# Prolog系言語ADL(1)： ADLの概要とインタプリター

長澤, 勲
九州大学中央計数施設

古川, 由美子
九州大学中央計数施設

https://doi.org/10.15017/1468074

出版情報：九州大学大型計算機センター広報. 16 (3), pp.252-279, 1983-05-25. 九州大学大型計算機センター
バージョン：
権利関係：

# Prolog 系言語 ADL (1)
## ーADLの概要とインタプリターー

長澤　勲*　古川　由美子*

目　　次



## 1. はじめに

Prolog [ 1 ] という名の述語論理に基づくプログラミング言語が注目を集めている．Prologは，パタンマッチング，自動バックトラッキングといった基本機構を持ち，Lispに相当する汎用記号処理言語であり，自然言語処理，データベース問合せ処理，数式処理等の応用分野において，実用的な言語とするべく現在多くの研究が行われている．

筆者らは，電子回路や機械の設計の分野において，知的設計支援システム（ICAD）を実現する研究を行っている．この研究の過程で開発したのがADL（A Designer's Langnage ）である．ADLはPrologを核として機能拡張を行った言語であり，他の分野の研究者にとっても有用な言語であると考えられ，公開することにした．

本システムの主な考慮点は，

(1) F4／Lisp上に実現されており，Lispのプログラミング環境を全て使用できる．また，FORTRANとの結合機能が強化してあり，配列やロードライブラリが使用できる．

(2) インタプリタは機械語で書かれており，種々の応用システムを実現するのに一応の性能を有し

* 九州大学中央計数施設

ている．

(3) エディタ，デバッガ等のプログラム作成支援ツールが充分整備されている．

(4) 横式探索を可能にするオープンリスト，伝播法による拘束条件解法，対象指向プログラミングを可能とするフレーム等の拡張機能があること．

である．

本稿では，ADLの基本機能について述べ，(4)の拡張機能については別稿で解説する予定である．Prologに関する解説は省かせていただいたので，初心者は参考文献[ 2 , 3 , 4 , 5 ]を参照されたい．

なお，本システムの開発の一部には，九州大学大型計算機センターの開発課題を使用させていただいたことを付記する．

## 2. システムの構成と使用例

本システムは，インタプリタ，デバッガ，エディタ，パッケージ管理から成り，図2，1のように構成されている．

インタプリタはPrologインタプリタの機能を拡張したものであり，ADLで書かれたプログラムを実行する．

デバッガは，プログラムの実行中にアテンションを受けるか，あらかじめ指定された状態になったとき呼出され，プログラムのデバッグや解探索の制御に使用される．

エディタは，手続きの編集を行う．またエディタはプログラムの実行中，動的に呼び出され，手続きやオープンリストの編集，ゴールの参照などに使用される．

パッケージ管理は，プログラム・ライブラリの管理を行う．ライブラリには，LispのS式で表現されたソースライブラリ，機械語に翻訳されたコードを含むチエックポイントライブラリ，および他言語で書かれたプログラムのロードライブラリの三種類がある．本システム自身は，チエックポイントライブラリとソースライブラリ形式(例題)で提供されている．

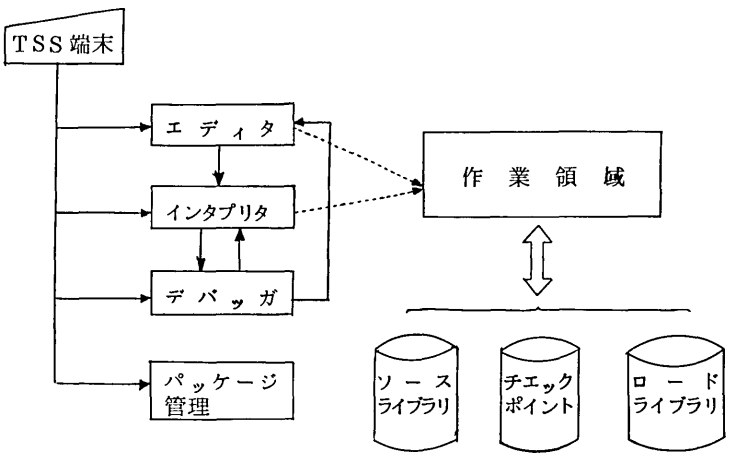



図2.1 システム構成

図2.2～図2.4にシステムの起動，ハノイの塔の問題を例とした，エディタの使用例，プログラムの実行例を示す．図中，下線の付された部分は端末からの入力を示す．

```
LOGON TSS ID/PASSWORD S(2048)
      :
      :
READY
EX ADL
LISP(V01L05) START
INIT:FCS=153600,BPS=30720,PDS=102400
I RESTORE(ADL)
  = NIL
  TIME 293MS.
I FREE(ADL)
  = ADL
  TIME 12MS.
      :
      :
```

