知識ベースに基づく Language-oriented Editor†

武田 正之††

プログラム言語の構文や意味を考慮した編集機能を持つ会話型エディタを設計し，その制御機構を中心にPrologによるインプリメントの一方法を提案した．本エディタはプログラム編集，静的意味解析，誤り診断・回復支援等の機能を持ち，知識ベースを導入することで，異なる言語への適用性やシステムの柔軟性を実現させることを目的としている．対象とする言語の構文ならびに編集時に利用される意味規則は，属性文法に基づき表現される．この属性文法は対象言語の構文に意味規則を付加して拡張した文脈自由文法であり，意味規則が言語の生成規則ごとに独立に表現されるため，モジュラリティ・拡張性が高く保たれるという特徴を持っている．本エディタの制御機構には forward/backtrace 推論を導入することで，構文・意味誤り発見時に，修正候補およびその理由の提示機能が容易に実現できた．またそれに伴い，制御の動きが明確で理解しやすく，さらに制御戦略の変更が容易であるという汎用性を特徴としている．これらの機能は，対象知識である編集対象言語の知識とメタ知識である制御機構を単一の論理プログラミング言語で表現し，両者を融合した形式で利用するメタ推論方式により実現されており，Prologによって複雑な機能を簡潔に記述できることが確認できた．

1. まえがき

ソフトウェア開発段階における仕様記述からプログラム作成への過程では，種々の知識—たとえば，ソフトウェア作成方法論，プログラム言語の構文・意味規則，プログラミング技法，最適化手法等—が必要とされる．これらの知識が計算機で処理できる程度に形式化されるならば，自然言語による表現のあいまい性が除去できると共に，プログラミング支援システムの実現も可能となり，ソフトウェアの生産性を高めることが期待される．

本論文では，ソフトウェア仕様記述からプログラムを合成する段階における，プログラム編集，静的意味解析，誤り診断・回復支援等の機能を持つ会話型エディタを考察対象とし，知識ベースを導入することで異なる言語への適用性，システム柔軟性の実現可能性を検討する．プログラム作成の過程では，エディタはユーザにとって最も身近に使うソフトウェアの一つである．そのためエディタは単なる文字列を編集するための道具ではなく，ソフトウェアの生産を支援するプログラミング環境[1]の一部として考えられる．したがって，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．

† Language-oriented Editor with Knowledge Base by MASAYUKI TAKEDA (Department of Information Sciences, Faculty of Science & Technology, Science University of Tokyo).

†† 東京理科大学理工学部情報科学科

第2章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システムの概要について述べる．属性文法[2]は対象言語の構文(BNF)に意味規則を付加して拡張した文脈自由文法であり，意味規則が言語の生成規則ごとに独立に表現されるため，モジュラリティ・拡張性が高く保たれるという特徴を持っている．ここで，対象言語としてはPascal などの手続き型言語を選び，意味規則としてはコンパイラでチェックできる範囲のものを準備すると，属性文法をプログラムの編集支援に利用した Language-oriented Editor が実現できる[3)~5)]．従来においても，特定のプログラム言語に限定した言語専用エディタと呼ばれるものは存在した．たとえば，Pascal用の MENTOR[6] や PL/I のサブセットのための Cornell Program Synthesizer[7] などがある．前者はソーステキストを直接入力してプログラムの構文だけをチェックするもので，後者はテンプレートを指定してプログラムを合成し，文法のチェックは構文・意味について行っている．しかし，本エディタ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という汎用性を有する．しかも，その知識は属性文法により記述されていることから，意味的な誤りの処理に対して強力なものとなっており，誤り原因箇所を指摘する支援機能を備えている．そのため，編集対象言語に関して未熟である言語利用者にとって，本エディタ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．

本エディタシステムには，知識作成更新サブシステムが付加されている．対象とする言語の属性文法・誤

り診断の方法およびその回復規則をシステム管理者がサブシステムに与えると，それらの規則は各種知識群に変換される．エディタシステムは，それらの知識に従ってユーザにエラーメッセージやその回復箇所などを指示しながら，テキスト編集を支援していく．第3章では，この知識作成更新サブシステムと各種知識群について述べ，第4章では，それらを用いて編集支援機能を実現するための Language-oriented Editor の制御機構について述べる．この制御機構は Prolog を用いたメタ推論により実現されており，制御の動きが明確で理解しやすく，さらに制御戦略の変更が容易であるという汎用性を特徴としている．

