ソフトウェア工学 61－2
（１９８８．７．２７）

メッセージの受渡しによる環境生成器について

海尻　賢二
信州大学　工学部

対話的なプログラミング環境の生成器の入力記述言語として属性文法は重要なものであるが、種々の欠点も持っている。これを補うものとしてメッセージシステムが提案されているが、不十分な点も多い。そこで我々は実用的な入力記述言語としての改良、対話型の環境には不可欠なインクリメンタルな評価アルゴリズムの提案、それに基づいての構文エディタ生成器の試作を行っている。プロトタイプが一応完成したので本稿ではメッセージシステム全体の概要を述べ、プロトタイプに基づく評価を行う。

an Environment Generator based on Message Systems

Kenji Kaijiri
Faculty of Engineering,Shinshu University

Attribute grammars have been used as description languages for environments generator, but they have some defects. Message systems are proposed to overcome the above defects, but it is not practical. We propose new evaluation method which evaluates messages incrementally.
We implemented the environment generator based on message systems. In this paper we will describe the message system that we propose and the overall structure of the generator. We will evaluate the method experimentally.

1。まえがき

プログラム開発における環境の重要性は昨今よく認識されており、たとえばユーザインターフェイスの優れていることでよく知られているマッキントッシュ上ではＣやＰａｓｃａｌに対して統合された環境が実現されている。過去において多くの言語、多くの機械に対するコンパイラの生成の労力を軽減するためにコンパイラ生成システムが考えられたように、環境に対しても生成器を考える段階に来ている。

統合されたプログラミング環境を仕様から自動生成する環境生成器は色々な角度から研究され、いくつかのプロトタイプが作られている。対象とする環境は構文エディタ、静的意味チェッカ、インタープリタ等からなり、それらの、仕様からの自動生成をサポートする。代表的なものには次のようなものがある。

（１）Synthesizer Generator[1]

代表的な構文エディタであるCornell Program Synthesizerの生成器として作られ、文脈依存的な構文的制約、データフロー異常等をインクリメンタルに認識する構文エディタを自動生成する。文法の記述には属性文法を用い、属性文法のインクリメンタルな解析により認識を行う。Synthesizerはインタープリタも含んでいるが、Generatorではそこまでサポートしてはいない。

（２）ALOEGEN[2]

ALOEGENは総合環境Gandalfの統一されたユーザインターフェイスとしての構文エディタALOEの生成器である。ALOEも文脈依存的な検査を行うが、仕様記述はaction routineとして与える。

（３）SPSG[3]

SPSGは次のような機能を持ったプログラミング環境を生成する。構文エディット、意味誤り検査、デバッグ機能を持つインタープリタ、コード生成。意味誤り検査等の仕様記述はC++の関数の形式で記述している。インクリメンタルな処理は考慮していない。

（４）PSG System[4]

構文、文脈条件、指示的意味等から、構文エディタ、インクリメンタルなインタープリタ等から成る対話的プログラミング環境を生成する。文脈条件は文脈関係として記述し、unificationに基づき、インクリメンタルに検査をおこなう。

編集も含めた環境においては（１）対話的に処理するためにインクリメンタルな処理が要求される、（２）消去等の場合undoが必要であり、部分的な記述が難しい、等の問題がある。このような要求に答え得る環境生成器の入力記述言語としては記述のモジュール性、宣言性等の点から属性文法[5]がよく用いられている。属性文法では非終端語毎にいくつかの属性を用意し、その属性の値を生成規則毎にきめられた意味規則により計算する。各ノードの情報の交換はその間にあるノードの意味規則の順次的な実行により行われる。そのため意味が生成規則単位でまとめてモジュール的に記述できる反面、次の様な欠点をもつため、記述及び処理が冗長となる。１）属性の値は導出木の隣合うノード間でしか伝播できない。２）導出木の個々のノードに於ける属性の数が文法に基づく定数によって抑えられる。

