演算子文法用汎用構造エディタ

佐藤　裕幸,　　坂井　公
Hiroyuki Sato　　Kou Sakai
(財)新世代コンピュータ技術開発機構

1．はじめに

エディタはユーザがコンピュータ・システムを利用する上で、不可欠な要素であり、今や、対話的システムのユーザがコンピュータ端末の前に座っている時間の大半をエディタが占めるに至っている。　エディタこそ、人間と機械とが協調して仕事をしている状態の最も端的な例であるという意味でマンマシン・インタフェースの典型である。　近年のマンマシン・インタフェースのアイデアで、エディタに取り入れられないものはないといえるし、実際、多くのアイデアは、まず、エディタに応用することを考えて提案されている。

そして、これらはすべて、エディタをより汎用により使いやすくするという目標に向かっているといえよう。

我々は、この汎用性と使いやすさを基本的目標として汎用構造エディタEdipsを逐次型推論マシンψ(PSI)上に開発した。　本稿では、このEdipsの特徴および機能について動作例をまじえて紹介する。

2．Edipsの特徴

Edipsは、以下のような特徴を持っている。

○マルチ・ウィンドウを使用したスクリーンエディタである

マウスによるカーソルの指定、メニューによるコマンドの指定が行え、テキストの表示領域、コマンドのエコーバック領域などをマルチ・ウィンドウによって分割している。

○構造エディタである

編集されるデータの中に構造を持ち込んだ構造指向型エディタである。　これにより以下のような利点が得られる。

★人間の考えている論理的な単位と編集操作の単位との一致を図れる。

★従来のエディタでは、文字，単語，行等のように構造の大きさ毎に類似のコマンドをたくさん持つことがあったが、それらを一元化できる

★編集対象を巨視的に眺めたり、微視的に眺めたりするホロフラスティング機能を利用できる

○編集対象の構文規則に依存せず汎用性がある

テキストを構造に変換する構文解析ルーチン（トランスデューサ）がエディタ本体から完全に独立しており、また、ユーザが文法を定義できるようになっているため、編集対象の構文規則に依存しない構造エディタとなっている。

○表層構造を重視したテキスト・エディタである

人間の編集作業は、本来持っている構造（深層構造）よりもそれを表現しているテキストとしての構造（表層構造）に着目して行われることが多い。

3．Edips の機能概要

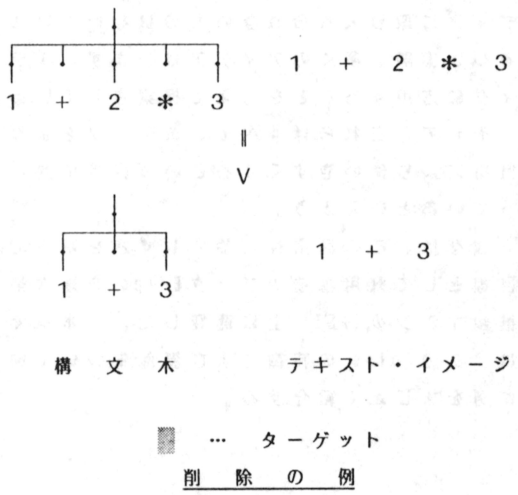
Edips には、次の二つの編集モードがある。

・構造編集モード
　しかるべき構文規則にのっとって編集を行う、いわゆる構造エディタ
・文字列編集モード
　編集対象の構造を無視した文字列としての編集を行う、いわゆるテキスト・エディタ

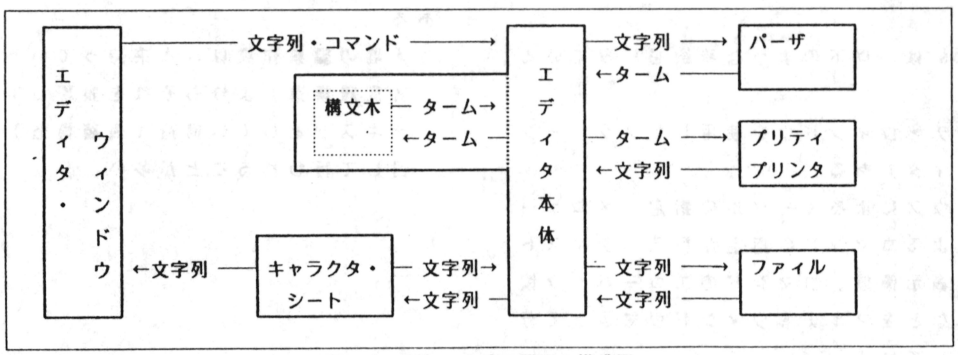
一部の構造エディタ特有のコマンドを除いてどちらのモードも同じコマンド体系になっている。　表－１（次ページ）はEdips のコマンド一覧表である。

