自由文法は BNF (Backus-Naur form) 記法とも呼ばれる．この記法は，言語の構文を指定するための便利な方法である．

動作ルーチン型意味記述

文脈自由文法が構文を指定するのに対して，意味を記述する方法には大きく分けて二つの方法がある．そのうちの一つが動作ルーチン (action routine) 型意味記述である．この方法は構文解析時に生成規則の右辺が左辺に還元されたり左辺の非終端記号が右辺に展開されたりするときに，その生成規則に対応した動作ルーチンであるプログラムの断片を実行し意味付けを行う方法である．

属性文法型意味記述

属性文法 (attribute grammar)[5),7)] 型意味記述は，意味を記述するもう一つの方法である．この方法は，生成規則中の非終端記号や終端記号に属性をもたせ意味付けを行う方法である．この方法は生成規則にローカルに関数の形で意味を記述し，生成規則中の非終端記号や終端記号の属性の値を求め意味付けを行う方法である．生成規則ごとにローカルに関数を記述するので，記述することは比較的やさしいという特徴がある．

3. 構造エディタの特徴

構造エディタは構造をもった対象を扱うエディタであり，構造エディタ生成系は構造エディタを作成する道具である．生成される構造エディタにはプログラミング言語のエディタ（静的意味検査，コンパイル機能，インタプリタ機能を含む），フローチャートのエディタ，証明図のエディタ，電子メールのエディタ，化学反応式の係数合わせをするエディタ，数式処理のエディタなどがあり，構造エディタの応用範囲は広い．

構造エディタとして考えやすいのは，プログラミング言語の構造エディタであろう．プログラミング言語向きの構造エディタを例にとりながら特徴について説明する．

一般ユーザにとって一般のテキストエディタと構造エディタの違いを感じるのは入力方法の違いで表面化される部分であろう．

テキストの入力方法には大別して2種類の方法がある．一つは直接テキストを入力する方法であり，もう一つは演算子のテンプレートを用いる入力方法である．

直接テキストを入力する方法は文字を一つずつ入力する方法である．これに対して演算子のテンプレートを用いる入力方法は，展開しようとする演算子のテンプレートをユーザ（プログラマ）が選択してテキストを入力していく方法である．

テンプレート方式でテキストを入力する場合，プログラミング言語の予約語などを入力する必要がなかったり，インデンテーションが自動的に行われたりするので，ユーザは比較的簡単にかつ分かりやすい形でプログラムを作成できるという利点がある（一般にテンプレートからある項目を選択すると，その選択した項目の構文がディスプレイ上に現れる．その時点で同時に予約語などが表示されたり，インデンテーションが行われていたりするため，プログラマは予約語などを入力する必要がなくなる）．

一般の構造エディタはテンプレートによる入力が主である．しかし，細部までテンプレートによる入力方法を用いると煩わしさが表面化するので，細部では直接テキストを入力する方法を用いているものが多い．

これらの入力方式から構造エディタ内部では演算子をノードとする木が構築される．実際にはテキストの入力とは，ユーザによりこのエディタ内部の演算子の木を構築することである．演算子のテンプレートを用いる入力方法は演算子の木をまさに構築していく過程である．

```
 1  itoa(n, s)   /* convert n to characters in s */
 2  char s[];
 3  int n;
 4  {
 5      int i, sign;
 6
 7      if ((sign = n) < 0 )  /* record sign */
 8          n = -n;           /* make n positive */
 9      i = ;
10      do {      /* generate digits in reverse order */
11          s[i++] = n % 10 + '0';   /* get next digit */
12      } while
~
~
~
~
~
~
~
~
~
~
```

図-1　テキストエディタの例

このような内部構造を取ることによりカーソルの概念が拡張される．テキストエディタにおけるカーソルは，文字列中の文字そのもの（または文字と文字の間）の位置を指している(図-1)．これに対して構造エディタではユーザが内部の演算子の木を操作するため，カーソルは編集対象の木のノードの位置を指すように拡張される．つまり，テキストエディタが一つの文字を指していたのに対し，構造エディタでは一つの部分木を指すようになる(図-2)．

```
{quicksort program}
program quicksort (input, output);
const
  n = 10000;
var
  a: array [1..n] of integer;
  i: integer;
procedure sort (p, q: integer);
  var
    i, j, x, w:        ;
  begin
    <statement>
  end { sort };
begin
  <statement>
end. { quicksort }


Positioned at type_denoter
```

図-2　構造エディタの例

さらに，テキストエディタと構造エディタの違いはテキストの表示方法である．一般のテキストエディタではエディタ内部に木をもっていないので一次元的なデータ構造とみなせる．したがって，そのデータを順にディスプレイなどの出力媒体に表示すればよい．これに対して構造エディタはエディタ内部に演算子の木をもっているため，その木のデータ構造を適当な形（テキストエディタにおける一次元的なデータ構造に相当するもの）に変換して出力媒体に送ることにより表示する．この変換を文字列から木を作ることであるパース（解析）に対してアンパース（逆解析）と呼ぶ．したがって，構造エディタではアンパースによりユーザに表示を行うことになる．このとき，カーソルの位置は反転表示や点滅表示される．テキストエディタの場合は1文字に相当する部分が反転表示や点滅表示されるが，構造エディタの場合は一つの部分木に相当する部分が反転表示や点滅表示される．