特に１）の欠点は、記述においては属性を伝播するためだけの規則（コピー規則）の増加、処理においては各ノード毎の保存する属性値の増大という好ましくない結果をうみ、そのため大域的属性、文脈依存属性等の改良案[6][7]が提案されている。また属性文法の評価法である依存木に基づく方法をインクリメンタルな評価に適用したも

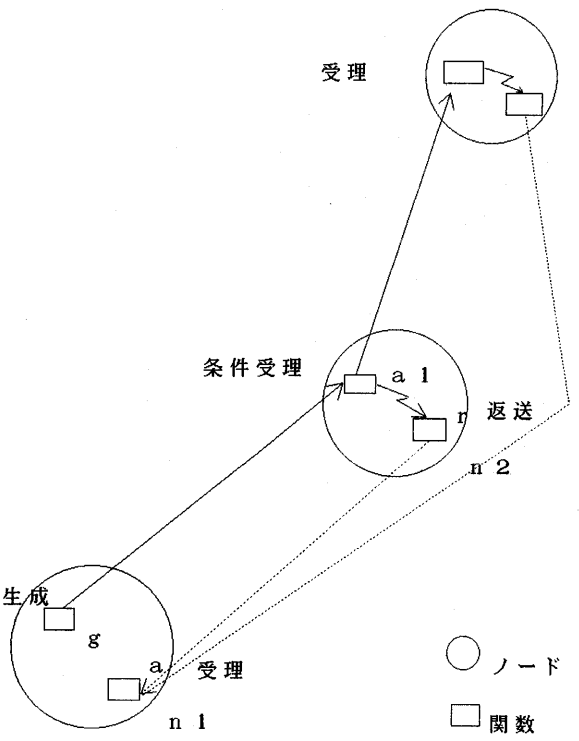

図１　メッセージシステムの概要

のが提案されており、再評価の必要な属性の数に比例する時間複雑性で再評価が可能であることが示されている[8]。

我々はDemersらによって属性文法の改良として提案されているメッセージシステム[9]の環境生成器への適用について検討しているが[10]、本稿では実現した環境生成器のプロトタイプの概要、その処理アルゴリズムを述べ、その評価を行う。

2。メッセージシステム

メッセージの受渡しによる属性評価法は、ユーザによるシステムの拡張を容易にするため、また隣接するノード間のみでの属性の伝播という属性文法の改良として、Alan Demersらによって提案されている。メッセージシステムにおいては属性はメッセージのクラスとして、個々の解析木上の属性はそのクラスのインスタンスとして表される。メッセージはそのクラス毎に決められた伝播順序に従い、受理されるまで伝播して行く。単に伝播していくだけのノードにおいてはなんの処理も書かず、メッセージの生成、受理に関わるノードでのみその処理を記述する。そのため属性文法に於けるいわゆるコピー規則のようなものは存在せず、関係のないノードでのメッセージの保存もない（コピー規則は記述の冗長性、処理のオーバヘッドにつながる。）　メッセージには情報を伝播する主張メッセージ、他のノードへの問い合わせを行う質問メッセージ、そして質問に対する回答を行う返送メッセージの３種類がある。又メッセージを処理する意味関数にはメッセージを生成する生成関数、情報の取得、質問の受付を行う受理関数、そして質問に対して回答を出す返送関数の３種類がある。（図１）

Demersらはメッセージシステムのインクリメンタルな評価法も与えているが、１）メッセージの伝播順序をノード毎に判定するため、結局コピー規則と同様の記述がノード毎に必要となる、２）インクリメンタルな評価においてメッセージを、関数の適用されるノードにすべて保存するため記憶量が大きく成りすぎる、３）メッセージの伝播においてどのノードから伝播してきたかが考慮されない、等の欠点をもつ。そこで次の様な改良を提案し、この提案に基づく環境生成器のプロトタイプを実現し、その性能を評価した。

（１）メッセージの伝播順序は全体としてきまる順序（inorder、parent等）のみとした。これにより個々のノード（クラス）で伝播順序を持つ必要はなくなるが、記述能力が幾分制約される。

