ソフトウェア基礎論 11－6
（1984. 12. 14）

属性文法に基づく Language-oriented Editorの実現

有山隆史，武田正之，井上謙蔵
（東京理科大学　理工学部　情報科学科）

1．はじめに

プログラム作成の過程では、多くの時間をプログラムの編集に費やす。従ってプログラム言語に特有な構文や意味規則などの知識をエディタに導入することにより、構文・意味誤りの発見やその回復支援を編集の初期に行なうならば、edit-run-edit-run のサイクルを短くし、ソフトウェアの生産性を高めることができる。

本報告では、属性文法により記述された言語の知識をテキスト編集時に利用したLanguage-oriented Editorについて述べる。属性文法[1]は言語の構文（ＢＮＦ）に意味規則（コンパイラでチェックできる範囲）を付加して拡張した文脈自由文法であり、知識が言語の生成規則ごとに独立に表現されるため、これら知識のモジュラリティ・拡張性が高く保たれるという特長をもつ。従来の言語向きエディタの多くは編集対象言語を一つに限定して設計されていたのに対し、本システムでは言語の知識ベースを変更するだけで種々の言語への対応や機能拡張を実現できる汎用性を有する。

また、編集時に発生した構文・意味誤りの回復を支援するので、編集対象言語に関して未熟である利用者にとって本システムはプログラム言語を学ぶためのエキスパートシステムの役割を果たすことができる。

第２章ではシステムの概要について、第３及び４章ではこのシステムを構成する各基本部分について述べる。そして第５章ではシステムへの追加機能を考える。

なお本システムはPrologによりインプリメントされている。

2．システムの概要

Language-oriented Editorは図１に示すように知識作成更新部を外部に持っている。

対象とする言語の属性文法・誤り診断の方法及びその回復規則の記述をシステム管理者が知識作成更新部に与えると、それらの記述は第３章で説明する各知識群に変換される

Language-oriented Editorには自由編集モードと解析編集モードがある。自由編集モードは普通のテキスト編集とまったく同じ使用法であり、解析編集モードは知識に従って利用者にエラーメッセージやその回復箇所を提示しながら、テキスト編集を支援していくモードである。

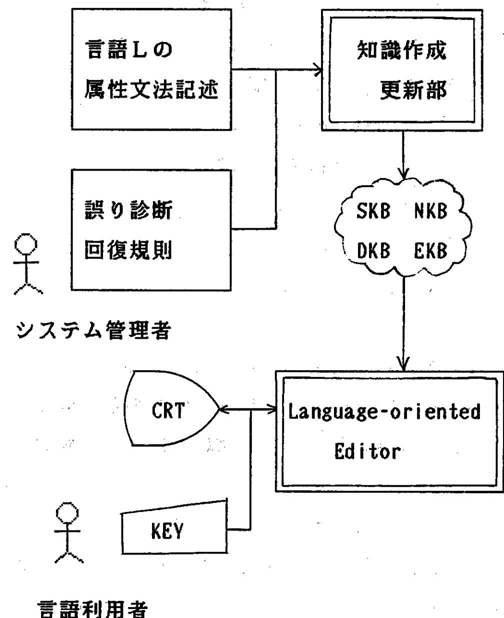

図１　エディタシステムの構成

3．知識作成更新部

3．1　知識の定義方法

対象言語特有の構文・意味や誤りの診断方法をLanguage-oriented Editorの中に知識として貯えておくために、システム管理者はこれらの記述をあらかじめ知識作成更新部に与える必要がある。

知識の定義方法はプログラム言語の形式的定義手法[2,3,4] に基づくが、本システムではKnuthの属性文法を採用している。

その利点として第１章で述べたように知識が言語の生成規則ごとに独立である他に属性文法記述からPrologの一形態であるＤＣＧ(Definite Clause Grammar)［ 5]記述への変換は非常に簡単に行なうことができ、意味定義に用いる属性値の評価がunification 機構により実現できる。