4. 構造エディタ生成系の技術的基礎

構造エディタ生成系について詳しく述べる前に，従来のソフトウェア生成系のいくつかの概念について述べる．代表的なソフトウェア生成系には，字句解析系生成系とコンパイラ生成系がある（たとえば，UNIX にはソフトウェア生成系用のツールとして字句解析系生成系 lex[3] とコンパイラ生成系 yacc[4] が用意されている）．

字句解析系生成系に入力するデータは，正規式を用いて表現した文字列パターンとその文字列パターンにマッチしたときに行う処理の意味記述からなる．図-3 は lex への入力データの例の一部分である．

動作ルーチン型意味記述と属性文法型意味記述の例を示す．

1)　動作ルーチン型意味記述の例

```
letter          [a-zA-Z_]
digit           [0-9]
letter_or_digit [a-zA-Z_0-9]
white_space     [ ¥t¥n]
blank           [ ¥t]

%%
">="            return token(GE);
"<="            return token(LE);
"=="            return token(EQ);
"!="            return token(NE);
"+="            return token(PE);
"-="            return token(ME);
"*="            return token(TE);
"/="            return token(DE);
"%="            return token(RE);
"++"            return token(PP);
"--"            return token(MM);
```

図-3 lex への入力記述の例

```
[A-Za-z][A-Za-z0-9]*
        {val=save (yytext);
         return IDENT;
        }
[0-9]+
        {val=atoi (yytext);
         return NUM;
        }
```

2) 属性文法型意味記述の例

```
[A-Za-z][A-Za-z0-9]*
        {IDENT. val:=save (yytext);
         IDENT. len:=strlen (yytext);
        }
[0-9]+
        {NUM. val:=atoi (yytext);}
```

これらは，識別子（英字で始まり0回以上の英数字の繰り返し）と非負の整数（1回以上の数字の繰り返し）の意味記述の簡単な例である．識別子と非負の整数を正規式で表し，動作ルーチンや属性文法の記述を{ }内に記述した．これらの意味記述の例は識別子名や整数値を保存する意味記述の一部分である．

コンパイラ生成系に入力するデータは，文脈自由文法を用いて記述した生成規則と，その生成規則に対応する処理の意味記述からなる．図-4 は yacc に入力するデータの例の一部分である．

動作ルーチン型意味記述と属性文法型意味記述を用いた例を示す．{ }内が意味記述である．

1) 動作ルーチン型意味記述の例

```
expr0→expr1 '+' term
        {op2=pop( );
         op1=pop( );
         push (op1+op2);
        }
term0→term1 '*' fact
        {op2=pop( );
         op1=pop( );
         push (op1*op2);
        }
```

2) 属性文法型意味記述の例

```
expr0→expr1 '+' term
        {expr0. val:=
         expr1. val+term. val
        }
term0→term1 '*' fact
        {term0. val:=
         term1. val * fact. val
        }
```

これらの例は算術式を評価するための意味記述の一部分である．ここで，expr と term の添字として付いている0と1は文法規則の右辺や左辺に出てくる同一な記号を区別するための添字である．

このようなコンパイラ生成系の発展として構造エディタを自動生成する試みが研究されている．いわゆる構造エディタ生成系である．コンパイラが扱うデータは構文などの構造をもったオブジェクトであり，同様に構造エディタも構造をもったオブジェクトを扱う．このため，コンパイラ生成

```
%token GE       /* >= */
%token LE       /* <= */
%token EQ       /* == */
%token NE       /* != */
%token PE       /* += */
%token ME       /* -= */
%token TE       /* *= */
%token DE       /* /= */
%token RE       /* %= */
%token '='
%right '=' PE ME TE DE RE
%left EQ NE
%left '<' '>' GE LE
%left '+' '-'
%left '*' '/' '%'
%%
program
        :       { init(); }
          definitions
                { end_program(); }
        ;
definitions
        : definition
        | definitions definition
                { yyerrok; }
        | error
        | definition error
        ;
```

図-4 yacc への入力記述の例

系と同様な技術を用いて構造エディタを自動生成することが可能である．つまり，構造を帰納法的に書き下し構造ごとに意味を記述することにより構造エディタは自動生成可能となる．一般に，構造エディタ生成系はコンパイラ生成系を基礎として作られる．

しかし，コンパイラ生成系がバッチ型のソフトウェアを生成するのに対し構造エディタ生成系は対話型のソフトウェアを生成することが両者の大きな相違点である．対話型ソフトウェアでは視覚的インタフェースが複雑となるため，構造エディタ生成系のインタフェースも複雑となる．

このように，構造エディタ生成系はコンパイラ生成系の発展とみなすことができる．次章以降の **5.**, **6.** でコンパイラ生成系の二つの意味記述の方法に基づく構造エディタ生成系について述べる．

5. カーネギメロン大学の ALOE 生成系

ここでは，米国カーネギメロン大学の Habermann らによって開発された構造エディタ ALOE を生成する ALOE 生成系について述べる．構造エディタ ALOE（エイロー）は A Language Oriented Editor の略称で，構造エディタ生成系により生成される構造エディタ群の総称である．**図-5** に ALOE の画面の例を示す．

ALOE生成系は動作ルーチン型意味記述に基づく構造エディタ生成系であり，構造エディタを生成するためのソフトウェア自身も構造エディタの形式で作られている．

構造エディタ ALOE は，次のようにして構造エディタ生成系 ALOEGEN から生成される．

構造エディタ生成系 ALOEGEN を起動する．起動された構造エディタ生成系 ALOEGEN 自身も構造エディタであり，テンプレート方式で数々の要素をカーソルを移動しながら入力していく．この入力する要素は，ALOE 文法と呼ばれる記述で，この文法を入力することにより構造エディタを生成する．ALOE 文法としては大きく次の三つに分類される．