2. エディタシステムの概要

本システムは図1に示すように知識作成更新サブシステムと Language-oriented Editor から構成されている．システム管理者はこの知識作成更新サブシステムを用いて対象言語特有の構文や意味規則，そして誤りの診断方法などを入力し，あらかじめ各種知識群を作成しておく．これらの知識の定義方法は属性文法に基づいているが，誤り診断の処理記述に関して拡張が施されている．

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．

3. 知識作成更新サブシステム

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サブシステム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[8)~10)]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．

属性文法を採用した理由は以下の点にある．

（1）知識記述が言語の生成規則ごとに独立である．

（2）属性文法記述から Prolog の一形態である DCG (Definite Clause Grammar)[11)] 記述へは機械的に変換可能である．さらに，属性文法の記述を直接 Prolog 上で操作することも容易である．

（3）意味定義に用いられる属性値の評価が Prolog のユニフィケーション機構により実現できる．

（4）属性はその値の受渡し方向に応じて，相続および合成属性の2種類に分類される．この評価方向の情報を，誤り診断・回復処理の推論に利用できる．

知識作成更新サブシステムは，システム管理者によって与えられた対象言語特有の構文・意味や誤りの診断・回復記述をキーワード・特殊キャラクタと知識群に分類する．Language-oriented Editor は語い解析部とエディタ部から成り，語い解析部はキーワード・特殊キャラクタの情報を使ってソーステキストをトークン列に変換する．トークンには次の4種類，①キーワード，②特殊キャラクタ，③識別子，④定数があり，さらにテキスト中の各トークンの位置を示す情報も付加されている．エディタ部は知識群を読み込み，編集対象言語専用のエディタになる（図2）．

3.2 知 識 群

エディタでは次に掲げるような知識が利用される．

（1）対象言語の構文・意味知識

（2）構文解析知識

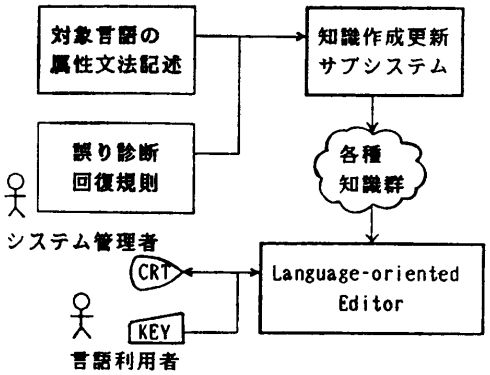


図1 エディタシステムの構成
Fig. 1 An organization of editor system.

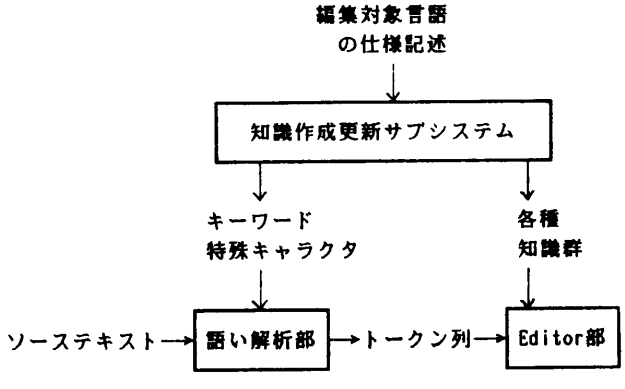


図2 知識作成更新サブシステム
Fig. 2 Knowledge management subsystem.