（２）依存木の概念に基づく（インクリメンタルな）評価法を提案した。この評価法では受理され得るノードでしかメッセージを保存する必要はない。いわゆるコピー規則に対応するノードではメッセージは保存されない。

（３）受理においてノードの概念を導入し、どのノードから伝播してきたかを区別できるようにした。これによりたとえば代入文の左辺と右辺の各々から伝播してきたメッセージを容易に区別できる。

次に本稿で提案するメッセージシステムの記述の形式を述べる。全体としては以下の形式をとる。

メッセージクラスの宣言
%
木ラベル毎の意味定義の記述
%

メッセージクラスの宣言は以下の形式のメッセージクラス宣言文を並べたものである。

メッセージの名前　メッセージの種類　伝播順序　返送メッセージ

メッセージの種類は

主張メッセージ　　質問メッセージ　　返送メッセージ

の３つである。返送型のメッセージに限り、対応する返送メッセージの名前を書く。返送メッセージについては単独のものとしては記述しない。伝播順序は次の９種類をシステムに定義済みのものとして用意する。

parent	left_sibling	right_sibling
preorder	inorder	postorder
reverse preorder	reverse inorder	reverse postorder

木ラベル毎の意味定義の記述では構文定義で与えた木ラベル毎に、必要なものに対してのみ次の形式で意味を記述する。

木ラベル　begin　意味定義　end

意味定義文は次の３種類がある。

受理記述文　　生成記述文　　返送記述文

受理記述文

形式１。ACCEPT <target class>
形式２。ACCEPT <target class> FROM <node> AS <as class>
形式３。ACCEPT <target class> WITH <with class list> <body>
形式４。ACCEPT <target class> FROM <node> AS <as class> WITH <with class list> <body>

<node>が指定されておれば、そのノードから伝幡してきたメッセージのみ受理する。<as class>があれば受理したメッセージのクラスを変換する。

生成記述文

形式１。GENERATE <target class> VALUE <value>

形式２。GENERATE <target class> WITH <with class list>

形式３。GENERATE <target class> WITH <with class list> <body>

形式１においては<value>で示された式の値をメッセージの値としてメッセージを生成する。一回だけ評価する。形式２においてはその時点において受理されているメッセージのなかで<with class list>と同じクラスのメッセージの値を値としてメッセージを生成する。受理メッセージ中の<with class list>に属するメッセージの数だけ評価する。形式３では<with class list>に属するメッセージに基づいて<body>を実行してメッセージを生成する。

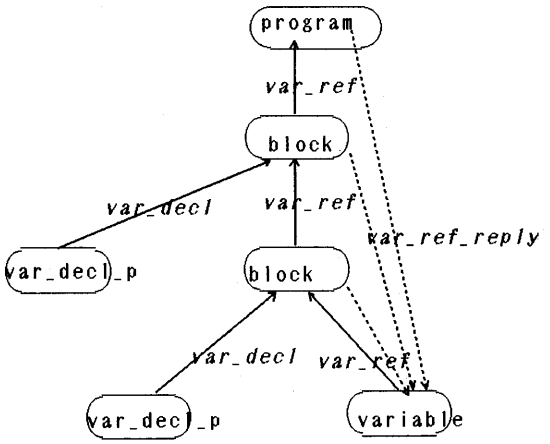

図２　宣言の有無の検査の流れ

［例１］形式１

```
variable -> IDENTIFIER
GENERATE id_name VALUE yylval
```

IDENTIFIERの値(yylval)をもとにメッセージを生成する。［例終わり］

［例２］形式２

```
assignment -> variable = exp
GENERATE var_id_right WITH(id_ref)
```

Expから伝幡してきたメッセージをもとにして，参照を指示するメッセージを生成する。Expが複数の項からなれば，そのようなメッセージは複数個存在し，その各々に対して生成関数を実行する。［例終わり］

［例３］形式３

```
exp -> exp + term
GENERATE exp_type WITH(left_type,right_type)
  {.............}
```

expとtermの型に基づいて左辺のexpの型を決める。ノード１からの受理メッセージとノード２からの受理メッセージをもとにしてこのノードの型を決め，その情報を上に送る。［例終わり］