（注）

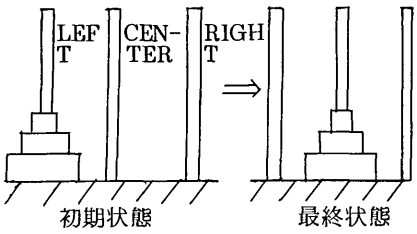
- 下線の付された部分が端末からの入力．
- リージョンサイズに2048以上を指定する．
- ADL・CLISTを各ユーザが作成する．
- IはLISPの入力促進記号
- ADL・CLISTの作成において，1行目のTACLIB，SSL 2は必要なければ省略可能，2行目のSHRは必ず指定すること．

ＡＤＬ・ＣＬＩＳＴの内容例

```
00010 LIB ('LIB.ADL.LOAD' 'SYS1.FORTLIB' 'SYS1.TACLIB' 'SYS1.SSL2')
00020 ALLOC  DA('LIB.ADL.CHKPT') F(ADL) SHR
00030 ADL  'MSG=2,FCS=150K,BPS=30K,PDS=100K'
```

図2.2　システムの起動方法

例題：ハノイの塔




```
I EDIT(HANOI EXAM)
   OK-ET
R PA
   ((LOCAL O O)
   /$
   (- (EVAL (READ) ¥N)
      - (MOVE ¥N LEFT CENTER RIGHT))
   )
R OK
  = HANOI
  TIME       17MS.
I EDIT(MOVE EXAM)
   OK-ET
R PA
   ((LOCAL O O)
   /$
   (+ (MOVE 1 ¥A ¥B ¥C) - / - (ANS MOVE ¥A TO ¥B))
   (+ (MOVE ¥N ¥A ¥B ¥C)
      - (¥M := ¥N - 1)
      - (MOVE ¥M ¥A ¥C ¥B)
      - (ANS MOVE ¥A TO ¥B)
      - (MOVE ¥M ¥C ¥B ¥A))
   )
R OK
  = MOVE
  TIME       39MS.
```

（注）

図中のＲは入力促進記号，／＄はカーソル記号を表わす．

詳細は5章参照

図2.3　ハノイの塔の手続きとゴールの表示

デバッグモードによる実行

トレース指定なし

```
I DEBUG NIL
    DEBUG
R RUN(HANOI EXAM)
      ADL (VERSION A-3) START
R 3
   (MOVE LEFT TO CENTER)
   (MOVE LEFT TO RIGHT)
   (MOVE CENTER TO RIGHT)
   (MOVE LEFT TO CENTER)
   (MOVE RIGHT TO LEFT)
   (MOVE RIGHT TO CENTER)
   (MOVE LEFT TO CENTER)
   (S  =  27 SU =  14 FU =  3 EV =  15 B  =  0)
   NIL
```

全トレース指定

```
R TR-A
R RUN(HANOI EXAM)
      ADL (VERSION A-3) START
    GIVEN  ( L= 0  EXAM )
    (- (EVAL (READ) (3 . ¥N)) - (MOVE (3 . ¥N) LEFT CENTER RIGHT))

    ((1) CALL (EVAL (READ) (3 . ¥N)))
         (EVAL 1 1)
         (/ 1)
R 3
    ((1) EXIT (EVAL (READ) 3))
    ((3) CALL (MOVE 3 LEFT CENTER RIGHT))
         (MOVE 2 0)
    ((4) CALL (:= (63 . ¥M) (¥¥¥-¥ 3 1)))
         (:= 1 0)
    ((5) CALL (EVAL (¥¥¥-¥ 3 1) (63 . ¥M)))
         (EVAL 1 1)
         (/ 5)
    ((5) EXIT (EVAL (¥¥¥-¥ 3 1) 2))
    ((4) EXIT (:= 2 (¥¥¥-¥ 3 1)))
    ((7) CALL (MOVE 2 LEFT RIGHT CENTER))
         (MOVE 2 0)
    ((8) CALL (:= (161 . ¥M) (¥¥¥-¥ 2 1)))
         (:= 1 0)
    ((9) CALL (EVAL (¥¥¥-¥ 2 1) (161 . ¥M)))
         (EVAL 1 1)
         (/ 9)
    ((9) EXIT (EVAL (¥¥¥-¥ 2 1) 1))
    ((8) EXIT (:= 1 (¥¥¥-¥ 2 1)))
    ((11) CALL (MOVE 1 LEFT CENTER RIGHT))
         (MOVE 1 1)
         (/ 11)
    ((13) CALL (ANS MOVE LEFT TO CENTER))
         (EVAL)
    (MOVE LEFT TO CENTER)
    ((13) EXIT (ANS MOVE LEFT TO CENTER))
    ((11) EXIT (MOVE 1 LEFT CENTER RIGHT))
    ((14) CALL (ANS MOVE LEFT TO RIGHT))
         (EVAL)
    (MOVE LEFT TO RIGHT)
    ((14) EXIT (ANS MOVE LEFT TO RIGHT))
    ((15) CALL (MOVE 1 CENTER RIGHT LEFT))
         (MOVE 1 1)
         (/ 15)
    ((17) CALL (ANS MOVE CENTER TO RIGHT))
         (EVAL)
    (MOVE CENTER TO RIGHT)
    ((17) EXIT (ANS MOVE CENTER TO RIGHT))
    ((15) EXIT (MOVE 1 CENTER RIGHT LEFT))
    ((7) EXIT (MOVE 2 LEFT RIGHT CENTER))
    ((18) CALL (ANS MOVE LEFT TO CENTER))
         (EVAL)
    (MOVE LEFT TO CENTER)
    ((18) EXIT (ANS MOVE LEFT TO CENTER))
    ((19) CALL (MOVE 2 RIGHT CENTER LEFT))
         (MOVE 2 0)
    ((20) CALL (:= (431 . ¥M) (¥¥¥-¥ 2 1)))
         (:= 1 0)
    ((21) CALL (EVAL (¥¥¥-¥ 2 1) (431 . ¥M)))
         (EVAL 1 1)
         (/ 21)
```

```
     ((21) EXIT (EVAL (¥¥¥-¥ 2 1) 1))
     ((20) EXIT (:= 1 (¥¥¥-¥ 2 1)))
     ((23) CALL (MOVE 1 RIGHT LEFT CENTER))
          (MOVE 1 1)
          (/ 23)
     ((25) CALL (ANS MOVE RIGHT TO LEFT))
          (EVAL)
     (MOVE RIGHT TO LEFT)
     ((25) EXIT (ANS MOVE RIGHT TO LEFT))
     ((23) EXIT (MOVE 1 RIGHT LEFT CENTER))
     ((26) CALL (ANS MOVE RIGHT TO CENTER))
          (EVAL)
     (MOVE RIGHT TO CENTER)
     ((26) EXIT (ANS MOVE RIGHT TO CENTER))
     ((27) CALL (MOVE 1 LEFT CENTER RIGHT))
          (MOVE 1 1)
          (/ 27)
     ((29) CALL (ANS MOVE LEFT TO CENTER))
          (EVAL)
     (MOVE LEFT TO CENTER)
     ((29) EXIT (ANS MOVE LEFT TO CENTER))
     ((27) EXIT (MOVE 1 LEFT CENTER RIGHT))
     ((19) EXIT (MOVE 2 RIGHT CENTER LEFT))
     ((3) EXIT (MOVE 3 LEFT CENTER RIGHT))
     (S  =  27 SU =  14 FU =  3 EV =  15 B  =  0)
     NIL
R Q
  = END_DEBUG
  TIME      140MS.
```