Edips およびその周辺の構成を図－１に示す。　パーザ，プリティプリンタは、それぞれ文字列と構造（ターム）との相互変換を行っている。　キーボードやマウスからの入力はウィンドウから送られ、パーザによりタームに変換される。　構造編集モードでの編集作業はエディタ本体が内部的に保持している構文木を操作することにより行われる（右の削除の例）。　キャラクタ・シートとは、編集中のテキストを文字列として保持しているシートであり、構文木の変更された部分を文字列としてキャラクタ・シートに送ってやることによりウィンドウの表示が変わる。　文字列編集モードの時は構文木での編集は行わず、直接キャラクタ・シートを操作することにより編集作業を行っている。

いわゆる構文エディタのテキスト入力方式は、テンプレート方式とパージング方式に分けることができる。　テンプレート方式では、システムが構文にそったテキストのテンプレートを提供しユーザがその中身を埋めていくことによりテキストを入力していく。　一方、Edips でのテキストの入力方法はパージング方式となっており、新たにテキストを入力するには文字列編集モードに入り、終了すると再び構造編集モードに入ることにより入力した部分が構文解析される。　Edips での編集操作はすべてターゲットとよばれる領域に対して行われる。　ターゲットとは一般のエディタにおけるカーソルに対応するが、連続した複数の構文要素（文字列モードの時は文字列）をターゲットとすることができる。



削　除　の　例



図－１　Edips とその周辺の構成図

コマンド名	機　　能
target	ターゲット範囲を指示する
scope	操作範囲を限定する（スコープ範囲を指示する）
add scope	操作範囲を追加する（スコープ範囲を追加する）
brother	ターゲット（スコープ）範囲を弟方向へ移動する
meta brother	ターゲット（スコープ）範囲を兄方向へ移動する
parent	ターゲット（スコープ）範囲を親へ移動する
child	ターゲット（スコープ）範囲を子供へ移動する
meta child	ターゲット（スコープ）範囲を子供へ移動する
find	与えられたパターンを順方向へ検索する
meta find	与えられたパターンを逆方向へ検索する
kill	ターゲット範囲の構造（文字列）を順方向へ削除する
meta kill	ターゲット範囲の構造（文字列）を逆方向へ削除する
duplicate	ターゲット範囲の構造（文字列）を与えられた位置の弟へ複写する
meta duplicate	ターゲット範囲の構造（文字列）を与えられた位置の兄へ複写する
move	ターゲット範囲の構造（文字列）を与えられた位置の弟へ移動する
meta move	ターゲット範囲の構造（文字列）を与えられた位置の兄へ移動する
substitute	ターゲット範囲の構造（文字列）を与えられた構造（文字列）に置換える
enbracket	与えられた括弧でターゲット範囲を囲む
debracket	ターゲット範囲の括弧をはずす
yank	削除された構造（文字列）を復活させる
passivate	ターゲットのない状態にする
visible level	ホロフラスティング・レベルを設定する
change grammar	定義文法を変更する
change mode	編集モードを変更する
visit file	指定されたファイルからテキストを読み込む
write file	指定されたフィアルへテキストを書き込む
to whiteboard	ターゲット範囲の構造（文字列）をホワイト・ボードへ書き込む
from whiteboard	ホワイト・ボードから文字列を読み込む
change font	文字フォントを変更する
set window size	ウィンドウのサイズを変更する
menu	メニューによりコマンドを指定する