返送記述文

形式１。REPLY <target class> FOR <for class> VALUE <value>

形式２。REPLY <target class> FOR <for class> WITH <with class list > <body>

形式１においては<value>としては定数のみを許す。形式２は<for class>に属するメッセージの数だけ<with class list>を使って<body>を評価する。受理メッセージ中の<for class>に属するメッセージの数だけ評価する。

次にPLOに対する意味記述例を２つ示す。

［例４］宣言の有無を調べる

この記述では各宣言から伝播してくる主張メッセージをブロックノードに保存し、変数を利用する代入文等からの質問メッセージに回答する。代入文では左辺での定義としての利用と右辺での参照としての利用を伝播ノードを区別することによりわけている。図２にメッセージ伝播の概要、図３に定義をしめす。各種宣言は各々その宣言に対応する文において、var_id_decl、cons_id_decl、そしてproc_id_declというクラスのメッセージを生成し、そのメッセージはいちばん内側のブロックで受理される。属性文法の属性とは異なり、メッセージは受理されるまでそのまま単一の要素としてノードを伝播していく。受理された時に集合として保存される。代入文で受理されたメッセージid_nameは右辺と左辺とによってid_refとid_defに区別され、それに基づき右辺の質問メッセージvar_id_rightと左辺の質問メッセージvar_id_leftが生成される。生成されたメッセージはその宣言を持つブロックノードを探して親ノードへ伝播していく。ブロックノードの受理メッセージの中に対応する宣言があればYESを値として持つ返送メッセージを返す。もしなければプログラムノードまで伝播し、そこで受理された後、UNDECLを値として持つ返送メッセージが返される。メッセージを生成した代入文はこの返送メッセージを受理し、その値に従ってエラー表示等を行う。［例終わり］

［例５］定義－定義anomalyの検出

この定義では代入文による定義－定義という系列を検出する。変数ノードで生成されたメッセージid_nameは代入文ノードまで伝播し、左辺と右辺に分けられ、各々id_ref、id_defとして受理される。定義のanomalyを調べるためにid_defに基づいてvar_id_leftが生成される．var_id_leftはreversepreorderでノードを走査して定義がないか調べる．もし同じ名前を持つid_refを受理している代入文を見つければそこで終了する．先に同じ名前をid