図2.4　ハノイの塔の実行例

## 3. 言語仕様

ここではADLの基本部分の言語仕様を説明する．ADLの構文は付録1を参照していただきたい．以下の章で※印の付された部分は，拡張機能に属し，別稿で解説する予定である。

### 3.1　ADLの対象

ADLはLisp上のシステムであるのでシステムの対象はすべてLispのアトムおよびリストで表現される．

(1) 変数

$，？，%，で始まる文字アトムおよび？で終る文字アトムはADLの変数である．？，%，が前置された変数を入力変数，？が後置された変数を出力変数と呼ぶ．－（P　$X　$X）は（＋（P　A　B））とマッチしないが　_（P　_　_）はマッチする．

‘_’（アンダーライン）は出現のたびに異る変数を表わす略記法に用いる．

変数のマッチングの規則を図3.1に示す．

例：　$X　$THIS－IS－A－VARIABLE　?INPUT　OUTPUT?

| 定義側＼呼出し側 | 非変数 | Y? | %Y | ?Y | $Y |
|---|---|---|---|---|---|
| 非変数 | U | X | X | ↑ | ↑ |
| X? | X | X | X | X | ↑ |
| %X | X | X | X | X | ← |
| ?X | ← | X | X | X | X |
| $X | ← | ← | ↑ | X | ↑ |

U：統一化
↑：上（呼出し側）へ代入
←：左（定義側）へ代入
X：失敗

図3.1　マッチングの規則

(2) 定　数

変数以外のLispアトムはすべて定数である．文字アトムは手続名としても使用される．

例：１９８３　１９．３０　ＣＯＮＳＴＡＮＴ　>=　ＡＤＤ

(3) リスト

構造のあるデータやプログラムはリストで表現する．リストの要素は変数，定数，リストである．また，リストの残りの部分を示すのにLispのそれに類似のドット記法が使用できる．たとえば，（Ａ　Ｂ　．＄Ｘ）はＡ，Ｂで始まる任意のリストとマッチ可能なリストを示している．

例：（ＡＰＰＥＮＤ　（Ａ　Ｂ）（Ｃ　Ｄ）　＄Ｘ）　（Ａ　＄Ｌ　．＄ＬＬ）
　　（Ａ　Ｂ　（Ｃ　Ｄ））

(4) プログラム

ADLのプログラムは手続き呼出しと手続き定義の集合からなる．手続定義は手続頭部と手続き本体に分けられ，手続き本体は手続き呼出しの列で構成されている．

例：

```
(a) (- (EVAL (READ) ¥N)                          ---------- 手続呼出 }
       - (MOVE ¥N LEFT CENTER RIGHT))            ---------- 〃      } ゴール

(b) (+ (MOVE 1 ¥A ¥B ¥C) - / - (ANS MOVE ¥A ¥B))
(c) (+ (MOVE ¥N ¥A ¥B ¥C)                         ---------- 手続頭部 }
       - (¥M := ¥N - 1)                           -------- 手続呼出 }
       - (MOVE ¥M ¥A ¥C ¥B)                       -------- 〃       } 手続本体 } 手続定義
       - (ANS MOVE ¥A TO ¥B)                      -------- 〃       }
       - (MOVE ¥M ¥C ¥B ¥A))                      -------- 〃       }
```