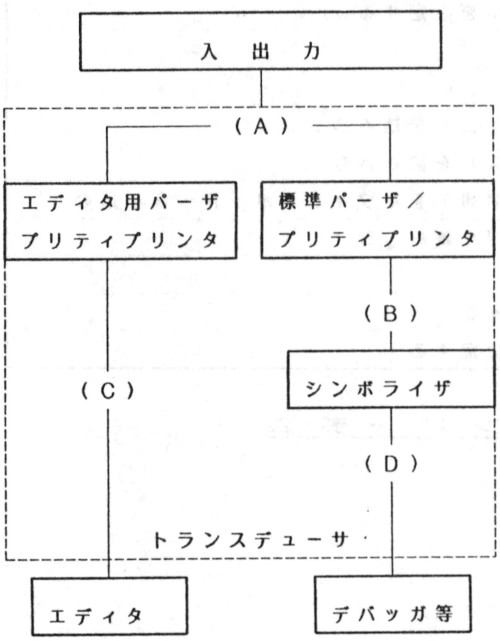
表－1　コマンド一覧表

4．トランスデューサ

トランスデューサは、PSI 内部のデータ構造とその表示イメージとの相互変換を行うユーティリティで、ユーザが定義した文法に従って構造を持ったデータを構文解析したりプリティプリンティングしたりする。

トランスデューサは、エディタのみではなくデバッガやコンパイラ等の他のサブシステムからも使用される。　図－２の破線に囲まれた部分が、トランスデューサの主な構成要素であり、それぞれ次のような変換を行う。

- 標準パーザ／プリティプリンタ
  表示イメージと構造化データ間の変換
- エディタ用パーザ／プリティプリンタ
  表示イメージとエディタ用構造化データ間の変換
- シンボライザ
  構造化データとPSI 内部表現間の変換

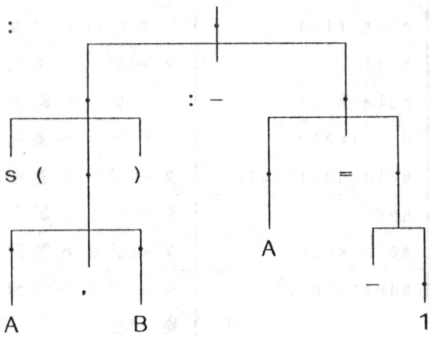


図－２　トランスデューサの構成

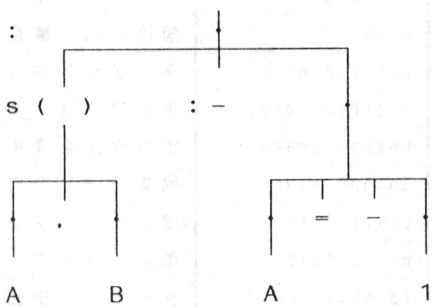
（A），（B），（C），（D）は、それぞれ表示イメージ，構造化データ，エディタ用構造化データ，PSI 内部表現である。　図－３にそれぞれのデータ形式の具体例を示す。

（A）：　s（A，B）　：－　A＝－１

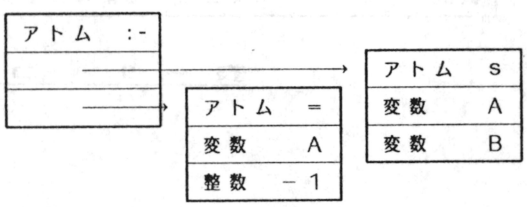
（B）：



（C）：



（D）：



図－３　データ表現

5．文法

ユーザ定義の文法により構文解析を行うシステムとして提案されているものは、文脈自由文法に基づいたものが多いようであるが、我々は、PSI のシステム言語であるESP との親和性とエディタ・コマンドとの対応のとりやすさから、演算子順位文法を採用した。文法は構文要素(token) を定義する部分と演算子順位を定義する部分からなる。

構文要素の定義には、BNF を用いる。 構文規則の右辺には正規表現が書けるようになっているが、再帰的規則は許さないので、定義できる構文要素は正規集合になる。 浮動小数点データを表現する構文要素の定義を例として挙げる。

```
digit ::= "0" | "1" | … | "9"
real := digit,{digit},".",{digit}
      ["^",["+"|"-"],digit,{digit}]
```

右辺で正規表現のために使用できる記号とその意味は次の通りである。

- －　（負符号）　…差集合
- ，　（コンマ）　…連接
- |　（縦線）　…和集合
- ”…”（引用符で囲む）…終端記号列
- […]（大括弧で囲む）…省略可
- { … }（中括弧で囲む）…０個以上の繰返し
- （…）（小括弧で囲む）…補助記号
- 英小文字で始まる英数字列…非終端記号
- JIS コード　…そのコードを持った文字
- . .（ピリオド二つ）…二つのJIS コード間のすべての文字