（3）誤り発生知識
（4）誤り診断知識
（5）誤り回復知識
（6）誤り説明知識

なお現在のシステムでは，上記（1）から（3）までの知識記述を中心に実現されており，（4）から（6）の知識記述はまだ不十分であり，今後検討すべき余地が残されていることを断わっておく．

（1） 対象言語の構文・意味知識

編集対象言語の構文や静的意味規則（すなわち，コンパイラでチェック可能な範囲）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） 構文解析知識

Top-Down または Bottom-Up に構文解析を行うための知識である．現在のシステムでは DCG による Top-Down 解析を採用している．

（3） 誤り発生知識

構文・意味誤りの発生条件を判断するための知識である．また，ユーザにより誤り発生レベルが指定されている場合にはその判断も行う．たとえば，誤り発生レベルを低くおさえると，軽微な誤りを無視して解析処理を先に進めることができる．現在のシステムにおいてこの誤り発生知識は，構文・意味知識中の｛｝で囲まれた Extra Condition（文献[12]に従い以下ではこれを補強項と呼ぶ），または以下に示す誤り診断・回復知識中に埋め込まれている．

（4） 誤り診断知識

（3）により誤りが発生した場合，その時点の解析環境下で誤りの原因を調べ指摘する．この時，複数の原因候補が存在する場合には，誤りやすさの傾向（経験的知識）を反映してそれらの原因に優先順位をつける処理も行う．誤り原因の推論には，属性の依存関係を認識して誤り原因と結果との因果関係を把握する能力が基本的に必要となる．

（5） 誤り回復知識

ユーザまたはシステムにより誤り原因を選択し，テキスト修正を行うための知識である．誤り回復に必要な最小の範囲だけを再評価するための方法論が課題となる．

（6） 誤り説明知識

誤りの起こった環境（たとえば変数のスコープ）を表示して，誤り理由の提示などを行う．

上述の誤り診断や回復知識を階層化することによって，利用者の能力（novice, expert 等）に応じたデバッグ支援を行うこともできる．また，誤り傾向の統計処理による誤り診断方法の改善等の機能を実現するには，ユーザモデルを導入する必要がある．このためには，認知心理学的検討が不可欠であり，この知識の質を決定する要因となるであろう．

```
<KB_definition> ::=
   <head> --> <body_list>
              <error_handler> .
<head> ::= <non_terminal>
<body_list> ::= <body>
              | <body_list> , <body>
<body> ::= <non_terminal>
         | <terminal>
         | <extra_condition>
<non_terminal> ::= <name>
                 | <name> ( <attributes> )
<terminal> ::= [ <symbol> ]
             | [ $ <symbol> ]
<symbol> ::= <keyword> | <symbol_char>
<extra_condition> ::= { <goals> }
<error_handler> ::= <empty>
                  | on_error <handler_list>
<handler_list> ::= <handler>
                 | <handler_list> // <handler>
<handler> ::= <error_name> => <recover>
<error_name> ::= <name>
               | <name> ( <attributes> )
<recover> ::= <goals>
```

図 3 知識記述言語の構文
Fig. 3 Syntax of a knowledge description language.

3.3 知識記述言語

本来，属性には評価の方向（相続・合成）を付加するが，現在のシステム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．

知識記述言語の構文を BNF の形で図 3 に示す．ここで，〈goals〉は Prolog の述語呼出し（連言，選言を含む）で，〈attributes〉は ',' で区切られた項 (term) の並びである．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している．

知識記述は次に示すような枠組みになっている．

```
Head --> Body_1, …, Body_k
on_error
  Error_1 => Recover_1 //
          :
  Error_n => Recover_n.
```

これは，左辺と右辺にそれぞれ Head と（Body_1,

…, Boby_k) を持つ生成規則において, Error_1, …, Error_n なる誤り名に対する回復処理が各々 Recover_1, …, Recover_n であることを表しており, 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. そして, 非終端記号には属性を付加することができる.

(2) 終端記号は文字列だけを許し [ ] で囲む.

(3) Prolog の述語呼出し (補強項) 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.

(4) 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び, 生成規則中のその終端記号の前に決定的終端記号であることを表す $ マークを付加して表す. たとえば, 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. したがって, この ':=' を決定的終端記号として表記しておけば, もし解析中に ':=' がないことを発見した場合, 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) DCG は Prolog の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うので探索の空間が必要以上に大きくなることがある. そして, 構文・意味誤りの発見が遅れる場合が生じ, 適切な誤り回復処理が行えなくなる. この問題を解決する手段としては, 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, できるだけ生成規則の無駄な選択を防ぐ方法や, 必要に応じて1トークンの先読みを行って, 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.