## 3.2 インフイックス・オペレータ

数式等を表現するのにインフイックス・オペレータが使用できると便利である．たとえば，(=(+(＊２　＄Ｘ）＄Ｙ）１０）の代わりに（２＊＄Ｘ＋＄Ｙ＝１０）と書くことができる．ADLではこの目的のために後述するパッケージ単位にインフイックス・オペレータを宣言する．インフイックスオペレータの宣言はエディタで編集することができる．オペレータの宣言は，（〈オペレータ名〉〈優先順位〉〈ＬＲ／ＲＬ〉）の形式で与える．優先順位は，かっこを省略した場合の結合順位である．LR/RLは同一のオペレータが連続したとき，かっこでくくる方向（それぞれ左から右，右から左）を示している．インフイックス・オペレータで書かれたリストはシステム内部ではプレフィックスで書かれたリストに変換され，同一視して取扱われる．システムで使用している記号（＋，－，／など）とインフイックス宣言が両立しない場合はインフイックス宣言をはずして使用する．

エディタ呼出形式：

ＥＤＩＴ（〈パッケージ名〉　ＩＮＦＩＸ）

宣言例：
```
((& 0 RL)
 (** 6 LR)
 (/ 5 LR)
 (* 5 LR)
 (¥¥¥- ¥ 3 LR)
 (¥¥¥+ ¥ 3 LR)
 (= 2 LR)
 (¥¥¥< ¥ 2 LR)
 (¥¥¥> ¥ 2 LR)
 (¥¥¥<=¥ 2 LR)
 (¥¥¥>=¥ 2 LR))
```

例：（＄Ｘ＋＄Ｙ＝２　＆　＄Ｘ＞＝０　＆　＄Ｙ＞＝０）は

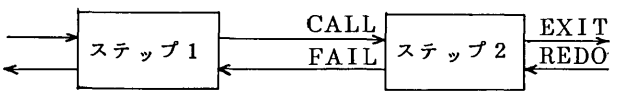
（＆（＝（＋　＄Ｘ　＄Ｙ）　２）（＆（＞＝　＄Ｘ　０）（＞＝　＄Ｙ　０））

と同じである．

### 3.3 プログラムの実行と手続き的意味

Prologのプログラムには宣言的意味と手続き的意味があると主張されている．しかし，現実のプログラムでは後者が多用されるので後者の説明だけを行っておく．なおこれは文献（２）にもとづいている．


通常の手続型プログラム




Prolog型プログラム

図3.2　通常の手続き型プログラムとProlog型のプログラム

通常の手続型プログラムは一命令ごとに実行されるが，失敗や後戻りの概念はない．一方，Prolog型のプログラムは実行に失敗すると自動的に後戻りし，以前実行したステップを再実行する．再実行に成功（別解を出力）すると再び次のステップを実行する．これを図示したものが図3.2である．ステップの最初の試行をCALL，成功出口をEXIT，再試行入口をREDO，失敗出口をFAILと呼ぶ．ボックス内の流れは，

ＣＡＬＬ－ＥＸＩＴ，ＣＡＬＬ－ＦＡＩＬ，ＲＥＤＯ－ＥＸＩＴ，ＲＥＤＯ－ＦＡＩＬ

の四種類である．

以下にリストを2つの部分リストに分けるプログラムの実行例を示す.

```
ゴール (- (APPEND ¥X ¥Y (A B)) - (QUERY ¥X ¥Y))
```

手続き[ 1 ] `(+ (APPEND NIL ¥X ¥X))` …空リストとリスト$Xを結合したものは$Xである.

[ 2 ] `(+ (APPEND (¥U . ¥X) ¥Y (¥U . ¥Z))`
`- (APPEND ¥X ¥Y ¥Z))` } …リスト$Xと$Yを結合したものが$Zならば($U. $X)と$Yを結合したものは($U. $Z)である.

図中,($3. $X)($5. $Y)等は$X. $Yの改名された変数を示している. まず, レベル(1)のゴール(APPEND ($3. $X)($5. $Y)(A B))でAPPEND手続きの[1]が呼ばれる. この手続きは手続本体がないからすぐEXITし, ($3. $X)($5. $Y)にそれぞれNIL, (A B)が代入される. このとき, 手続き[ 2 ]は未試行として記録される.