_defとして持つ代入文があれば定義－定義anomalyである．図４にメッセージの流れを，図５に定義を示す．

```
［例終わり］ id_name     assertion       parent;
var_id_decl     assertion       parent;
cons_id_decl    assertion       parent;
proc_id_decl    assertion       parent;
var_id_left     query       parent      var_id_left_reply;
var_id_right    query       parent      var_id_right_reply;
proc_id_ref     query       parent      proc_id_ref_reply;
id_ref      query       parent      var_id_right_reply;
id_def      query       parent      var_id_left_reply
%
T_program
   begin
      ACCEPT var_id_left;
      ACCEPT var_id_right;
      ACCEPT proc_id_ref;
      REPLY var_id_left_reply FOR var_id_left VALUE UNDECL;
      REPLY var_id_right_reply FOR var_id_right VALUE UNDECL;
      REPLY proc_id_ref_reply FOR proc_id_ref VALUE UNDECL
   end;
T_block
   begin
      ACCEPT var_id_decl;
      ACCEPT cons_id_decl;
      ACCEPT proc_id_decl;
      ACCEPT var_id_left WITH (var_id_decl,cons_id_decl,proc_id_decl)
         {set <- accmes();
            if size(find(target.value,class(var_id_decl,set))) > 0 then
            ACCEPT
            else if size(find(target.value,class(cons_id_decl,set))) > 0 then
                  error("a constant may not be used in LHS")
            else if size(find(target.value,class(proc_id_decl,set))) > 0 then
                  error("a proc_id may not be used in LHS")
          };
         ACCEPT var_id_right WITH (var_id_decl,cons_id_decl,proc_id_decl)
          {set <- accmes();
             if size(find(target.value,class(var_id_decl,set))) > 0 then
             ACCEPT
             else if size(find(target.value,class(cons_id_decl,set))) > 0 then
             ACCEPT
             else if size(find(target.value,class(proc_id_decl,set))) > 0 then
                   error("a proc_id may not be used in RHS")
          };
        ACCEPT proc_id_ref WITH (var_id_decl,cons_id_decl,proc_id_decl)
           {set <- accmes();
              if size(find(target.value,class(var_id_decl,set))) > 0 then
                   error("a proc_id is declared as a variable")
              else if size(find(target.value,class(cons_id_decl,set))) > 0 then
                   error("a proc_id is declared as a constant")
              else if size(find(target.value,class(proc_id_decl,set))) > 0 then
              ACCEPT
            };
        REPLY var_id_left_reply FOR var_id_left VALUE YES;
        REPLY var_id_right_reply FOR var_id_right VALUE YES;
        REPLY proc_id_ref_reply FOR proc_id_ref VALUE YES
     end;
```

```
T_vardec
   begin
     ACCEPT id_name;
     GENERATE var_id_decl WITH (id_name)
   end;
T_condec
   begin
      ACCEPT id_name;
      GENERATE cons_id_decl WITH (id_name)
   end;
T_prodec
   begin
      ACCEPT id_name;
      GENERATE proc_id_decl WITH (id_name)
   end;
T_asign
   begin
      ACCEPT id_name FROM 1 AS id_def;
      GENERATE var_id_left WITH (id_def);
      ACCEPT var_id_left_reply WITH
         {ACCEPT;
          if target.value=UNDECL  then
            error("this variable is not decl")
          };
      ACCEPT id_name FROM 2 AS id_ref;
      GENERATE var_id_right WITH (id_ref);
      ACCEPT var_id_right_reply WITH
         {ACCEPT;
          if target.value=UNDECL  then
            error("this variable is not declared")
         }
   end;
T_call
   begin
      ACCEPT id_name;
      GENERATE proc_id_ref WITH (id_name);
      ACCEPT proc_id_ref_reply WITH
         {ACCEPT;
          if target.value=UNDECL  then
            error("this procedure is not declared")
         }
   end;
T_ident
   begin
      GENERATE id_name VALUE yylval
   end
%
```

図３　宣言の有無の検査の記述

３。プロトタイプとしてのメッセージシステム

環境生成器は仕様より各種データ及びプログラムライブラリを生成する。実行速度を早めるために依存関係を文法に基づくもの（ＲＧ）とプログラムに基づくもの（ＲＰ）の二つに分けている。また意味関数の依存関係に対するサイクルの有無もＲＧで判定する。なおＲＧ、ＲＰ等の詳しい定義は文献［１０］に譲り，本稿では省略する。