たとえば Pascal の変数宣言部の解析時にこの問題が起こる. 次の生成規則が与えられているとする.

```
var_decl_part(Env, Env3) -->
        [var],
        var_decl(Env, Env2), var_decls(Env2,
        Env3)
    on_error
        multi_decl(var) => catch.
var_decl_part(Env, Env) --> {true}.
var_decls(Env, Env) -->
        lookaheadsym(X), {reserved_word(X)},
var_decls(Env, Env3) -->
        var_decl(Env, Env2), var_decls(Env2,
        Env3).
```

入力テキスト

```
var x : integer ; begin
```

の下線部まで解析された時, var_declsの一番目の生成規則で1トークンの先読みを行って (lookaheadsym) そのトークン ('begin') が予約語であることを確認し (述語 reserved_word), 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.

ここで, 属性 Env, Env2, Env3 は変数名の宣言情報を保存するための環境 (Environment) である. 生成規則 var_decl_part 中の誤り回復処理 (multi_decl) については 4.3 節で述べる.

知識記述は知識作成更新サブシステムによって Prolog 記述に変換され, 次に示すような rule なるデータベースを構成する.

```
rule(RuleNo,
    (Head' :- Body_1', …, Body_k'),
    (Error_1 => Recover_1 //
                :
    Error_n => Recover_n)).
```

この rule の第1引数は生成規則の番号, 第2引数は生成規則の本体, 第3引数は誤り回復処理記述に対応している. 生成規則の本体は次のように変換される.

(1) 非終端記号には2つの引数を付加する. この引数は各々生成規則に入る前のトークンの列と生成規則から出た後のトークンの列に相当しており, 重リストを構成している.

(2) 終端記号は3引数の述語 sys_terminal に変換される. その第1引数は終端記号自体であり, 第2と第3引数はトークン列の重リストである.

(3) 補強項は単に { } をはずす.

(4) 決定的終端記号は3引数の述語 sys_terminal_000 に変換される. その引数の対応は終端記号と同様である.

3.4 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および記述を容易に行うためのユーティリティとして, 知識作成更新サブシステムには次の機能が提供されている.

(1) 生成規則中に一度しか現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) 未定義または未使用の生成規則の表示
(4) 生成規則の呼出し関係を示す木構造の表示
(5) $ 表記の自動付加機能

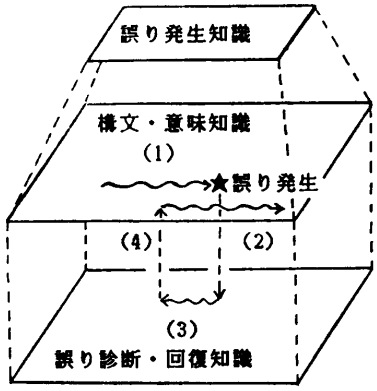


図 4 エディタの制御機構と知識の関係
Fig. 4 Relation between control mechanism and knowledges.

(6) return に対する catch の不足や，適切な catch の位置を指摘する機能

4. Language-oriented Editor

4.1 制御機構

図 4 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．

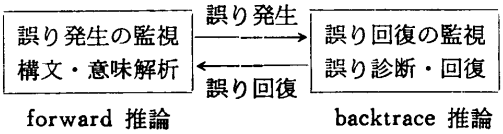
(1) ソーステキストを語い解析して得られたトークン列を対象言語の構文・意味知識に基づきながら前向きに解析を進める．

(2) この解析中に誤り発生知識で指定された誤りの発生条件が成り立つと，エディタは誤りを検出し，その箇所を利用者に指摘する．そして誤り診断・回復知識に基づきながら，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の回復処理を行う．