①
```
((1) CALL (APPEND (3 . ¥X) (5 . ¥Y) (A B)))
     (APPEND 1 1)
              ↑ ↑
              |  └──── 未試行手続きの数
              └─────── 実行中の手続きの番号
((1) EXIT (APPEND NIL (A B) (A B)))
```

次に(QUERY NIL (A B))が実行されるが, これはシステムプリミティブであり, (NIL (A B))を印字して, 失敗する. そこでレベル(1)のAPPENDが再試行(バックトラック)され, 今度はAPPENDの[ 2 ]を実行する. $U, ($3. $X), $Y. $Zにそれぞれ A, (A. (29. $X)), ($5. $Y), (B)が代入される.

②
```
((2) CALL (QUERY NIL (A B)))
     (EVAL)
(NIL (A B))
((2) FAIL (QUERY NIL (A B)))
((1) REDO (APPEND NIL (A B) (A B)))
     (APPEND 2 0)
```

APPEND[ 2 ]は手続き本体があるから, (APPEND (29. $X)($5. $Y)(B))を実行する. APPEND[ 2 ]を未試行として記録し, APPEND[ 1 ]を実行すると, (29. $X), ($5. $Y)にそれぞれNIL, (B)が代入され, ($3. $X), ($5. $Y)は(A), (B)となる.

③
```
((2) CALL (APPEND (29 . ¥X) (5 . ¥Y) (B)))
     (APPEND 1 1)
((2) EXIT (APPEND NIL (B) (B)))
((1) EXIT (APPEND (A) (B) (A B)))
```

以下同様に繰返し, 3通りの解が得られたことになる.

```
((3) CALL (QUERY (A) (B)))
     (EVAL)
((A) (B))
((3) FAIL (QUERY (A) (B)))
((1) REDO (APPEND (A) (B) (A B)))
```

```
((2) REDO (APPEND NIL (B) (B)))
     (APPEND 2 0)
((3) CALL (APPEND (58 . ¥X) (5 . ¥Y) NIL))
     (APPEND 1 1)
((3) EXIT (APPEND NIL NIL NIL))
((2) EXIT (APPEND (B) NIL (B)))
((1) EXIT (APPEND (A B) NIL (A B)))
((4) CALL (QUERY (A B) NIL))
     (EVAL)
((A B) NIL)
((4) FAIL (QUERY (A B) NIL))
((1) REDO (APPEND (A B) NIL (A B)))
((2) REDO (APPEND (B) NIL (B)))
((3) REDO (APPEND NIL NIL NIL))
((3) FAIL (APPEND (58 . ¥X) (5 . ¥Y) NIL))
((2) FAIL (APPEND (29 . ¥X) (5 . ¥Y) (B)))
((1) FAIL (APPEND (3 . ¥X) (5 . ¥Y) (A B)))
(S  =  3 SU =  5 FU =  1 EV =  3 B  =  4)
FAIL
```

### 3.4 基本述語

ADLで用意した基本述語を機能別に述べる．記述中の〈英小文字列〉はメタ変数であり，他のメタ記号は付録1の構文に従う．

#### (1) 制御述語

FAIL

つねに失敗を返し，バックトラックを引起す．

( LEVEL 〈 level 〉 )

変数 level に現時点の実行のレベル（実行に成功した手続きの呼出し回数）が代入される．また，手続の本体に$ PARENT_LEVEL という変数を指定すると，呼び出された親のレベルが与えられる．

(／ 〈 level 〉) または／

カットオペレータと呼ばれる述語で，指定したレベルからこの述語の呼出に至るまでに記録した全てのバックトラック点を除去する．／は(／ $PARENT_LEVEL ) と同じ意味である．〈 level 〉は呼出し時にレベルを示す整数が代入されていなければならない．

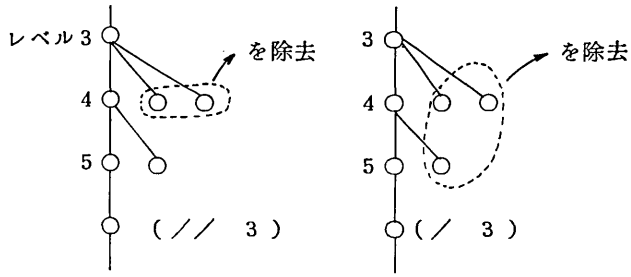
例：

```
(+ (NEQ ¥X ¥Y) - (EQ ¥X ¥Y) - / - FAIL)
(+ (NEQ ¥X ¥Y))
(+ (EQ ¥X ¥X))
```

この例は二つの引数の構造が等しくないことをテストするプログラムで，カットオペレータを用いることによって記述が可能となる例である．EQが成功ならば，／によりバックトラック点が除去され，次のステップのFAILでバックトラックが起っても，別の選択枝は存在しないので，NEQは失敗となる．一方EQに失敗すれば，二つ目の手続きとマッチしNEQは成功となる．

(／／ 〈 level 〉)

〈 level 〉は変数又は数値定数で呼出し時には整数が代入されていなければならない．〈 level〉へのバックトラックを禁止するオペレータで，( LEVEL〈 level 〉) と対で使用するか，(／／ $PARENT_LEVEL ) を用いる．(／ 〈 level 〉) は現在のレベルから指定したレベルの間のすべてのバックトラック点を除去するのに対して，(／／ 〈 level 〉) は指定したレベルのバックトラック点のみを除去する．