3．2　知識群

エディタが必要とする知識には次に掲げる４つのものがある。

対象言語の構文・意味知識
(Semantic Knowledge Base, SKB)
誤り条件の否定的知識
(Negative KB, NKB)
誤り診断・回復処理知識
(Debug KB, DKB)
誤り状態の知識
(Error KB, EKB)

① 対象言語の構文・意味知識 (SKB)

編集対象言語の構文や静的意味規則から成り、対象言語が与えられると明確に表現される知識である。

② 誤り条件の否定的知識 (NKB)

文法・意味誤りの発生条件を表現している。現在はＳＫＢ中の {} で囲まれたExtra Condition（文献[6]に従い以下ではこれを補強項と呼ぶ）又はＤＫＢに埋め込まれている。

③ 誤り診断・回復処理知識 (DKB)

ＮＫＢのチェックにより誤りが発生した場合、現時点の解析中の環境下でその原因を調べ回復処理を行なうために必要な知識で、誤りの起こった環境（例えば変数のスコープ）の表示、誤りの原因の候補・その理由の提示等を行なう。この知識の階層化により、利用者の能力に応じたデバッグ支援を行なうことができる。

④ 誤り状態の知識 (EKB)

誤りの状態や誤り番号に対応するメッセージを含んでいる。

3．3　知識記述言語の仕様

本来、属性には評価の方向（相続・合成）を付加するが、本報告では評価の方向を除いたＤＣＧ風の記述により知識を表現している。

知識記述言語の仕様はＤＣＧの記述に誤り回復規則をAda の例外処理の文法に基づいて付加したものである。

3．3．1　ＤＣＧとその拡張

ＤＣＧの一般形は

```
head --> body .
```

のように表わし、次の方法で文脈自由文法を拡張している。

① 非終端記号はPrologの変数・整数を除くどのような項でもよい。

② 終端記号は文字列だけを許し、［］で囲む。

③ Prologの述語呼び出し（補強項、ここでは意味処理を行なうSemantic function ）を生成規則の右辺の任意の場所に {} で囲んで書くことができる。

以上の機能に加えて、本システムのためにいくつかの拡張を行なっている。

④ 決定的終端記号の表記

構文上必ずくることがわかっている終端記号を決定的終端記号と呼ぶ。例えばPascalのfor 文の場合、制御変数の後に必ず終端記号':='が来る

べきである。もし解析中に':='がないことを発見すると、システムは利用者に':= expected' のメッセージを与え、修正するかどうかを質問してくる。

このような構文的誤りの発見及びその回復を行なうための記法として、生成規則のその終端記号の前に決定的終端記号であることを表わす$マーク（第3.3.2 節の <d_word> に相当）を導入する。

for 文の例ではシステム管理者は次のような生成規則を与えるとよい。

```
for_statement -->
    [for],
    identifier,
    [$':='],
    expression,to_or_downto,expression,
    [$do],statement.
```

上の例を用いて構文的な回復処理を次に示す。もし解析中に':='がないことを発見した場合、システムはこの誤りに対して利用者に修正するかどうかを質問してくる。この時、利用者が修正すると答えたなら、システムは自動的にその欠けている終端記号（この場合':='）をテキスト中のしかるべき場所に挿入し、再解析を始める。もしそうでない時は自由編集モードに戻り、利用者に修正を任せる。

この回復処理の記述はシステム管理者に委ねずシステム中に埋め込まれている。

⑤　キーワード・特殊キャラクタの判定

現在読み込んだトークンがキーワード又は特殊キャラクタに属するかを判定する組み込み述語を用意している。

・keyword(Token)　Token がキーワードであるとき真、そうでないときは偽となる。

・symbol_char(Token)　Token が特殊キャラクタであるとき真、そうでないときは偽となる。

⑥　トークンの先読み

ＤＣＧはPrologの制御戦略の性格上、バックトラックを用いたトップダウンで縦型探索(depth-first)による構文解析を行なうので探索の空間が必要以上に大きくなることがあり、誤りの発見が遅れる場合が生じる。