構文要素定義の右辺にでてくる非終端記号は、中間的な文字列を定義するためで、それ自身は、構文要素にはならないが、再びBNF によって定義される。 この中間的な文字列には、システムであらかじめ次のものが用意されている。

- graphic character
  ディスプレイ上に他の文字とはっきり区別されるように表示され、一文字のスペースだけを占有する文字。
- delimiter character
  コンマ，コロン，ピリオド等の、文字列を二つに分離するために使われる文字
- formatting character
  タブ，改行等の書式を制御するために主に使われる文字
- digit　０～９
- lowercase letter　ａ～ｚ
- uppercase letter　Ａ～Ｚ
- kanji　漢字
- quoted(JISコード，文字列)
  指定された文字列を０個以上並べたものの中の指定されたJIS コードの文字を２個の同じ文字で置換え、さらに両側をその文字で囲んだ文字列

構文要素は、以下の６種類に分類される。

- アトム　…　定数、変数等の単独で意味をもつデータ
- 前置演算子…　負符号の"-" 等、引数の前に置かれる１項演算子
- 間置演算子…　減法の"-" 等、２つの引数の間に置かれる２項演算子
- 後置演算子…　階乗記号"!" 等、引数の後に置かれる１項演算子
- 左括弧　…　"(" , begin 等
- 右括弧　…　")" , end 等

これらのうち、左括弧と右括弧を除いた４つのグループ間には、重なりがあってもよいが、左括弧や右括弧として定義されたものが他のグループにも属している場合は、構文解析の結果は保証されない。

前置，間置，後置各演算子には、引数との結合に関する優先順が定義される。 DEC-10 Prolog等では、各演算子に自然数を割り当て、その大小で優先順を定めるが、これは人間には直感的にわかりにくいので、我々は次のような記法を採用した。

```
relation> operator
(operator <operator) <relation
```

relationとoperatorは、ユーザが定義した構文要素名である。 1行目はrelationが左にoperatorが右にある場合はoperatorが先に引数と結合することを意味する。 2行目は左右が逆転しても、operatorが優先することと、operator同志では左のものが優先することを意味する。

例えば、構文要素定義で”－”がoperatorに”＝”がrelationになっていれば、図－2（A）のA＝－1は（B）のように解析される。

```
string_definition

    alphanumeric        ::= lowercase_letter |
                            uppercase_letter |
                            digit ;
    special_character   ::= "@" | "#" | "$" | "^" | "&" |
                            "*" | "+" | "-" | "`" | "?" |
                            "\" | "=" | "_" | ":" |
                            "." | "/" | "<" | ">" | "|" ;
    delimiter           ::= "(" | ")" | "|" | "[" | "]" |
                            "[" | "]" | ";" | "," ;
    digit_sequence      ::= digit, {digit} .

token_definition

    atom        ::= lowercase_letter , { alphanumeric | "_" } ;
    atom        ::= quoted( #"'" , { graphic_character } )    ;
    atom        ::= ( special_character , { special_character } )
                    -- "." ;
    comment     ::= "%" , { graphic_character } , new_line ;
    real        ::= digit_sequence, "." , { digit } ,
                    [ "^" , [ "+" | "-" ] , digit_sequence ] ;
    integer     ::= digit_sequence,
                    [ "^" , [ "+" | "-" ] , digit_sequence ] ;
    nil         ::= "[]" | "{}" ;
    cut         ::= "!" ;
    semicolon   ::= ";" ;
    leftvector  ::= "{" ;
    rightvector ::= "}" ;
    leftlist    ::= "[" ;
    rightlist   ::= "]" ;
    comma       ::= "," ;
    string      ::= quoted( #"""" , { graphic_character } ) ;
    meta_variable ::= "$" , { alphanumeric | "_" } ;
    variable    ::= ( uppercase_letter | "_" ) ,
                    { alphanumeric | "_" } ;
    fullstop    ::= "." .
```