例　PはAまたはBの方法で解かれるとする．このときAが一意解でなく，Aのすべての解が知りたい時，／の代りに／／を用いてBの実行を禁止する．

(＋P－A－(／／ $PARENT_LEVEL))

(＋P－B)

(AND 〈 procedure call 〉* )

定義

```
(+ (AND))
(+ (AND ?L . ?LL) - ¥L - (AND . ¥LL))
```

通常のバックトラックをともなった手続き呼出し列の実行と同じである．すなわち，呼出しの並びを左から右へ実行し，すべての手続き呼出しが成功した時にANDは成功である．一旦，ANDが成功した後でバックトラックが起った場合は，一番最近に成功した手続きの別の選択枝が再試行される．少くとも一つの手続き呼出しが失敗した場合ANDは失敗である．

(－$Q_1$ －$Q_2$ －$Q_3$ ……－Qn)

は　－(AND $Q_1$ $Q_2$ $Q_3$ ……Qn)

と同じである．

(OR 〈 procedure call 〉*)

定義

```
(+ (OR))
(+ (OR ?L . ?LL) - ¥L)
(+ (OR ?L . ?LL) - (OR . ¥LL))
```

手続き呼出しの並びを左から右へ実行し，少くとも一つの呼出しに成功すればORは成功である．どの手続き呼出しにも失敗した時，又は，一旦成功した後でバックトラックが起り，別の選択枝が存在しない場合ORは失敗である．

(＋P－$Q_1$)

(＋P－$Q_2$)

(＋P－$Q_3$)

は

(＋P－(OR $Q_1$ $Q_2$ $Q_3$))

と同じである．

(DAND 〈 procedure call 〉* )

定義

```
(+ (DAND))
(+ (DAND ?L . ?LL)
   - (LEVEL ¥N)
   - ¥L
   - (/ ¥N)
   - (DAND . ¥LL))
```

手続き呼出しの並びを左から右へ実行し，すべての手続き呼出しがバックトラックなしで成功した時にDANDは成功である．手続き呼出しの一つが失敗すると，バックトラック点は存在しないのでDANDは失敗である．また，DANDが一旦成功した後，バックトラックが起っても，DANDへバックトラックすることはない．

( DOR 〈 procedure call 〉* )

定義

```
(+ (DOR))
(+ (DOR ?L . ?LL) - ¥L - /)
(+ (DOR ?L . ?LL) - (DOR . ¥LL))
```

手続き呼出しの並びを左から右へ実行し，少くとも一つの呼出しに成功すればDORは成功である．ただし，ORと異って，DORが一旦成功した後，バックトラックが起ってもDORには別の選択枝がないので，DORへバックトラックすることはない．

(＋ P－$Q_1$－／)

(＋ P－$Q_2$－／)

(＋ P－$Q_3$－／)

は

(＋ P－( DOR $Q_1$ $Q_2$ $Q_3$ ) )

は同等である．

( NOT 〈 procedure call 〉 )

定義

```
(+ (NOT ¥X) - ¥X - / - FAIL)
(+ (NOT ¥X))
```

NOTは，〈 procedure call 〉が失敗のとき成功，成功のとき失敗を返す．

( TRUE 〈 procedure call 〉* )

定義

```
(+ (TRUE . ¥L) - (AND . ¥L) - /)
(+ (TRUE . ¥L))
```

手続き呼出し列の実行が成功，失敗にかかわらず常に成功を返す述語である．手続呼出し列が失敗の場合変数への代入はない．

( FOR 〈 control var 〉〈 initial val 〉〈 final val 〉〈 increment 〉 )

定義

```
(+ (FOR ¥M ?I1 ?I2 ?I3) - (EQ ¥M ¥I1))
(+ (FOR ¥M ?I1 ?I2 ?I3)
   - (LT ¥I1 ¥I2)
   - (EVAL (PLUS ¥I1 ¥I3) ¥M1)
   - (FOR ¥M ¥M1 ¥I2 ¥I3))
```

引数列は左からそれぞれ，制御変数，初期値，終値，増分で，実行時に整数または実数が代入されていなければならない．MにＩ１を代入してFORの後につづく手続き呼出し列を実行する．FORにバックトラックしたとき，Ｉ１に増分Ｉ３を加えた値が代入され，繰返される．どのMに対しても手続き呼出し列が成功せず，Ｉ１がＩ２より大きくなった時点でFORは失敗である．

例　$x$が１から１０まで，$y$が１から１０までの整数のとき，$x+y=10$の解をすべて求める

```
(- (FOR ¥X 1 10 1)
   - (FOR ¥Y 1 10 1)
   - (¥X + ¥Y = 10)
   - (QUERY ¥X ¥Y))
```

( IF 〈 condition part 〉 〈 then part 〉 〈 else part 〉 )

( IF 〈 condition part 〉 〈 then part 〉

定義

```
(+ (IF ?COND ?THEN_PART ?ELSE_PART)
   - ¥COND
   - /
   - ¥THEN_PART)
(+ (IF ?COND ?THEN_PART ?ELSE_PART) - / - ¥ELSE_PART)
(+ (IF ?COND ?THEN_PART) - ¥COND - / - ¥THEN_PART)
(+ (IF ?COND ¥THEN_PART))
```