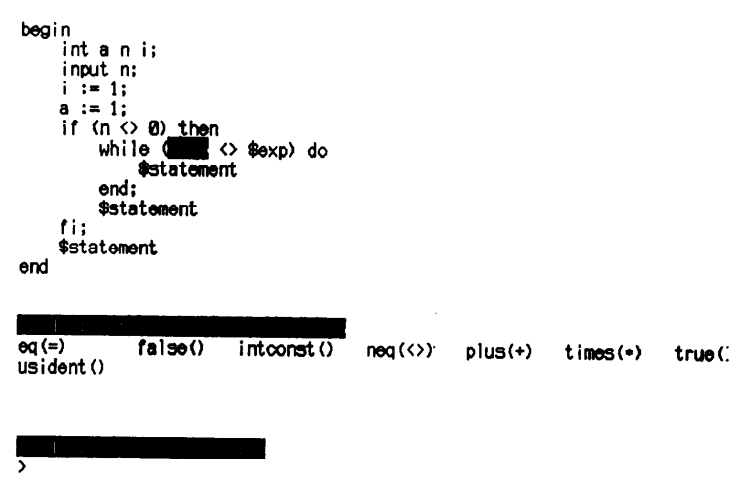

図-5 ALOE の例

Ⅰ）文法を文脈自由文法で表したときの終端記号に関する記述

Ⅱ）文脈自由文法の非終端記号に関する記述

Ⅲ）クラスと呼ばれるカテゴリに関する記述

終端記号に関する記述を行うときには，終端記号に関する情報を記入する場所（ディスプレイ上では，＞Terminals＜ と記述されている部分）にカーソルを移動し，その内部の部分木をテンプレートに従い入力する（**図-6**（a））．**図-6**（b）は終端記号BMODEに関する記述の例である．この部分に記入する主な内容6つを以下にあげる．

Ⅰ-1）終端記号の型の宣言

終端記号の型には，static, representation, usesite, defsite の4種類がある．static と宣言された終端記号は，その終端記号自身に値をもたない．これは，プログラミング言語における型名や true や false といった列挙型の終端記号に相当する．representation と宣言された終端記号は，ユーザが与えた値を示す内部表現の文字列を蓄えることができる．たとえば整数 123 は，整数に対応する終端記号の内部では文字列"123"として蓄えられる．さらに usesite や defsite と宣言された終端記号は，representation と宣言された終端記号がもつ値に加えて，同じ名前をもつほかのすべての usesite や defsite 宣言された終端記号の名前のリストをもつ．この名前のリストは，意味処理時の特に記号表の操作のときに非常に役立つリストである．

図-6(b)では，

{static}

と記されている部分がこの宣言で，終端記号

BMODE は static 型であることを示している．

I-2) 意味処理ルーチン名の記述

終端記号が認識されたときに行う意味処理は意味処理ルーチンを呼び出すことにより行われる．したがって，呼び出す意味処理ルーチンの名前を記述する必要がある．意味処理ルーチンは，意味処理用の言語 ARL (Action Routine Language) を用いてほかの部分に記述しこの部分には記述しない．図-6(b)では，

daemon : 〈none〉

と記されている部分がこの宣言で，終端記号 BMODE はそれが認識されたときに意味処理を行わないことを示している．

I-3) シノニムの定義

この部分には終端記号と同義語になる文字列を記入する．これは終端記号の名前が長い場合に有用である．図-6(b)では，

synonym : "BOOL"

と記されている部分がこの宣言で，終端記号 BMODE の同義語が BOOL であり，BMODE の代わりに BOOL と入力しても同じ構造になることを示している．

I-4) 字句解析ルーチン名の記述

終端記号の型が static 以外で宣言された場合はその終端記号が値をもつため，その値を求めるためのルーチンが必要である．この字句解析用のルーチンの名前を記述する．構造エディタ ALOE のカーネルに組み込まれている字句解析ルーチンには，lexreal (実数のための字句解析)，lexstring (文字列のための字句解析)，lexchar (文字のための字句解析)，lexcomment (注釈のための字句解析)，lexvariable (変数のための字句解析)，lexinteger (整数のための字句解析)，lextext (テキストのための字句解析) の 7 種類がある．このほかに字句解析用のルーチンが必要な場合は，C 言語[6] を用いて字句解析用の関数を作りその関数名をここに記述する．図-6(b)では，

lex : 〈none〉

と記されている部分がこの宣言で，終端記号 BMODE は static 型でありそれ自身に値をもたないので記入しない（デフォルトの値が 〈none〉 となっている）．

```
Language name: 'asple'
Language root: $languageroot
>Terminals<
>Non-terminals<
>Classes<
```

(a) ALOE の初期画面の構成

```
Language name:  'asple'
Language root:  PROGRAM
Terminals:

BMODE =
        |static| | daemon:  <none> | synonym: 'BOOL' |
        lex: <none> | infop: 19 |
        attributes:
          <none>
        unparsing:
        (0) traproot: FALSE | trapdoor: <none>

            'bool'
```