図－3 ＥＳＰの標準文法

エディタを使用している時は、人間は、論理的な構造よりも、目に見える構造に着目しがちである。 また、プリティプリンティングを行う場合、深い構造を忠実に表現しようとすると大きなスペースを必要とする。 そこで、演算子順位は、エディタやプリティプリンタには一般に無視されるようになっている。 エディタやプリティプリンタにも意識させたい優先順を指定するには">","<" の代りに">>","<<" を用いる。

例えば、":-"が logicalsymbol で

logicalsymbol > > relation

と定められていれば、図－２（Ａ）はエディタ用パーザによって（Ｃ）のように解析される。

図－３は、PSI のシステム記述言語ESP の標準文法定義である。

```
operator definition

    atom(   real | integer  | cut  | string | nil | meta_variable ) ;

    prefix( "class"         | "instance"    | "local"       |
            "nature"        | "before"      | "after"       |
            "component"     | "attribute"   |
            "mode"          | "public"      | ":-"          |
            "?-"            | "spy"         | "nospy"       |
            "\+"            | "+"           | "-"           |
            ":"             |
            "#"             | comment       ) ;

    infix(  "has"   | ":-"  | "-->" | semicolon   |
            "->"    | comma | cut   |
            "+"     | "-"   | "/\"  | "\/"  |
            "*"     | "/"   | "<<"  | ">>"  | "|"   |
            "="     | "is"  | "=.." | "=="  | "\=="  |
            "@<"    | "@>"  | "@=<" | "@>=" | "=:=" |
            "=\="   | "<"   | ">"   | "=<"  | ">="   |
            ":="    | "mod" | "^"   | "#"   ) ;

    parenthesis( leftvector, rightvector ) ;
    parenthesis( leftlist,   rightlist   ) .

precedence_definition

    "class"
    >> "has"  > (semicolon  > "instance" | "local"  > semicolon )
         > "before" | "after"
         > "nature" | "component" | "attribute"
    >> ":-"   | "?-"  | "-->"
         >   "mode" | "public"  > ("->"  > "->")
         >   (comma | "|"     >   comma | "|")
    >> ("spy" | "nospy" | "\+"  > "spy" | "nospy" | "\+")
         > ( "="     | "is"  | "=.." | "=="  |
             "\=="   | "@<"  | "@>"  | "@=<" | "@>=" | "=:=" |
             "=\="   | "<"   | ">"   | "=<"  | ">="  | ":="  )
         > "+" | "-" | "/\" | "\/"
         > "*" | "/" | "<<" | ">>"
         > "mod"  > ("^" | cut  > "^" | cut)
         > ("#" | ":"  > "#" | ":")     ;

    "#" | ":" <  "^" | cut <  "mod"
        <    ("*" | "/" | "<<" | ">>"    <    "*" | "/" | "<<" | ">>")
        <    ("+" | "-" | "/\" | "\/"    <    "+" | "-" | "/\" | "\/")
        <    ( "="   | "is"  | "=.." | "=="  | "\==" |
               "@<"  | "@>"  | "@=<" | "@>=" | "=:=" | "=\=" |
               "<"   | ">"   | "=<"  | ">="  | ":="  )
        <    "spy" | "nospy" | "\+"
    << comma | "|" <  "->" <  "mode" | "public"
        <    ":-"  | "?-"  | "-->"
    << "nature" | "component" | "attribute"
        <  "before" | "after" <  semicolon       .
```

6．編集操作単位

Edips は文字列編集モードでない限り、常に文法的に正しい（構文規則に適合した）ものを編集対象として保持する。 ところが、ユーザが編集対象に対して削除や複写などの操作を行うと文法的に正しくない状況に陥る場合がある。 例えば、 a＋b／c で b のみを削除しても a＋／c となってしまい、文法的に正しくなくなってしまう。

Edips ではそれを避けるために操作が行なわれた編集対象の前後の要素を同時に削除や複写することがある。 これを説明するために、構文要素列からなる部分構造を次のように分類する。