条件部〈 condition part 〉の呼出しが成功ならば，THEN部〈 then part 〉を，失敗ならば，ELSE部〈 else part 〉を呼出す．ELSE部は省略することができる．ＩＦは，〈 then part 〉または〈 else part 〉が成功ならば成功，失敗ならば，失敗である．また，一旦成功した後，バックトラックが起った時，一番最近に成功した呼出しの別の選択枝が実行されるが，選択枝が全く存在しない場合はＩＦは失敗となる．

例　第１引数と第２引数の小さい方を第３引数に与える手続き

```
(+ (MIN ?A ?B ¥C)
   - (IF (¥A <= ¥B) (EQ ¥A ¥C) (EQ ¥B ¥C)))
```

これは

```
(+ (MIN ?A ?B ¥C) - (¥A <= ¥B) - (EQ ¥A ¥C))
(+ (MIN ?A ?B ¥C) - (¥A > ¥B) - (EQ ¥B ¥C))
```

と同等である．

( REPEAT_UNTIL 〈 body part 〉〈 condition part 〉 )

定義

```
(+ (REPEAT_UNTIL ?L ?COND)
   - (LEVEL ¥N1)
   - (REPEAT1)
   - (LEVEL ¥N2)
   - ¥L
   - (/ ¥N2)
   - ¥COND
   - (/ ¥N1))
```

手続き言語型の繰返し述語で，条件部〈 condition part 〉が成功するまで，〈 body part 〉を同一環境で繰返す．REPEAT_UNTILが成功した後，その手続き呼出しに失敗した場合，別の選択枝は残されていないので，REPEAT_UNTILへのバックトラックは起らない．なお，この述語は，〈 body part 〉や〈 condition part 〉が成功しない場合は無限ループになる可能性がある．

例　リストを読みこんで，二つの部分リストに分ける手続きを，入力リストがNILになるまで繰返す．QUERYは$x$，$y$を印刷した後，常に失敗を返す述語であるので，無限ループを避けるためにはTRUE述語を用いて常に成功を返すようにすれば良い．

```
(- (REPEAT_UNTIL
    (AND (EVAL (READ) ¥LIST)
     (TRUE (APPEND ¥X ¥Y ¥LIST) (QUERY ¥X ¥Y)))(EQ ¥LIST NIL)))
```

### ⑵ メタ述語

( VAR 〈 $x$ 〉 )

〈 $x$ 〉が変数ならば成功，変数でなければ失敗

( NONVAR 〈 $x$ 〉 )

〈 $x$ 〉が変数でなければ成功，変数ならば失敗

( LITATOM 〈 $x$ 〉 )

〈 $x$ 〉が定数で文字アトムならば成功，さもなければ失敗

( REAL 〈 $x$ 〉 )

〈 $x$ 〉が定数で数値アトムならば成功，さもなければ失敗

( FIXP 〈 $x$ 〉 )

〈 $x$ 〉が整数アトムならば成功，さもなければ失敗

( FLOATP 〈 $x$ 〉 )

〈 $x$ 〉が実数アトムならば成功，さもなければ失敗

( EQ 〈 $x$ 〉 〈 $y$ 〉 )

〈 $x$ 〉と〈 $y$ 〉の構造が統一化可能ならば成功，さもなければ失敗，どちらか一方が変数の場合は，他方の値が代入される．例えば，( EQ $X 3 ) は成功でかつ$Xに3が代入される．

( NEQ 〈 $x$ 〉 〈 $y$ 〉 )

〈 $x$ 〉と〈 $y$ 〉の構造が統一化可能ならば失敗，さもなければ成功

( VARCONST 〈 S − exp 〉$^{+}$ )

引数列中の変数を任意定数へ変換する．評価値は常に成功

例　一次方程式を解く手続きの一部

```
(+ (¥X = ¥Y)
   - (SINGLEVAR (¥X = ¥Y))
   - (VARCONST (¥X = ¥Y))
   - (NORM (¥X - ¥Y) ¥Z)
   - (CONSTVAR ¥Z)
   - /
   - (SOLVE ¥Z))
```

```
(+ (¥X = ¥Y)
   - (FREEOFVAR (¥X = ¥Y))
   - (NORM ¥X ¥XX)
   - (NORM ¥Y ¥YY)
   - (AEQUAL ¥XX ¥YY))
```

( CONSTVAR 〈 s − exp 〉$^{+}$ )

引数列中の任意定数( VARCONST で変換したもの )を変数へ逆変換する．評価値は常に成功

( FREEOFVAR 〈 s − exp 〉$^{+}$ )

引数列中に変数が含まれていなければ成功．含まれていれば失敗

( FREEOFCONST 〈 s − exp 〉$^{+}$ )

引数列中に任意定数( VARCONST で変換したもの )が含まれていれば成功．さもなければ失敗

( CONSTP 〈 $x$ 〉 )