(b) ALOE における終端記号情報の入力記述例

```
Non-Terminals:

PROGRAM =
        declarations statements | daemon: aProgram | synonym: <none> |
        precedence: <none> | scope: TRUE | infop: 26 |
        attributes:
            name: env | type: TREE | grammar: env | daemon: <none>
            name: stack | type: TREE | grammar: stack | daemon: <none>
        unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            'begin@+@n@l@n@20-@nend'
```

(c) ALOE における非終端記号情報の入力記述例

```
Classes:

declaration =
        DECLARATIONS
```

(d) ALOE におけるクラス情報の入力記述例

図-6

Ⅰ-5) 属性の記述

終端記号のもつ値のほかに別の補助的な値を必要とするときにこの部分に属性として記述する．ここには属性の名前と型を記述する．属性の型には，文字型，整数型，論理型，文字列型，構造化された木の5種類がある．図-6(b)では，

attribute :
〈none〉

と記されている部分がこの宣言で，終端記号BMODEは補助的な値を必要としないことを示している．

Ⅰ-6) アンパース法の記述

終端記号が認識されたときにディスプレイ上に出力する方法についての記述を行う．一般に，構造エディタ ALOE 内部には最小限必要な情報だけで抽象木と呼ばれる内部構造の木を作っている．したがって，その構造をディスプレイ上に出力するときにはユーザにとって分かりやすい形に変換してから出力する必要がある（抽象木を具象木と呼ばれる詳しい情報をもった木に変換して出力することになる）．この変換法と出力仕様を記述する．ここに記述するアンパース法は1種類に限らず，数種類の記述ができる．図-6(b)では，

unparsing :
(0)traproot : FALSE | trapdoor : 〈none〉
"bool"

と記されている部分がこの宣言で，終端記号BMODEが認識された場合に bool と表示することを示している．

次に非終端記号に関する記述を行うときについて述べる．非終端記号に関する情報を記入する場所（ディスプレイ上では 〉Non-terminals〈 と記述されている部分（図-6(a)））にカーソルを移動し，その内部の部分木をテンプレートに従い要素を埋めていく．この部分に記入する主な内容7つを以下にあげる（**図-6**(c)）．

Ⅱ-1) 非終端記号を左辺とする生成規則の右辺の記述

文脈自由文法で表したときの生成規則の左辺の非終端記号に対する右辺の終端記号や非終端記号（後に述べるクラスも含む）の列を記入する．また，非終端記号はその非終端記号が展開（分解）されるときの規則（ALOE 内部の抽象木表現における子供の数）によって，二つの種類に分けることができる．一つは，非終端記号が展開されるときに，展開後の非終端記号や終端記号の数が一定であるような非終端記号である（この種の非終端記号は Fixed arity nonterminal と呼ばれる）．もう一つは，展開後の非終端記号や終端記号の数が一定でないような非終端記号である（この種の非終端記号は Variable arity nonterminal と呼ばれる）．この部分に記入するときに，Variable arity nonterminal の場合には，〈 〉で囲まれてディスプレイに表示され，Fixed arity nonterminal と区別できるようになっている．図-6(c)では，

declarations statements

と記されている部分がこの宣言で，生成規則

PROGRAM : :=declarations statements

を示している．

Ⅱ-2) 意味処理ルーチンの記述

終端記号の場合と同様に，この非終端記号が認識されたときに行う意味処理を言語 ARL で書かれた意味処理ルーチンの名前として記述する．図-6(c)では，

daemon : aProgram

と記されている部分がこの宣言で，非終端記号PROGRAM が認識されたとき意味処理ルーチン aProgram が呼び出されることを示している．

Ⅱ-3) シノニムの定義

終端記号の場合と同様に同義語の文字列を記入する．図-6(c)では，

synonym : 〈none〉

と記されている部分がこの宣言で，非終端記号PROGRAMには同義語がないことを示している．

Ⅱ-4) 優先順位に関する記述

非終端記号（非終端オペレータ）に優先順位をつけたいときに記入する．優先順位は整数値で表現する．これはアンパースする際，優先順位によって括弧を付ける操作などに用いられる．図-6(c)では，

precedence : 〈none〉

と記されている部分がこの宣言で，非終端記号PROGRAM には優先順位がついていないことを示している．

Ⅱ-5) スコープの記述

ここには論理値 (TRUE, FALSE) を記入する．この論理値が真のとき，この非終端記号はそれをルートとする抽象木の部分木内にある終端記号の

中でその型が usesite と defsite である終端記号の名前の表をもつことを示している．図-6（c）では，

scope : TRUE

と記されている部分がこの宣言で，非終端記号 PROGRAM は終端記号の名前の表をもつことを示している．

Ⅱ-6) 属性の記述

終端記号の場合と同様にこの非終端記号に属性が必要な場合に記述する．図-6(c)では，

attributes :

name : env | type : TREE | grammar : env | daemon : 〈none〉

name : stack | type : TREE | grammar : stack | daemon : 〈none〉

と記されている部分がこの宣言で，非終端記号 PROGRAM は二つの TREE 型属性 env と stack をもつことを示している．

Ⅱ-7) アンパース法の記述

これも終端記号の場合と同様にディスプレイに出力するための方法を記述する．図-6(c)では，

unparsing ;

(0) traproot : FALSE | trapdoor : 〈none〉

"begin@＋@n@1@n@2@－@nend"

と記されている部分がこの宣言で，非終端記号 PROGRAM が認識されたとき，

begin

$declaration ;