アルゴリズム１　文法に基づく依存関係ＲＧの計算

入力：メッセージシステムの記述

出力：ＲＧ及び記述の妥当性

手順：

１．生成規則毎の基本ノード関係を計算する。

２．基本ノード関係から９つの標準伝播関係に対して基本伝播関係を計算する。

３．基本伝播関係より７つの依存関係を計算する。

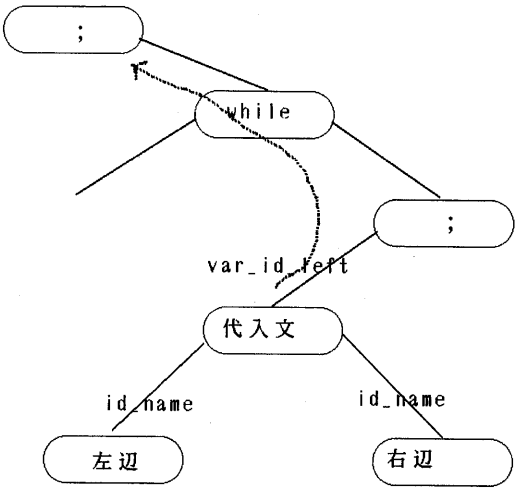

図４．定義－定義anomalyのメッセージの流れ

```
id_name        assertion       parent;
var_id_left      assertion       reversepreorder;
id_ref         dummy           dummy;
id_def         dummy           dummy;
%
T_block
        begin
                ACCEPT var_id_left;
        end;
T_asign
        begin
                ACCEPT id_name FROM 1 AS id_def;
                ACCEPT id_name FROM 2 AS id_ref;
                GENERATE var_id_left WITH (id_def);
                    ACCEPT var_id_left WITH (id_def)
                    {set <- accmes();
                     if size(find(target.value,class(id_def,set)))>0)
                            error("DD anomaly");
                    };
                    ACCEPT var_id_left WITH (id_ref)
                    {set <- accmes();
                     if size(find(target.value,class(id_ref,set)))>0)
                            ACCEPT;
                    };
        end;
T_ident
        begin
                GENERATE id_name VALUE yylval
        end
%
```

図５　定義－定義anomalyの検査の記述

A_1 A_2 P G R_1 R_2 R_3

４．７つの依存関係よりＲＧを計算する。

５．ＲＧ－A_2のサイクルの有無により記述の妥当性を判定する。［終］

メッセージの評価はメッセージシステムの記述とプログラムだけから処理することも可能であるが、ＲＧに基づいてＲＰを計算したほうが効率的である。但しそのためにはＲＧ－A_2にサイクルがないことが要件となり、属性文法における絶対非循環属性文法に対応するもののみに対象が限定される。

アルゴリズム２。全体の評価法

入力：文法より導かれた伝播依存関係ＲＧ
　　解析木Ｔ、依存木ＤＴ

手順：

１．ＲＧをＴに写像して，ローカルな依存関係を計算する。　A_1，G，R_1，R_2

２．ＲＧとＴより，遷移的な依存関係を計算する。　A_2，P，R_3

３．上で計算された依存関係に基づき，依存木ＤＴを作る。

４．ＤＴのindgreeが０である対の集合をｓとする。

```
while　ｓが空でない　do
    ｓの要素を１つ取り出し(n,f)とする。
    case　kind(f)　of
    生成関数：class=f.withclass
        for　すべてのACCEPT(class)の要素　do
            意味関数を実行する。
            メッセージが生成されたなら伝播動作を行う。
    受理関数：class=f.target
        for　すべてのcandidateの要素　do
            受理関数を実行する。
            メッセージが受理されたらACCEPTに加える。そうでなければさらに伝播する。
    返送関数：class=f.for
        for　すべてのACCEPT(class)の要素　do
            返送関数を実行する。
            返送メッセージが生成されたなら返送先に加える。
    endcase
endwhile
```

［終］

アルゴリズム２では伝播するメッセージを関連するするノードで２つの形で保存する。１つは受理メッセージであり、そのノードで受理されたメッセージをクラス毎に保存する。他は保留メッセージであり、ノードまで伝播し、受理を待つメッセージをクラスと伝播してきたノードの種別毎に保存する。保留メッセージの保存はアルゴリズム２では必要ないが、インクリメンタルな評価を行うアルゴリズム３では必要となる。しかし受理される可能性のあるノードでのみの保存であるからDemerらの方法よりも必要なメモリー量は少ない。

アルゴリズム３。インクリメンタルな評価法

入力：修正前の解析木に対する依存木ＤＴ、文法に基づく依存関係ＲＧ，修正された解析木T_3+T_2

手順：

１．T_3+T_2に対してＤＴをインクリメンタルに更新する。

2．ＤＴにおいてT_2及びT_2に直接依存するT_3の要素を評価要とする。
3．indgreeが０のものをリストにつなぐ。
4．｛インクリメンタルに評価を行う｝
while　リストが空でない　かつ　評価要の要素がある　do
　　リストの先頭より要素を一つとる。
　　indgreeが０の要素ができればリストに加える。
　　要素が評価要ならば評価する。
　　受理メッセージが変化すればそれに依存する要素を評価要とする。
　　保留メッセージが作られたらそれに依存する要素を評価要とする。
［終］