それを解決する手段として、生成規則の右辺の先頭に終端記号を置いてガードの役目をさせ、できるだけむだな生成規則の選択を防ぐ他に、必要に応じて１トークンの先読みを行なって、そのトークンの情報を生かして生成規則の選択を決定的にする方法を導入する。

例えばPascalの変数宣言部の解析時に上で考えた問題が起こる。

```
var_decl_part(Env,Env1) -->
    [var],
    var_decl(Env,Env0),var_decls(Env0,Env1).
var_decls(Env,Env) -->
    {lookaheadsym(X),keyword(X)}.
var_decls(Env,Env1) -->
    var_decl(Env,Env0),var_decls(Env0,Env1).
```

という生成規則が与えられていて、入力テキスト

　　<u>var x:integer;</u> begin

の下線部まで解析された時、var_decls の一番目の生成規則は１トークンの先読みを行なって（述語lookaheadsym）そのトークン('begin')がキーワードであることを確認し変数宣言部から抜け出すことにより、解析を一意にすることができる。

３．３．２　知識記述言語の文法

知識記述言語の文法をＢＮＦの形で図２に示す。但し<goals> はPrologの述語呼び出しで、<attributes>は',' で区切られた変数のリストである。その他<empty> を除いて定義されていない生成規則は文字列を表わしている。

```
<KB_definition> ::= <gr_head> --> <gr_body>
                    <error_handler> .
<gr_head> ::= <non_terminal>
<gr_body> ::= ( <gr_body> , <gr_body> )
              | <non_terminal>
              | <terminal>
              | <gr_condition>
<non_terminal> ::=
              <name> ( <attributes> )
<terminal> ::= [ <d_word> <symbol> ]
<d_word> ::= <empty> | $
<symbol> ::= <keyword> | <symbol_char>
<gr_condition> ::= { <goals> }
<error_handler> ::= <empty> |
                    onerror  <handles>
<handles> ::= <handle>
              | <handles> // <handle>
<handle> ::= <errname> => <handle_method>
<handle_method> ::=
             <goals>
            | <handle_method> or <goals>
```

図２　知識記述言語の文法

３．４　知識記述からPrologへの変換例

以下にPASCALの代入文の知識記述とその変換例を示す。

idの属性Nameと環境Env によりそのidのTypeVを述語search_type で求め、expressionの属性TypeE と比較している（Nameが未宣言変数である場合の回復処理について省略している）。もし型が合わなければerr129('type conflict of operands') を発生し、onerror 以下に記述してある回復処理をする（詳しくは4.2 節で後述する）。

・代入文の知識記述

```
assign_statement(Env) -->
  id(Name),
  {search_type(Name,Env,TypeV)},
  [':='],
  expression(TypeE),
  {TypeV==TypeE -> true ; raise(err129)}
 onerror err129 =>
         correct(expression)  or
         correct(id)  or
         return(env_err,id(Name)).
```

・変換後の知識

```
skb(1, * ).
dkb(1,err129, * ,(correct(expression) ;
                  correct(id) ;
                  return(env_err,id(Name)).
```

なお、skb 及びdkb の第一引数は生成規則の番号で、'*' の部分は次の節を表わしている。

```
assign_statement(Env,S0,S3) :-
  id(Name,S0,S1),
  search_type(Name,Env,TypeV),
  terminal(':=',S1,S2),
  expression(TypeE,S2,S3),
  (TypeV==TypeE -> true ; raise(err129)).
```

変換後の非終端記号に２つの引数が加わっている。前者は生成規則に入る前のトークンの列で後者は生成規則から出た後のトークンの列で、差分リストを表現している。なお、誤りの番号及びメッセージは文献[7]による。