$statement

end

のように表示することを示している．

最後に，クラスに関する記述を行うときにはクラスに関する情報を記入する場所（ディスプレイ上では 〉Classes〈 と表示されている部分（図-6(a)））にカーソルを移動し，その内部の部分木をテンプレートに従い入力する（図-6(d)）．ここにはクラス名とクラスの構成要素を入力する．一つの非終端記号が数種類の生成規則により展開される場合に，これらの展開される要素を構成要素とする集合を考える．これをクラスと名付ける．一つのクラスがどのような非終端記号や終端記号から構成されるかを示している．図-6(d)では，

declaration＝

DECLARATIONS

と記されている部分がこの宣言で，クラス declaration は DECLARATIONS から構成されることを示している．

以上の記述に対してエディタ生成用のコマンドを起動する．すると，ALOE 生成系は上記の記述を自動的にC言語のプログラムに変換し，それがコンパイル・リンクされ，目的とする構造エディタが生成される．このように，文法の記述と意味の記述を与えるだけで生成系のメカニズムにより比較的簡単に構造エディタを生成することができる．

6. コーネル大学の Synthesizer 生成系

Synthesizer生成系は，米国コーネル大学の Reps と Teitelbaum によって開発された構造エディタ生成系である．

コーネル大学では，C.P.S. (Cornell Program Synthesizer) と呼ばれる特定の言語に対するプログラムの編集・実行・デバッグをサポートする対話的なプログラム開発環境の研究が行われていた．これは，コーネル大学のプログラミング言語の初等教育用システムとして，年間 1500 人の学生に対して使用された．Synthesizer 生成系は，この C.P.S. を発展させて構造エディタ生成系とした属性文法に基づいたシステムである．

Synthesizer生成系は大きく分けて二つの部分から構成される．一つは，SSL (Synthesizer Specification Language) と呼ばれる言語で記述したエディタの仕様から構文表を生成するトランスレータであり，もう一つは，属性付き木を対話的に操作するドライバによって構成されるエディタカーネルである．エディタカーネルは，キーボードやマウスからの入力に対し，対象としている木上に適当な操作を施す．

Synthesizer生成系を用いてエディタを作成するための方法は次のとおりである．

1) システムに与える仕様の定義

SSL でシステムに与える仕様を記述する．

2) 構造エディタの作成

sgen と呼ばれるシェルスクリプトを用いてトランスレータを起動する．そこで作成された構文表とエディタカーネルから目的の構造エディタが生成系のメカニズムにより生成される．図-7 に Synthesizer生成系を用いて作成した簡単な電卓機能をもつ構造エディタの例を示す．

構造エディタの仕様を記述するための言語 SSL は，項の代数 (term algebra) と属性文法 (attribute grammar) の概念を基礎としている．この SSL で記述される仕様は，一般のテキストエディタを用いて作成できるが，仕様記述用の構造エディタを用いて作成することも可能である．

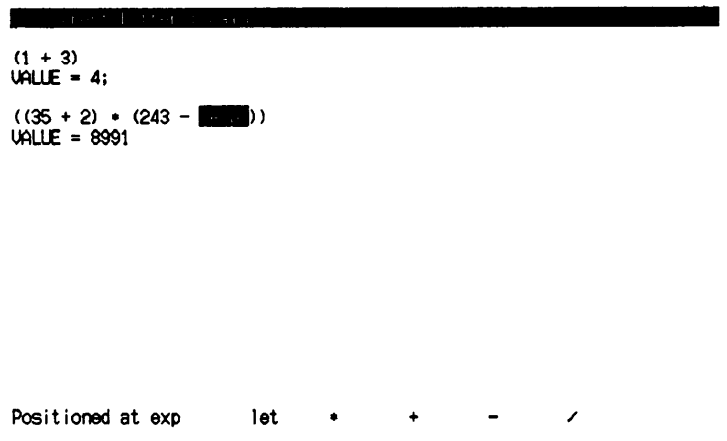

図-7 Synthesizer 生成系を用いて作成した構造エディタの例

SSL では生成する構造エディタの仕様を宣言のリストの形式で記述する．記述する宣言には，抽象構文の宣言，属性のルール，アンパースの方法，テンプレートの記述，字句解析部の記述などがある．図-8 は図-7 に示した構造エディタを生成するための SSL の記述の一部である．

1) phylum 宣言

phylum 宣言は，生成する構造エディタの抽象構文を記述する部分であり，次のような形式である．

phylum$_0$:

operator (phylum$_1$ phylum$_2$…phylum$_k$);

この宣言では，phylum$_0$ が phylum$_1$ から phylum$_k$ までの k 個の phylum からオペレータ operator により構成されることを宣言している．これは文脈自由文法の非終端記号と生成規則の定義に相当する．また，正規式を用いて phylum を宣言することも可能であり，その場合は次のような形式を用いる．

phylum :

operator 〈regular-expression〉;

これは終端記号の定義となる．図-8 では，

calc : CalcPair (exp calc);

INTEGER : 〈[0-9]+〉;

などがこの宣言で，この場合 phylum calc は二つの phylum exp と calc からなること，phylum INTEGER は 0〜9 までの数字の 1 回以上の繰り返しであることをそれぞれ示している．