・アトム類…アトムと同様に扱われる。
アトム，前置演算子，左括弧のいずれかで始まり、アトム，後置演算子，右括弧のいずれかで終わる。

・前置類 …前置演算子と同様に扱われる。
アトム，前置演算子，左括弧のいずれかで始まり、前置演算子，間置演算子のいずれかで終わる。

・後置類 …後置演算子と同様に扱われる。
間置演算子，後置演算子のいずれかで始まり、アトム，後置演算子，右括弧のいずれかで終わる。

・間置類 …間置演算子と同様に扱われる。
間置演算子，後置演算子のいずれかで始まり、前置演算子，間置演算子のいずれかで終わる。

削除を行う場合、文法的に正しく保つためには、削除の単位を前置類，または後置類にしなければならない。 従って、Edips ではテキストの削除を行う場合、その単位が前置類，または後置類になるように操作対象（ターゲット）を拡張する。

例 ▒ を削除（▒ はターゲット）

・操作対象がアトム類の場合

a＋▒b▒／c ＝左へ削除＝＞ ▒a▒／c
a＋▒b▒／c ＝右へ削除＝＞ a＋▒c▒

・操作対象が間置類の場合

a＋b▒／▒c ＝左へ削除＝＞ a▒＋▒c
a＋b▒／▒c ＝右へ削除＝＞ a＋b

挿入，複写の単位も削除と同様に前置類，または後置類でなければならない。

例 ▒ を z の左又は右へ複写

a＋▒b▒！／c ＝左へ＝＞ b！／z
a＋b▒！▒／c ＝左へ＝＞ b！／z
a＋b▒！▒／c ＝左へ＝＞ b！／z
a▒＋▒b！／c ＝右へ＝＞ z＋b
a＋▒b▒！／c ＝右へ＝＞ z＋b
a＋b▒！▒／c ＝右へ＝＞ z！
a＋b▒！▒／c ＝右へ＝＞ z／c
a▒＋▒b！／c ＝左へ＝＞ a＋z

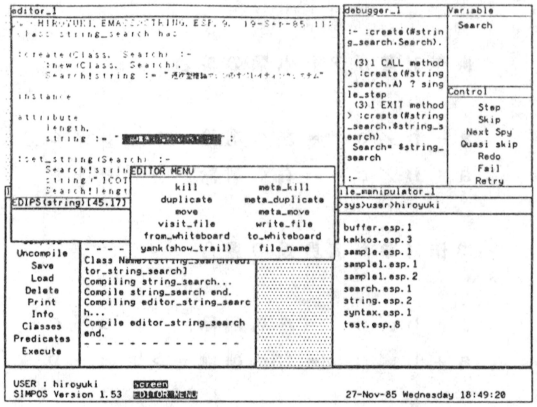
括弧をつける単位はアトム類でなければならない。

例 ▒ の部分に括弧をつける

a▒＋▒b！／c ＝＞ （a＋b）！／c
a＋▒b▒！／c ＝＞ a＋（b）！／c
a＋b▒！▒／c ＝＞ a＋（b！）／c
a＋b▒！▒／▒c ＝＞ a＋（b！／c）

7．動作例

Edips の動作例を以下に示す。



これは、SIMPOS（PSI のオペレーティング・システム）全体のハード・コピーであり、Edips はデバッガ、ライブラリアン、ファイル・マニピュレータ等と共に使用される。

```
editor_1
class
    test_program has
        :test_method( ... ) :-
        test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [ ... ], number( # ),
            { ... }
        )
         :- !,
        test_program(Object, Tail,  ... , Process)
    ; end
EDIPS(struct)[56,17] sample1.esp
ARG>mouse_click
```

Edips のウィンドウは、テキスト表示部分、コマンドのエコーバック部分からなっている。テキストはウィンドウの横幅に合わせてプリティプリンティングされており、…と#はホロフラスティング機能による構造の細部が省略された表示である。 ▒▒ で囲まれた部分がターゲットであり、編集作業はすべてこの部分に対して行われる。

ここで、ターゲット範囲の構造を [ ] の弟（右側）へ複写すると次のようになる。

```
editor_1
class
    test_program has
        :test_method( ... ) :-
        test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [ ... ], number( # ),
            { ... }
        )
         :- !, test_process(Object, Program, Process),
        test_program(Object, Tail,  ... , Process)
    ; end
EDIPS(struct)[56,17] sample1.esp
ARG>mouse_click
<duplicate>
```