(4) 誤りの回復が終了するか，または誤り発生レベルが低く設定されると，修正部分を含む最小の構文木の解析から処理は再び前向きに進行する．

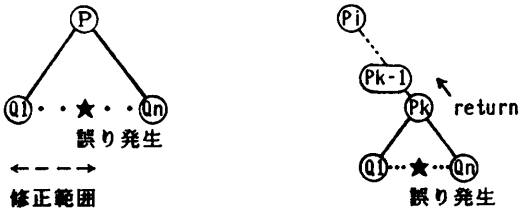
このようにシステムはユーザの編集作業に応じて前向き (forward 推論)/後向き (backtrace 推論) の処理を繰り返していく．この制御機構は Prolog を用いたメタ推論によって容易に実現されている．



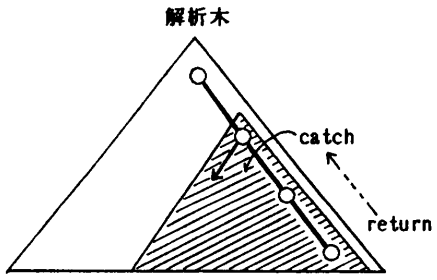
4.2 誤り回復処理の動作

誤り回復処理の動作は次の 3 つに分類される．

(1) 生成規則(P --〉 Q1, …, Q*n*)における子 (Q1, …, Q*n*) で誤りが発生すると，この生成規則内の誤り回復処理の適用を試みる．もし回復が終了した場合，処理は Q1 の解析から続行する (図 5 (a))．



(a) 生成規則内の回復　(b) 誤りの親側への伝搬



(c) 誤りの子側への伝搬

図 5 誤り回復処理の動作
Fig. 5 Three ways of error recovery.

(2) 誤り回復処理中に return の記述がある場合 (すなわち，その生成規則内で誤り回復ができない場合)，現在の生成規則の親に新しい誤りを伝搬する．return の第 1 引数は新しい誤りを表しており，それを解析木上の祖先へ次々と伝える途中で，新しい誤りに関する回復処理の記述があると，その記述を使って誤り回復を行う (図 5 (b))．

(3) 解析木中の祖先のノードでは誤り回復ができない場合，次々と親へ伝搬していく誤りを catch でとらえる．そして，とらえた生成規則から誤りが回復したものとして再び前向きに解析を始める．この再解析の途中で，return または catch の第 2 引数で与えられた非終端記号のインスタンスが起こった時点で第 1 引数の誤りを発生する．このときシステム管理者は catch で誤りをとらえた生成規則が作る構文木中 (図 5 (c) 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 に基づく構文解析木と Prolog のメタ述語の計算木 (computation tree) とを重ね合わせて実行す

* この対応関係は知識作成更新サブシステムによってチェックされる．

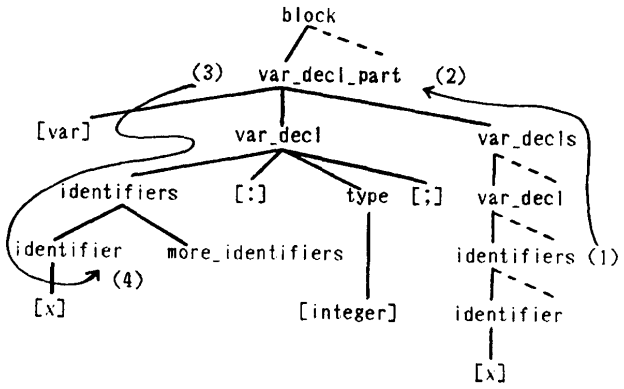


図 6 二重宣言誤りの回復
Fig. 6 A recovery for 'identifier declared twice' error.

るため，本方法は記憶効率の良い誤り回復処理の一方法であると考えられる．

4.3 動 作 例

次の例は変数の二重宣言の場合の誤り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 ;
```

のテキストを用いる．

二重宣言の誤り回復方法としては，

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の方法であったとして，その回復処理を**図 6** (実線は解析が終った部分で，破線は未解析の部分である) を用いて示す．

変数宣言に関する生成規則を次のように与える．なお，誤りの番号およびメッセージは文献[13]による．