３．５　仕様記述のための支援機能

システム管理者が知識記述言語に従って対象言語の設計及び記述を容易に行なうためのユーティリティとして次の四機能があり、システム管理者の手助けをしている。

①生成規則中に一度しか表われない属性名の警告
②キーワード・特殊キャラクタのリスト
③未定義又は未使用の生成規則の表示
④生成規則の呼び出し関係を表わす木の表示

4. Language-oriented Editor

4.1 制御機構

図3はエディタの制御機構を示しており、その動作を以下で説明する。これらはすべて解析編集モードで実行される。

① 原テキストを語い解析して得られたトークン列をＳＫＢに基づきながら前向きに構文・意味解析処理を進める。

② この処理中にＮＫＢで指定された禁止条件が成り立つと、エディタは誤りを検出し、その箇所を利用者に指摘する。そしてそこを修正するか、または次の修正候補の診断に移るかを問い合わせる。

③ 利用者の応答に従って構文木を後戻りしながら誤り回復処理を行なう。ＤＫＢにはこのような誤り診断・回復処理を支援するために必要な誤り発生時点の環境の表示、誤り原因の候補とその理由の提示機能が含まれている。

④ 誤りの回復が終了すると、修正部分を含む最少の構文木の再解析から処理は前向きにＳＫＢを用いて進行する。

この制御機構はPrologを用いたメタ推論により容易に実現されている。[8]。

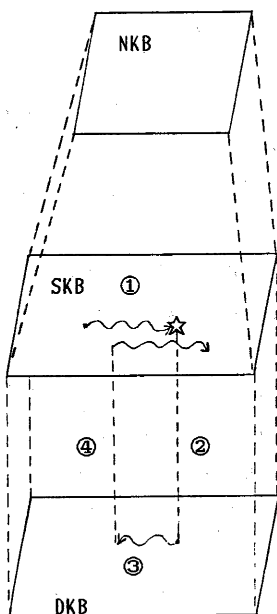

図3　エディタの制御機構と知識の関係

4.2 誤り回復処理の動作

ある生成規則(P --> Q1,...,Qn) における子（Q1,...,Qn)でraise により誤りが発生すると、その親(P) に制御が移り（図4）、親(P) は誤り回復規則(DKB) を呼び出す。このとき３つの回復に関する動作ケースがある。

① 現在使われている生成規則のＤＫＢにより誤りの回復が終わると、処理は4.1 節の④と同様にQ1の解析から続行する。

② ＤＫＢ中にreturnの記述がある場合、現在の生成規則の親に新しい誤りを伝搬する。returnの第一引数は新しい誤り、第二引数には非終端記号のインスタンスを書く。

③ すべてのＤＫＢが利用者の意図するものでなかった場合、自由編集モードになる。

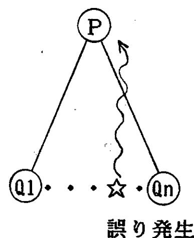

図4　誤りの親への伝搬

解析木

catch

return

raise

図５　誤りの受渡し

ケース②において、次々と親へ伝搬していく誤りをcatch でとらえ、とらえた生成規則から誤りが回復されたものとして前向きに処理を再び始める。ケース①と異なる点は、再解析の途中でreturnの第二引数で与えた非終端記号のインスタンスが起こった時点で第一引数の誤りを発生する。当然システム管理者は、catch でとらえられた生成規則が作る構文木中（図５の斜線部分）にこの非終端記号のインスタンスが必ず存在するようにしなければならない。

ＤＣＧの構文解析木をPrologのメタ述語の計算木(computation tree)と重ね合わせて実行するため、ここで述べる方法は効果的な誤り回復処理の一方法であると考えられる。

４．３　動作例

下の例は変数の二重宣言の場合の回復処理を記述したものである。説明を簡単にするため、

var *x*:integer; *x*:boolean;

のテキストをつかう。

二重宣言の誤りの回復方法として、

①　一番目の変数の修正　(*x*:integer)

②　二番目の変数の修正　(*x*:boolean)