2) property 宣言

それぞれの phylum に対して，リストであるとかオプショナルであるとかの性質を宣言することが可能である．リストであると宣言された phylum は線形リストのような働きをし，オプショナルであると宣言された phylum は選択的に画面表示などを変更することができる．この場合次のような宣言を用いる．

list phylum$_0$, …, phylum$_k$;

optional phylum$_0$, …, phylum$_k$;

optional list phylum$_0$, …, phylum$_k$;

図-8 では，

list calc;

がその宣言で，phylum calc はリスト構造の phylum であることを示している．

3) root 宣言

文脈自由文法の開始記号に相当するシステムが扱う編集対象を指定する宣言である．これは次のような形式である．

root phylum ;

図-8 では，

root calc;

がその宣言で，phylum calc が開始記号に相当することを示している．

4) attribute 宣言

終端記号や非終端記号に対して属性を割り当てる宣言であり，次のような形式である．

phylum$_0$ {synthesized phylum$_0$ attribute$_0$;

…

inherited phylum$_k$ attribute$_k$;

};

この宣言では，phylum$_i$ 型の属性 attribute$_i$ を宣言しているが，synthesized とされた属性は合成属性であり，inherited とされた属性は相続属性である．図-8 では，

exp {synthesized INT v;} ;

などがこの宣言で，この場合 phylum exp は INT 型（整数型）合成属性 v をもつことを示している．

5) equation 宣言

属性文法における各生成規則の意味記述を行う部分であり，次のような形式である．

```
phylum :
  operator  {output-attribute=expression ;
              ...
             output-attribute=expression ;
            } ;
```

このとき output-attribute は，生成規則の左辺の phylum の合成属性，右辺の phylum の相続属性，局所属性のうちのいずれかである．局所属性とは一つの生成規則にローカルな変数である．図-8 では，

```
exp : Null  {exp. v=0 ;} ;
```

などがこの宣言で，この場合 phylum exp の合成属性 v に 0 を代入することを示している．

6) function 宣言

意味処理などを行う際に用いられる関数を定義する宣言であり，次のような形式である．

```
phylum0 identifier0 (
```

```
root calc;      /* root宣言 */
list calc;      /* property宣言 */
/* phylum宣言 */
calc    : CalcPair(exp calc)
        | CalcNil()
        ;
exp:  Null()
    | Sum, Diff, Prod, Quot(exp  exp)
    | Const(INT)
    ;
exp {synthesized INT v;};    /* attribute宣言 */
/* equation宣言 */
exp:  Null{ exp.v = 0; }
    | Sum{ exp$1.v = exp$2.v + exp$3.v; }
    | Diff{ exp$1.v = exp$2.v - exp$3.v; }
    | Prod{ exp$1.v = exp$2.v * exp$3.v; }
    | Quot{ local STR error;
        error   = (exp$3.v == 0) ? "<--DIVISION BY ZERO-->" : "";
        exp$1.v = (exp$3.v == 0) ? exp$2.v : (exp$2.v / exp$3.v); }
    | Const{ exp$1.v = INT; }
    ;
/* unparsing宣言 */
calc: CalcPair[ @ ::=  @  "¥nVALUE = " exp.v ["¥n¥n"] @];
exp:  Null[ @ ::=  "<exp>" ]
    | Sum [ ^ ::=  "(" ^ " + " ^ ")" ]
    | Diff[ ^ ::=  "(" ^ " - " ^ ")" ]
    | Prod[ ^ ::=  "(" ^ " * " ^ ")" ]
    | Quot[ ^ ::=  "(" ^ " / " error ")" ]
    | Const[ @ ::=  ^ ]
    ;
/* transformation宣言 */
transform exp
        on "+"  <exp> : Sum ([exp],[exp]),
        on "-"  <exp> : Diff ([exp],[exp]),
        on "*"  <exp> : Prod ([exp],[exp]),
        on "/"  <exp> : Quot ([exp],[exp])
    ;
transform exp
        on "factor-left" Sum(Prod(a,b),Prod(a,c)) : Prod(a,Sum(b,c)),
        on "factor-right"  Sum(Prod(b,a),Prod(c,a)) : Prod(Sum(b,c),a),
        on "commute" Sum(a,b) : Sum(b,a),
        on "commute" Prod(a,b) : Prod(b,a),
        on "commute" Diff(a,b) : Diff(b,a),
        on "commute" Quot(a,b) : Quot(b,a)
    ;
INTEGER: < [0-9]+ >;                    /* phylum宣言 */
WHITESPACE: < [" "¥t¥n] >;              /* phylum宣言 */
Calc { synthesized calc abs; };         /* attribute宣言 */
Exp { synthesized exp abs; };           /* attribute宣言 */
left '+' '-';                           /* precedence宣言 */
left '*' '/';                           /* precedence宣言 */
/* parsing宣言 */
Calc::= (Exp)           | Calc.abs  = CalcPair(Exp.abs, CalcNil()); |
      | (Exp  Calc)     | Calc$1.abs = CalcPair(Exp.abs, Calc$2.abs); |
      ;
Exp::=  (INTEGER)       | $$.abs = Const(STRtoINT(INTEGER)); |
      | (Exp '+' Exp)   | $$.abs = Sum( Exp$2.abs, Exp$3.abs); |
      | (Exp '-' Exp)   | $$.abs = Diff(Exp$2.abs, Exp$3.abs); |
      | (Exp '*' Exp)   | $$.abs = Prod(Exp$2.abs, Exp$3.abs); |
      | (Exp '/' Exp)   | $$.abs = Quot(Exp$2.abs, Exp$3.abs); |
      | ('(' Exp ')')   | $$.abs = Exp$2.abs; |
      ;
calc  Calc.abs;         /* entry宣言 */
exp   Exp.abs;          /* entry宣言 */
```