アルゴリズム３ではまずT_2に関わる（ノード，関数）対を評価要として，ＤＴのindgreeが０である対を順次再評価していく。保留メッセージが作られたら依存する対を評価要にする。受理メッセージがあり、新たな受理であれば依存する要素を評価要にする。既存の対であればそこでその評価の伝播は終了する。

4。プロトタイプとしての環境生成器、及びその評価

上記のメッセージシステムを核として環境生成器のプロトタイプを試作した。現在のところメッセージシステムの意味関数のbodyの能力から、静的意味の検査の可能な構文エディタ生成器となっている。入力の記述は次の様な構成をとる。

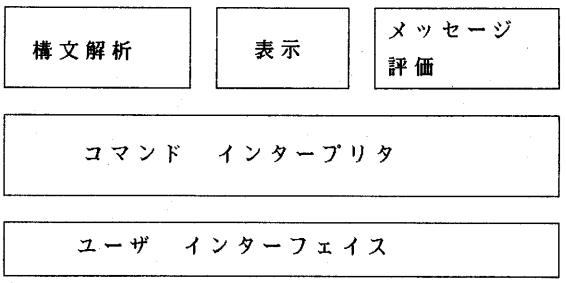

図６　プログラミング環境の概要

%%ＴＯＫＥＮ
　lexの入力形式での語彙の定義
%%ＥＮＤ
%%ＳＹＮＴＡＸ
　yaccの入力形式での構文の定義
%%ＥＮＤ
%%ＳＥＭＡＮＴＩＣＳ
　メッセージシステムによる意味の定義
%%ＥＮＤ
%%ＥＤＩＴ
　基本編集命令とキーとの対応の定義

全体はＵＮＩＸ上で実現し、上記の様な定義からＣの関数および各種の表を生成する。生成される環境は図６の構成をとり、語彙、構文より構文解析及び表示の部分が、意味よりメッセージ評価の部分が、そして編集の部分よりコマンドインタープリタのプログラム／データが作られる。

構文エディタにおいては意味のインクリメンタルな評価に先だって、インクリメンタルな構文解析が必要となる。インクリメンタルな構文解析にも色々な方法が提案[11]されているが、インクリメンタルな意味解析との整合性については調べられていない。今回はゴールを予めいくつか決めておく方法を採用した。

編集命令についてはｅｍａｃｓの様にユーザによるカストマイズができることが望ましいが今回は少数の基本命令に対するキーの割当を指定するというにとどめた。またメッセージ評価の時期については編集操作終了後、構文誤りがなければ即行うこととした。

次にプロトタイプに基づいて各種観点からの評価を行う。

（１）記述の評価

記述は属性文法によるよりも確かに簡潔になる。しかし全体的に決まる伝播順序にもとづいて記述しなければならないため、無理な表現になることもある。標準的に用意した９つの伝播順序では不足の場合もありうる。その場合別の伝播順序を加えることも可能であるが、１）生成器のプログラム自体の変更が必要となる、２）伝播順序に対しては、その伝播順序での次のノードが現在のノードだけから計算できなければならない、という２つの問題がある。１）の点についてはユーザ定義の伝播順序を許す様に生成器を拡張することにより解決される。２）の点はユーザがその様な伝播関数を作ることにより解決される。

属性文法でも同様であるが、記述の能力は１つには意味関数のｂｏｄｙの記述言語の表現能力による。しかしながらメッセージシステムでは評価順序等はｂｏｄｙの内容に関係なく決定される。幅広い記述能力を持たせるためにはｂｏｄｙの記述言語を一般的にする必要があるが、表現の読解性とのトレードオフを考える必要がある。

（２）処理アルゴリズムの評価

アルゴリズム２はプログラムの解析木上の意味関数の数に比例する時間でメッセージを評価する。一般的に意味関数はメッセージの集合を対象とし、例えば受理関数はtarget-classのメッセージがすべて保留メッセージと