ターゲット範囲の構造はアトム類なので、そのまま複写しても文法的におかしくなってしまう。 そのためターゲットの弟のカンマ（，）もいっしょに複写される。

次にターゲットの範囲を現在の３番目の子供へ移動して、その範囲を削除する。

```
editor_1
class
    test_program has
        :test_method( ... ) :-
        test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [ ... ], number( # ),
            { ... }
        )
         :- !, test_process(Object, Program, Process),
        test_program(Object, Tail,  ... , Process)
    ; end
EDIPS(struct)[56,17] sample1.esp
ARG>3
<child>
```

```
editor_1
class
    test_program has
        :test_method( ... ) :-
        test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [ ... ], number( # ),
            { ... }
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail,  ... , Process)
    ; end
EDIPS(struct)[56,17] sample1.esp
<child>
<kill>
```

ここでも同様に文法を正しく保つためにターゲットの弟のカンマがいっしょに削除されている。

次に　＄Ｘ－＄Ｙ　という構造パターンを順方向に検索する。　検索パターン中の　＄　で始まる構文要素はメタ変数として扱われ（図－３のＥＳＰ標準文法で定義されている）、それはどんな構文要素ともマッチする。

```
editor_1
class
    test_program has
        :test_method( ... ) :-
        test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [ ... ], number( # ),
            { ... }
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail, ...., Process)
    ; end

EDIPS(struct)[56,17] sample1.esp
ARG>$X - $Y
<find>
```

パターンが見付かったが、その部分の表示が省略されている。　そこで、可視レベルを上げてみる。

```
editor_1
class
    test_program has
        :test_method(Object, Argument, Program, Process)
         :- test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [Head|Tail],
            number(Process_no), {| ... |}, Process)
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail, Head-Process_no,
            Process
        )
    ; end

EDIPS(struct)[56,17] sample1.esp
ARG>5
<visible-level>
```

メタ変数は、それとマッチした構造を次回の検索コマンドまで値として保持している。

従って、今はそれぞれ　＄Ｘ＝Ｈｅａｄ　＄Ｙ＝Ｐｒｏｃｅｓｓ＿ｎｏとなっている。

次に　－　の両側の構造（ＨｅａｄとＰｒｏｃｅｓｓ＿ｎｏ）を入替える。　それには、メタ変数の保持している値を用いてターゲットの範囲を　＄Ｙ－＄Ｘ　に置換えることによって行える。

```
editor_1
class
    test_program has
        :test_method(Object, Argument, Program, Process)
         :- test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [Head|Tail],
            number(Process_no), {| ... |}, Process)
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail, Process_no-Head,
            Process
        )
    ; end

EDIPS(struct)[56,17] sample1.esp
ARG>$Y - $X
<substitute>
```

次にターゲット範囲にfunctor（関数子）ｎｕｍｂｅｒをつける。　Edipsでは、関数子を括弧と同様に扱っているので、括弧付けのコマンドによりそれを行う。

```
editor_1
class
    test_program has
        :test_method(Object, Argument, Program, Process)
         :- test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [Head|Tail],
            number(Process_no), {| ... |}, Process)
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail, number( ... ),
            Process
        )
    ; end

EDIPS(struct)[56,17] sample1.esp
ARG>number(_)
<enbracket>
```

次に、新たにテキストを入力するために文字列編集モードに入る。

```
editor_1
class
    test_program has
        :test_method(Object, Argument, Program, Process)
         :- test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [Head|Tail],
            number(Process_no),
            [(Head-Process_no*3), Process]
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail,
            number(Process_no-Head), Process
        )
    ; end
EDIPS(string)[56,17] sample1.esp
<enbracket>
<change-mode>
```

文字列編集モードに入るとすべてのレベルまで表示するようにプリティプリンティングされる。　文字列編集モードでは、一般のテキスト・エディタと同様に編集対象の構文規則を気にせずにテキストを入力することができる。

```
editor_1
class
    test_program has
        :test_method(Object, Argument, Program, Process)
         :- test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [Head|Tail],
            number(Process_no),
            [(Head-Process_no*3), Process]
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail,
            number(Process_no-Head), Process
        )
    ;
test_process(X, X) :- ! ;
test_process(X, X, X) :- ! ; end
EDIPS(string)[56,17] sample1.esp
```