の２つが考えられる。ここで利用者の意図する修正が①の方であったとして、その回復処理を図６（実線は解析が終わった部分で、破線は未解析の部分である）を用いて示す。

まず変数宣言に関する生成規則を次のように与える。

```
var_decl_part(Env,Env1) -->
  [var],
  var_decl(Env,Env0),var_decls(Env0,Env1)
    onerror multi_decl => catch .
var_decl(Env,Env1) -->
  id(Name),
  {check_store(Name,Type,Env,Env1) ->
                    true ; raise(err101)},
  [':'],type(Type),[';']
    onerror
      err101 => ask_correct,edit_text  or
                  return(multi_decl,id(Name)).
id(Name) --> [id(Name)]
  onerror multi_decl =>
            display('Another declaration'),
            edit_text.
```

var_decls の生成規則は3.3.1 の⑥を用いる。述語check_store は変数名とその型を環境中にないことを確認して登録する手続きである。

①　型integer の変数名'*x*' が既に宣言されているので、生成規則idの属性Nameとtypeの属性TypeとをEnv へ登録（この時Typeの値が未定義であることに注意したい）することが失敗してraise を起動させ、err101('identifier declared twice')を利用者に知らせる。ここで修正をするかどうかというシステムからの質問に対して、利用者は型boolean の'*x*' を修正しないと答える。すると一番目の回復処理を諦め、次の生成規則のreturn を実行し、非終端記号のインスタンス(id(*x*)) を

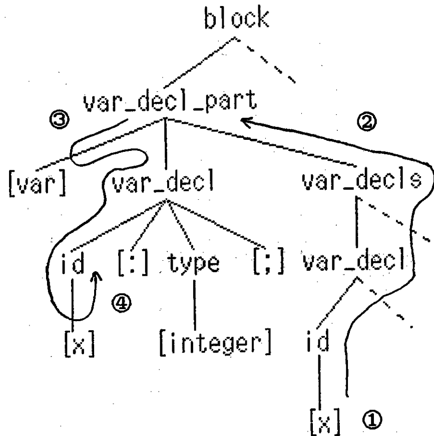

図６　二重宣言の回復の様子

伴って新しい誤り(multi_decl)を親に伝える。

② multi_declのcatch 記述のある生成規則（var_decl_part)まで、誤りを次々と親に伝え解析を遡る。

③ var_decl_part のcatch 文により、この時点から再解析する（入力トークンは'var' になっている。

④ ①のreturnで与えられた非終端記号のインスタンス(id(x))が再解析の途中で表われ、multi_declの誤りが起こる。そのときのトークンは型integer の'x' になっている。idのＤＫＢは'Another declaration' のメッセージを出しテキストを修正するように利用者に指示する。

このようにして、利用者の意図する修正を行なうことができる。

５．システムへの追加機能

より使いやすいユーザインタフェースを備えたシステムを構築するために、現在開発中のマルチウィンドウシステム[9]を用いている。

マルチウィンドウシステムについて考え始めて日が浅いため、まだ十分に生かしきれていないが、現在次の機能を追加している。

システムの管理情報・テキストの編集・誤りの表示及びその回復指示などを各ウィンドウ内で行なう。

また、テキストの任意の位置に誤り発生時だけでなく解析の最中に環境を表示させるために複数のブレークポイントを設定し、その出力先を一つのウィンドウに割りあてることができる。

以上で述べた機能を生かしたPASCALのテキスト編集例を図７に示す。

６．おわりに

システム記述用言語としてのPrologの大きな利点の一つに文献[10]で紹介されている未定義の変数の取り扱いがある。例えば、変数宣言の生成規則の記述で変数の名前が属性としてあがってきた時、二重宣言のチェックをするためにすぐに変数と環境のテーブルを照合する必要がある。従来の手続き型言語では、