なってから、その保留メッセージの集合に対して実行される。

アルゴリズム３は再評価の必要な関数のみ評価すると言う意味で最適である。しかしつぎの様ないくつかの問題点も持っている。１）ＤＴからindgreeが０の（ノード、意味関数）対を抽出する時には評価の要らない対も選ばれてしまう。この様な関数については再評価はしないが、選ぶ事自体が無駄である。２）７つの依存関係のうちA_1，Ｇ，R_1，そしてR_2はＲＧより計算するが、A_2、Ｐ、そしてR_3はＲＧとＴより計算する。そのため関係の数が非常に多くなる。

（３）実現したプロトタイプの評価

実現した環境生成器（構文エディタ生成器）を使ってＰＬ０に対する、意味検査機能を持つ構文エディタを生成した。意味部の記述は図３に示した通りである。文法に基づく依存関係ＲＧは３９個になる。プログラムに基づく依存関係はノード数２９３、行数９１のプログラムで１４７８になる。

アルゴリズム２に基づく全体の評価では対話環境で支障がない程度の立ち上がり時間を示している（ＧＮＵｅｍａｃｓのたち上げと同程度である。）　数百行程度のプログラムでは立ち上がりの時間には問題はないと考えられる。

アルゴリズム３に基づくインクリメンタルな評価については修正箇所にもよるが、一行程度の修正であれば、全体を再評価するのに較べて１０数分の１程度の時間で評価が完了する。

以上よりほぼ応答時間については問題がないことが確認された。但し数百行を越えるプログラムについてはデータ量、応答時間の両面でまだ問題がある。

（４）環境生成器として見たときの問題点

対話的な環境を考えた時にはインクリメンタルな構文解析とインクリメンタル意味解析の２つが必要となる。インクリメンタルな構文解析にはいわゆる照合動作とスキップ動作がある。現在のインクリメンタルな意味解析では照合動作に対応する部分のみ考慮しており、スキップ動作は考慮されていない。またインクリメンタルな構文解析のアルゴリズムは構文解析をできるだけ早く終了させることを目的としており、インクリメンタルな意味解析を最小にするという観点はない。

５。結び

属性文法の一変形としてのメッセージシステムの改良およびそれに基づく環境生成器のプロトタイプについて述べた。現在はまだＰＬ０という非常に小さい文法に対する適用しか試みていない。ＰＬ０について各種の意味記述を行って、構文エディタを生成した範囲では機能性、応答時間等について基本的には問題がないことが確認された。本格的なｐａｓｃａｌやC、そしてＡｄａなどへの適用可能性を調べることが今後必要となる。また４。でものべたように構文と意味の双方の効率的なインクリメンタルな処理法という観点からの考察が今後必要となる。

参考文献

[１] Thomas reps and Tim Teitelbaum:The Synthesizer Generator,Sigplan Notices 19,5 (1984)

[２] R.Medina-Mora,David S.Notkin, and Robert J.Ellison:Aloe User's and Implementators' Guide, The Second Compendium of Gandalf Documentation (May 1982)

[３] 浜田，郡：構文を利用するプログラミングシステムの生成法，電子情報通信学会論文誌 J69-D,11 (1986)

[４] Rolf Bahlke and Gregor Snelting:The PSG System:From Formal Definitions to Interactive Programming Environments,ACM TOPL 8,4 (1986)

[５] D.E.Knuth:Semantics of context-free languages,Math.Syst.Theory, 2,2,(June 1968)

[６] G.M.Beshers,etl:Maintained and Constructor Attributes,Sigplan Notices,20,7 (1985)

[７] G.F.Johnson,etl:A Meta-Language and System for Nonlocal incremental Attribute Evaluation in language-Based editors, 12th ACM POPL (1985)

[８] T.Reps:Generating Language-Based Environments, The MIT Press, 1984

[９] A.Demers,etl:Attribute Propagation by Mesage Passing, sigplan Notices 20,7 (July 1985)

[１０] 海尻：メッセージシステムのインクリメンタルな評価，情報処理学会第３７回全国大会

[１１] 海尻：直構文エディタのためのインクリメンタルパース、電子情報通信学会論文誌　87/7