```
var_decl(Env, Env2) -->
      identifiers(var, Type, Env, Env2),
      ['$':'], type(Env, Type) ['$':'].
identifiers(Class, Type, Env, Env3) -->
      identifier(ID),
      {enter_id(Class, ID, Type, Env, Env2)},
      more_identifiers(Class, Type, Env2, Env3)
    on_error
      err101 =>/* identifier declared twice */
        ( ask_correct, return(edit);
          search_id([Class0], ID ,_, Env),
          return(multi_decl(Class0),
                 identifier(ID))          ).
more_identifiers(Class, Type, Env, Env2) -->
      [','], identifiers(Class, Type, Env, Env2).
more_identifiers(_, _, Env, Env) --> {true}.
identifier(ID) --> [id(ID)]
   on_error
      multi_decl(_) =>
         writef ('+++ Previous Declaration'),
         return(edit).
identifier(_) --> {raise(err002)}.
      /* identifier expected */
```

生成規則 var_decl_part と var_decls は，3.3 節で示したものを用いる．生成規則 identifiers 中の述語enter_id(Class, ID, Type, Env, Env2) は変数名 ID とそのクラス Class (この場合は変数を表す var) とその型を環境 Env 中にないことを確認して登録する処理をする．ここでクラスは名前の種類を表しており，変数，手続き，型などに分類される．

(1) 生成規則 identifiers において，型 integer の変数名 'x' が既に宣言されているので，生成規則 identifier の属性 ID と生成規則 type の属性 Type とを環境 Env へ登録することが失敗して誤り err101 ('identifier declared twice') が発生する．ここで修正をするかどうかというシステムからの質問 (ask_correct) 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，次の回復処理に移る．そして既に宣言されている 'x' のクラス (Class0=var) を取り出し (search_id)，次の return によって，非終端記号のインスタンス identifier(x) を伴って新しい誤り multi_decl(var) を親に伝える．

(2) multi_decl(var) 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て解析木を遡る．

(3) var_decl_part の catch 記述により，この時点から再解析が行われる．この時入力トークンは '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 identifier(x) が再解析の途中で現れると，multi_decl(var) の誤りが起こる．その時のトークンは型 integer の 'x' になっている．生成規則 identifier の誤り回復処理により，'+++ Previous 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 (return(edit))．

このようにして，利用者の意図する修正をシステムと会話しながら行うことができる．ここで，利用者がシステムの指摘に対して誤った選択を与えた場合，そ

の誤りをシステムが直接認識することは不可能である．たとえば，変数の未宣言による誤りを考えてみると，使用すべき変数名を書き間違ったのか，それともその変数を宣言し忘れたのかという判断には，利用者の意図を理解する必要がある．しかし，誤り原因が修正されない限り，システムは再解析の時点で同じ誤りを発見するので，利用者が誤った選択を与えた場合にも，強制的に再解析を始めることによって回復処理をやり直すことはできる．

4.4 制御機構の Prolog による実現

対象知識とメタ知識を単一の論理プログラミング言語で表現し，両者を融合した形式で利用することで，知識同化／知識調節などの高度なメタ推論を簡単に実現することができる[14),15)]．Prolog による推論とメタプログラミングにおいては，対象知識は Prolog で表現可能なホーン論理の公理集合，そしてメタ知識は Prolog 処理系の使い方に関する知識であり，与えられた知識ベースから与えられた目標を証明する過程を反映している．本エディタシステムにおいても対象知識である編集対象言語の知識とメタ知識である制御機構を融合している．

現在，本システムは PC-9801 の Prolog-KABA[16)] 上で，マウスやマルチウィンドウ機能を持つユーザインタフェースを備えて稼働している．システムの大きさは，Prolog 定義におけるホーン節の数で表すと，知識作成更新サブシステム (381)，Language-oriented Editor (246)，マルチウィンドウ部 (62) である．ここで，編集対象言語としては次に示すような Pascal のサブセットを試みた．

Pascal サブセットの仕様：