図-8 SSL の記述例

phylum$_1$ identifier$_1$,
…
phylum$_k$ identifier$_k$)
{expression} ;

この宣言では，関数 identifier$_0$ が k 個の phylum$_i$ 型の引数 identifier$_i$($1 \leqq i \leqq k$) をとり，計算結果として phylum$_0$ 型の値を返すことを示している．また，関数 identifier$_0$ の本体は expression であり，引数として指定された identifier$_i$ を用いて phylum$_0$ 型の値を計算する．図-8 にはこの宣言はない．

7) unparsing 宣言

この宣言では次の三つの宣言を同時に行う．

① 画面への出力方法の宣言

② ノードが選択可能かどうかの宣言

③ ノードがテキストとして編集可能かどうかの宣言

次のような形式で宣言する．

```
phylum : operator
    [@ : exp-list @ exp-list
         …
         exp-list @ exp-list] ;
```

このとき，@は＾や..に換えることができ，：は::＝に換えることができる．画面への出力方法としては，この生成規則が採用された場合に，：の右辺に書かれたように画面に出力することを示している．また，@を..に換えることにより対応する phylum を画面に出力しないようにすることも可能である．ノードが選択可能かどうかについては，@の場合に選択可能な phylum であり＾や..の場合には選択可能ではない．選択可能なノードにはそのノードにカーソルを移動することができそこに部分木を作成することができる．また，：を::＝で置き換えることによりテキストとして編集可能であることを示す．図-8 では，

```
calc : CalcPair[@ ::=@"¥nVALUE = "exp. v
         [";¥n¥n"]@] ;
```

などがこの宣言で，phylum calc がオペレータ CalcPair のとき

```
〈exp〉
VALUE = 0
```

のように画面に表示されることを示している．

8) parsing 宣言

この宣言は具象構文を定義する宣言である．この宣言に属性方程式 (attribute equation) を加えることもでき，抽象構文の phylum の属性の値を求めるための記述ができる．この宣言は次のような形式である．

```
phylum0 : :=operator(
          tokens phylum1
          tokens phylum2
          …
          tokens phylumk
            tokens) ;
```

図-8 では，

```
Exp : :=(INTEGER)
    {$$. abs=Const (STRtoINT (INTEGER)) ;}
```

などがこの宣言で，この場合

```
Exp : :=INTEGER
```

という具象構文を示している．オペレータ名は明らかなので省略されている．また，｛ ｝内が属性方程式であり，phylum Exp の属性 abs は終端記号 INTEGER の実際の値を phylum exp 型に変換した値であることを示している．

9) entry 宣言

この宣言は，抽象木上の位置と具象構文の入力位置 (parsing 宣言における phylum) との対応を決定する宣言であり，次の形式である．

```
abstract-syntax-phylum
    ~ concrete-syntax-phylum. attribute ;
```

この宣言により，カーソルが抽象木の abstract-syntax-phylum に位置している場合，parsing 宣言の concrete-syntax-phylum により入力が解析され，その phylum の合成属性 attribute の値がカーソルの位置に挿入される．図-8 では，

```
exp ~ Exp. abs ;
```

などがこの宣言で，この場合抽象木上の phylum exp の位置にカーソルがある場合に対応する phylum Exp の合成属性 abs の値がその位置に挿入されることを示している．

10) precedence 宣言

それぞれのトークンには優先順位や結合性（左結合性，右結合性，非結合性）を指定することが可能であり，次のような形式で行う．

```
left token1…tokenk ;
right token1…tokenk ;
nonassoc token1…tokenk ;
```

このとき，一つの宣言の中のトークンは同じレベ

ルの優先順位となり，複数の precedence 宣言が指定された場合は先に宣言されたトークンの優先順位が低くなる．また，left と宣言されたトークンは左結合性をもち，right と宣言されたトークンは右結合性をもつ．図-8 では，

left '+' '−';
left '*' '/';

がこの宣言で，+，−，*，/はすべて左結合性であり，*と/のほうが+や−より優先順位が高いことを示している．

11) transformation 宣言

現在注目している抽象木上のノードの形態が，指定されたパターンと同一である場合に構造の変形を行うエディタコマンドの宣言である．これは次のような形式である．

transform phylum on string
pattern : expression ;

これは，注目している phylum のノードが pattern という形態であった場合に，string というコマンドによって expression という形態に変形できることを示している．図-8 では，

transform exp
on "+" 〈exp〉 : Sum([exp], [exp]);

などがこの宣言で，この場合カーソルの位置にある phylum が exp でその値が 〈exp〉 であるとき "+" コマンドを選択することにより Sum([exp], [exp]) という構造に変形できることを示している．

7. おわりに

UNIX システムの限界の一つは，すべての情報が良い意味でも悪い意味でもバイト列でしかないということである．これは，多くの部品の再利用と組合せを可能にするが，一つ一つの部品は知的に振る舞わないため，ユーザが部品の組合せの着想と責任をもたねばならない．この限界を超えるために構造をもった情報を扱うエディタである構造エディタが出現し，個別開発の負担を救うために構造エディタ生成系が出現した．構造エディタは，もっと知的な振舞いが可能であり，統一的なインタフェースともなり得るからである．