「変数名が決まった時に環境のテーブルを二重宣言のためにチェックして、型が決まった時に同じ環境のテーブルに変数名と型を登録する」

を記述しなければならないが、Prologでは

「変数名が決まった時に二重宣言チェックを行なって、もし誤りでないことが分かった時点で変数名とその型（このとき未定義）を環境のテーブルに登録する」

といった記述が簡潔にできる。

また、Prologのインタープリテーションにより構文解析が行なわれるため、特別なパーザが必要でないことも利点のひとつであるが、解析できる生成規則のクラスが限られるため今後ボトムアップの解析[6]について考えたい。

現在、属性方向の情報を誤りの診断に利用せず属性値の評価はunification 機構により行なっているが、今後の課題として属性の依存関係を生かしたより柔軟な制御機構を検討したい。

本システムはPC-9801FのProlog-KABA 上で稼働しており、知識更新作成部は約300 行で、Editor部は約400 行の大きさである。

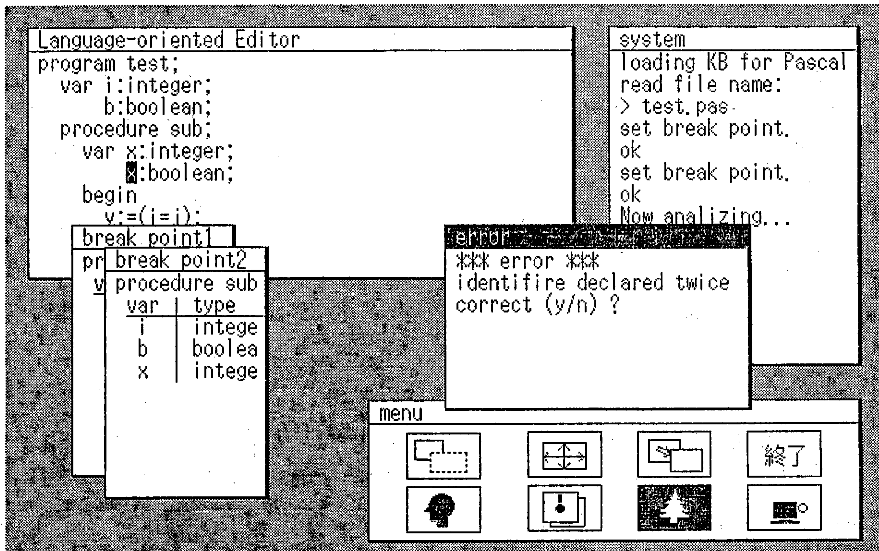

図７　PASCALのテキスト編集例

参考文献

[1]　Knuth,D.E.:
Semantics of Context-Free Languages,
Math. Syst. Th., 2 (1968) 127-145

[2]　Hoar,C.A.R. and Lauer,P.E.:
Consistent and Complementary Formal Theories of the Semantics of Programming Languages,
Acta Informatica, 3 (1974) 135-153

[3]　Marcotty,M.,Ledgard,H.F. and Bochmann, G.V.:
A Sampler of Formal Definitions,
Computing Serveys, 8,2 (1976) 191-276

[4]　Stoy,J.E.:
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,
MIT Press (1977)

[5]　Bowen,D.L.:
DECsystem-10 PROLOG USER'S MANUAL (1981)

[6]　田中穂積：
Prologによる構文解析
Computer today, No.1 (1984) サイエンス社

[7]　Kathleen Jensen and Niklaus Wirth:
PASCAL User Manual and Report (2nd edition)
Springer-Verlag (1974)

[8]　武田：
知識ベースに基づくLanguage-oriented Editorの設計
ICOT WG4　資料 (Jul.1983)

[9]　武田，山田弘，有山：
マルチウィンドウ使用手引き
井上研究室　内部資料 (1984)

[10]　Warren,D.H.D:
Logic for Compiler Writing
Software Practice and Experience,
V10 , (1980) 97-125

[11]　有山，武田，井上：
属性文法に基づくLanguage-oriented Editorの設計
情報処理学会第27回全国大会　7E-5 (1983)