- 型：integer，boolean，
- Label, Constant, Type, Function の各宣言なし，
- 意味処理：
  （1）名前の有効範囲を表す静的スコープ規則，
  （2）名前の型衝突チェック，
- 構文規則数：77 ルール，
- 知識記述：113 節，
- Prolog への変換後の知識記述：144 節．

このように非常に短い記述で制御機構を表現できることも特徴の一つであり，ラピッド・プロトタイピングの支援に関して，メタプログラミングは効果があることが分かる．

5. むすび

プログラム言語の構文や意味を考慮した編集機能を持つ会話型エディタを設計し，その制御機構を中心に Prolog によるインプリメントの一方法を提案した．対象とする言語の構文ならびに編集時に利用される意味規則は，属性文法に基づき表現されるため，これら知識のモジュラリティ，拡張性が高く保たれている．また，forward/backtrace 推論を導入することで，構文・意味誤り発見時に，修正候補およびその理由の提示機能が容易に実現できた．これらの機能はメタ推論方式により実現されており，Prolog によって複雑な機能を簡潔に記述できることが確認できた．

今後の課題としては，制御機構をメタ推論の階層構造で構築する方法[17)]，属性の向きや依存関係を考慮した誤り診断・回復処理，および最小部分木の再計算による修正処理などを検討したい．

謝辞 本研究を行う機会を与えていただいた東京理科大学井上謙蔵教授，および，本システムを作成するにあたり多大な協力をいただいた有山隆史氏ならびに小原忍氏に深謝いたします．

参考文献

1) 永田守男，黒川利明（編）：特集：エディタ，情報処理，Vol. 25, No. 8 (1984).
2) Knuth, D. E.: Semantics of Context-Free Language, *Math. Syst. Th.*, Vol. 2, pp. 127-145 (1968).
3) 武田正之：知識ベースに基づく Language-oriented Editor の設計，（溝口，古川（編）：WG4 ワークショップ '83 "知識表現"，ICOT TR-070 (1984)), 20/25 (1983).
4) 有山隆史，武田正之，井上謙蔵：属性文法に基づく Language-oriented Editor の設計，第 27 回情報処理学会全国大会論文集，7E-5 (1983).
5) 有山隆史，武田正之，井上謙蔵：属性文法に基づく Language-oriented Editor の実現，情報処理学会ソフトウェア基礎論，11-6 (1984).
6) Donzeau-Gouge, V. et al.: A Structure-oriented Program Editor: a First Step towards Computer Assisted Programming, Research Rep. 114, INRIA (1975).
7) Teitelbaum, T. and Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Envionment, *CACM*, Vol. 24, No. 9, pp. 563-573 (1981).
8) Hoar, C. A. R. and Lauer, P. E.: Consistent and Complementary Formal Theories of the Semantics of Programming Languages, *Acta*

Informatica, Vol. 3, pp. 135-153 (1974).
9) Marcotty, M., Ledgard, H. F. and Bochmann, G. V.: A Sampler of Formal Definitions, *ACM Comput. Serv.*, Vol. 8, No. 2, pp. 191-276 (1976).
10) Stoy, J. E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge (1977).
11) Bowen, D. L.: DEC System-10 Prolog User's Manual, DAI, University of Edinburgh (1981).
12) 田中穂積：Prolog による構文解析，Computer Today，サイエンス社，No. 1 (1984).
13) Jensen, K. and Wirth, N.: *PASCAL User Manual and Report* (2nd edition), Springer-Verlag, New York (1974).
14) 国藤 進ほか：Prolog による対象知識とメタ知識の融合とその応用，情報処理学会，知識工学と人工知能，30-1 (1983).
15) 国藤 進ほか：論理型言語 Prolog による知識ベースの管理，*Proc. of the Logic Programming Conference '85*, 7. 1 (1985).
16) Prolog-KABA Reference Manual，岩崎技研 (1984).
17) 武田正之：Language-oriented Editor におけるメタ推論，日本ソフトウェア科学会第2回大会，5B-4 (1985).

(昭和61年10月20日受付)
(昭和62年 9 月 9 日採録)

武田 正之（正会員）

昭和28年生．昭和52年東京理科大学理工学部電気工学科卒業．昭和57年東京工業大学大学院博士課程（電子物理工学専攻）修了．同年東京理科大学理工学部情報科学科助手となり，現在同大学講師．工学博士．著書（共著）に「Prolog とその応用2」，総研出版（昭和60年）がある．プログラミング言語の意味，証明論，知識情報処理に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．