構造エディタ生成系は，明らかに高度な機能をもつソフトウェアをいわば軽々と生成してみせてくれる点で大きな威力をもっている．ところが，ここに構造と非構造のパラドックスとでも呼ぶべき問題がある．たとえば，次のような現象が生じるのである．

1) C言語に熟知した人が構造エディタで簡単な C プログラムを作るのに困難を感じる場合がある．

2) 構造エディタの利用者で，テキストエディタ的アクセス法も構造エディタにもたせるべきであると主張する人が出てくる．

3) テキストエディタでテキストファイルを作り，次に構造エディタの内部形式に変換するほうが，構造エディタで最初から作成するより容易であると主張する人が出てくる．

4) 構造エディタの出力ファイルは，完全に同一の構文規則に基づいていないと相互作用ができなくなってしまう．つまり強く型付けされているため自由に再利用できない．

しかし，次のような点で構造エディタ生成系は優れている．

1) エキスパートでないユーザにも簡単に対話型ソフトウェアが作成できる．

2) 作成されたソフトウェアの保守が容易である．

3) ソフトウェアの作成の効率がよい．

4) プロトタイピングに向いている．

このように，ソフトウェア開発環境改善への構造エディタ生成系の貢献は大きいと思われる．

参考文献

1) Demers, A., Reps, T. and Teitelbaum, T.: Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors, Conference Record of the 8th ACM Symposium on Principles of Programming Language, pp. 105-116 (1981).
2) Donzeau-Gouge, V. G., Kahn, G., Huet, B., Lang and J-J, Lévy : Programming Environments Based on Structured Editors : The MENTOR Experience, in Interactive Programming Environments, D. R. Barstow, H. E. Shrobe and E. Sandewell (Eds.), McGraw-Hill (1984).
3) Habermann, A. Nico et al.: The Second Compendium of Gandalf Documentation, Dept. of Comp. Science, Carnegie-Mellon University (May 1982).
4) Johnson, S. C.: YACC—Yet Another Compiler-Compiler, Bell Laboratories (July 1978).
5) Kastens, U.: Orderd Attribute Grammars, Acta Inf. Vol. 13, No. 3, pp. 229-256 (1980).

6) Kernighan, B. W. and Ritchie, D. M.: The C language, Prentice-Hall (1978).
7) Knuth, D. E.: Semantics of Context-Free Language, Math. Syst. Theory, Vol. 2, No. 2, pp. 127-145 (June 1968).
8) Lessk, M. E.: Lex—A Lexical Analyzer Generator, Bell Laboratories, 39 Comp, Science (Oct. 1975).
9) Reps, T.: Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors, Conference Record of the 9th ACM Symposium on Principles of Programming Languages, pp. 169-176 (1982).
10) Reps, T.: Generating Language-based Environments, MIT press (1984).
11) Reps, T. and Teitelbaum, T.: The Synthesizer Generator, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notice Vol. 19, No. 5, pp. 42-48 (May 1984).
12) Reps, T. and Teitelbaum, T.: The Synthsizer Generator : A System for Construction Language-Based Editors, Springer-Verlag (1988).
13) Reps, T. and Teitelbaum, T.: The Synthsizer Generator Reference Manual 3rd edition, Springer-Verlag (1988).
14) Reps, T., Teitelbaum, T. and Demers, A.: Incremental Context-Dependent Analysis for Language-Based Editors, ACM Trans. Program. Lang. Syst., Vol. 5, No. 3, pp. 449-477 (July 1983).
15) Staudt, J., Barbara, Krueger, W., Charles, Habermann, A. N. and Ambriola, Vincenzo : The GANDALF System Reference Manuals, Dept. of Computer Science, Carnegie-Mellon University (May 1986).
16) Teitelbaum, T. and Reps, T.: The Cornell Program Synthesizer : A Syntax-Directed Programming Environment, Commun. of the ACM, Vol. 24, No. 9, pp. 563-573 (Sep. 1981).

(平成2年7月5日受付)

渡辺 喜道（正会員）

1964年生．1986年山梨大学工学部計算機科学科卒業．1988年同大学院修士課程修了．現在，同大学工学部電子情報工学科助手．ソフトウェア開発環境，対話型ソフトウェアに興味をもつ．ACM, IEEE, 電子情報通信学会，ソフトウェア科学会，人工知能学会各会員．

今泉 貴史（正会員）

1965年生．1987年東京工業大学工学部情報工学科卒業．1989年同大学院理工学研究科修士課程修了．同年同博士後期課程入学，現在に至る．属性文法に基づくプログラミング環境におけるエディタの開発に携わる．属性文法を用いたバージョン管理，コンフィグレーション管理などに興味を持つ．日本ソフトウェア科学会会員．

徳田 雄洋（正会員）

1951年生．1974年東京工業大学理学部数学科卒業．1977年同大学院博士課程情報科学専攻中退．同年同理学部情報科学科助手．1982年山梨大学工学部計算機科学科講師．1985年同助教授．1983-84年カーネギーメロン大学計算機科学科客員科学者．1988年東京工業大学工学部情報工学科助教授．現在に至る．理学博士．この間，言語処理系，ソフトウェア生成系，ソフトウェア開発環境の研究に従事．著書「構文解析」（昭晃堂），「はじめて出会うコンピュータ科学」（全8冊，岩波書店．韓国語版，探求堂）．ACM, IEEE, 電子情報通信学会，日本ソフトウェア科学会各会員．