テキストの入力が終り再び構造編集モードに入ると、テキストが構文解析されウィンドウにプリティプリンティングされる。もしテキストに構文エラーがあった場合は、その場所をターゲットにしてユーザに知らせ、構造編集モードには入らない。

```
editor_1
class
    test_program has
        :test_method(Object, Argument, Program, Process)
         :- test_process(Object, Program, Process),
        test_program(Object, Argument, Program, Process)
    ; local
        test_program(Object, [Head|Tail],
            number(Process_no),
            [(Head-Process_no*3), Process]
        )
         :- !, test_process(Object, Process),
        test_program(Object, Tail,
            number(Process_no-Head), Process
        )
    ; test_process(X, X) :- ! ;
    test_process(X, X, X) :- ! ; end
EDIPS(struct)[56,17] sample1.esp
```

以上、Edips の一部のコマンドの動作例を紹介した。

8．おわりに

以上のようにEdips はPSI 上で稼動中であるが、現在は構造エディタとしての拡張・改良とテキスト・エディタとしての拡張・改良の二本立てで進められている。　それぞれ以下のようになっている。

・構造エディタ

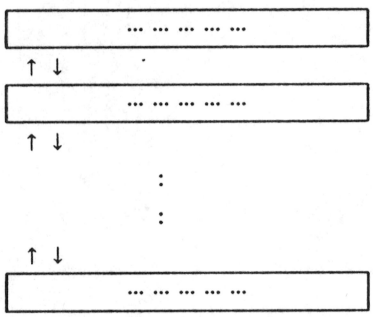
プログラムやテキストを編集するのではなく主に構造をもったデータを編集するようにコマンド体系を改良し、又扱いやすさの関係から文法を演算子順位文法から文脈自由文法に変更する。

現在採用している演算子順位文法は、仕組みが比較的単純なわりにユーザが必要な構造を定義するためだけならば、十分に記述力が高いのでそれほど複雑な文法を書かなくてもよいという長所がある。しかし入力時にかなり複雑なチェックをしてそれにあわないデータを最初から構文解析時にはねつけるようなことをするには向かない。又、先に述べたテンプレート方式による文法ガイダンスをしようとする場合にそのために必要な情報を十分

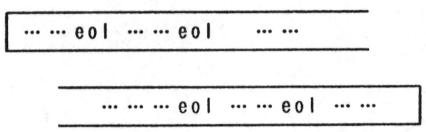
に書込む為の枠組みがない。もう一つ重大な事に演算子順位文法には文脈自由文法の場合のように明確な定義や信頼するにたる理論が存在していないようである。

文脈自由文法を書かせるというのは演算子順位文法に比べ最初のユーザの負荷が大きくなるという問題が有るが、先に述べたような入力チェックは結局はユーザがプログラムの中に書かなければならないことであり、全体としては作業量が増えるということはないだろうし、構文解析時にある種のエラーを発見できるということは、有用な場合も多い。

我々は文脈自由文法を採用することによりパージング方式とテンプレート方式がうまく共存したような新しい入力方式を考えている。又、構文解析のためには、Prologの機能を十分に利用できるDCG とそのためのボトムアップ構文解析方式（BUP ）があるので、それを文脈自由文法のために機能縮小して採用することを考えている。



図－４　キャラクタ・シートの構造



図－５　emacs のbufferの構造

・テキスト・エディタ

プログラムやテキストを編集するのに用いる。　現在の文字列編集モードは先に述べたようにキャラクタ・シートを操作することにより編集を行っている。　このキャラクタ・シートでのテキストの持ち方は、図－４のように各行が双方向のリンクで結ばれている構造をしている。　この構造は、図等を編集している時には適しているが、一般にテキストを編集している時には図－５に示したemacsのbufferのようにeol （end of line または改行コード）も一般の文字として扱いテキスト全体が一本の文字列のようになっている方が編集しやすいと思われる。　そこで、Edipsでもキャラクタ・シートのような構造ではなく、emacs の　bufferのような構造を採用することにした。　また、ユーザが好みに合せて自分用のエディタを作れるという点でemacs の設計方針を、現在世の中で最も多く使われているエディタであるという点でemacsのコマンド体系やユーザ・インタフェースを参考にしてテキスト・エディタを開発していく予定である。

＜参考文献＞

・佐藤他：SIMPOSのプログラミング・システム　－エディター第30回情処全国大会4E-3
・坂井他：SIMPOSのプログラミング・システム　－トランスデューサー同上4E-4