〈 $x$ 〉が任意定数( VARCONST で変換したもの )ならば成功．されなければ失敗

( PICKVAR 〈 s − exp 〉 〈 variable list 〉 )

第1引数に含まれる変数のリストを第2引数に指定された変数に与える．評価値は常に成功．

( ASSERT 〈 procedure 〉 )

指定された手続きをその手続き集合の一番後に置く．

実行結果を用いて，手続きを手続き集合に追加するのに用いる．

( ASSERTA 〈 procedure 〉 〈 n 〉 )

手続き集合の〈 n 〉番目に置く．n≧1 でなければならない．

例　階乗計算の結果を手続き集合の一番目に追加し高速化する．

```
(- (FACT 5 ¥X)
   - (ASSERTA (+ (FACT 5 ¥X) - /) 1))
```

ASSERTA前の手続き

```
(+ (FACT 0 1) - /)
(+ (FACT ?X ¥Y)
   - (¥X1 := ¥X - 1)
   - (FACT ¥X1 ¥Y1)
   - (¥Y := ¥X * ¥Y1))
```

ASSERTA後の手続き

```
(+ (FACT 5 120) - /)
(+ (FACT 0 1) - /)
(+ (FACT ?X ¥Y)
   - (¥X1 := ¥X - 1)
   - (FACT ¥X1 ¥Y1)
   - (¥Y := ¥X * ¥Y1))
```

( ASSERT_ONBT 〈 procedure 〉 )

ASSERTした後で，バックトラックが生じたときに，ASSERTした手続きを取り消す．

( RETRACT 〈 procedure 〉 )

手続き集合から指定された手続きの頭部とマッチする最初の手続きを取り除く

( RETRACT_ONBT 〈 procedure 〉 )

RETRACTした後，バックトラックが生じた時に，RETRACTを取り消す．

### (3) 算術計算述語

述語名( ＝，＞，＞＝，…など )はインフィックスオペレータとして宣言されている．また，述語

中の〈 $x$ 〉, 〈 $y$ 〉には，呼出し時に算術項又は数値定数が，〈 n 〉, 〈 m 〉には数値定数が与えられていなければならない．

(〈 $x$ 〉 ＝ 〈 $y$ 〉)　( equal )
(〈 $x$ 〉 ／＝ 〈 $y$ 〉)　( not equal )
(〈 $x$ 〉 ＞ 〈 $y$ 〉)　( greater )
(〈 $x$ 〉 ＞＝ 〈 $y$ 〉)　( greater or equal )
(〈 $x$ 〉 ＜ 〈 $y$ 〉)　( less )
(〈 $x$ 〉 ＜＝ 〈 $y$ 〉)　( less or equal )

〈 $x$ 〉と〈 $y$ 〉を実行し，結果を比較し，条件を満せば成功である．〈 $x$ 〉と〈 $y$ 〉が変数を含む場場合や，実行結果が数値でない場合，条件を満さなかった場合は失敗が返される．

例　－( $A＋3 ＞＝ 2 * $B )
　　－( $X＋$Y＝0 )

(ADD 〈 n 〉 〈 m 〉 〈 var 〉)
(SUB 〈 n 〉 〈 m 〉 〈 var 〉)
(MULT 〈 n 〉 〈 m 〉 〈 var 〉)
(DIVIDE 〈 n 〉 〈 m 〉 〈 var 〉)

〈 n 〉, 〈 m 〉は整数または実数，〈 n 〉と〈 m 〉の演算結果が変数〈 var 〉に代入される．
〈 n 〉, 〈 m 〉が数値定数でない場合は失敗が返される．

### (4) 基本算術演算項

〈 n 〉, 〈 m 〉には，呼出し時に数値定数（整数又は実数）が与えられていなければならない．
演算子はインフィックス宣言されている．

(〈 n 〉＋ 〈 m 〉)
(〈 n 〉－ 〈 m 〉)
(〈 n 〉＊ 〈 m 〉)
(〈 n 〉／ 〈 m 〉)
(〈 n 〉＊＊〈 m 〉)　　mは整数

項の評価値を変数に代入する必要がある場合は，3・5 節で述べる‘：＝’述語を用いればよい．

### (5) そ の 他

(ANS 〈 s－exp 〉$^{+}$)

引数列の解を印刷し，常に成功を返す．

(QUERY 〈 s－exp 〉$^{+}$)

引数列の解を印刷し，常に失敗を返す．

(WITH 〈 package 〉 〈 procedure eall 〉$^{+}$)

指定したパッケージを用いて，手続き呼出し列を実行する．指定した手続きがパッケージに存在しないかまたはglobal 宣言されていなければ，SYSパッケージの手続きと見なす．

### 3.5 Lispとの結合

(1) Lisp関数の評価

手続きの本体でLisp関数を評価し，評価値を変数に代入することができる．このために，EVALと：＝の2種類の述語が利用できる．

（EVAL 〈 form 〉 〈 var 〉）または（EVAL 〈 form 〉）

〈 form 〉を評価し，評価値を変数へ代入する．但し，引数は評価されない．

例 －（EVAL （CAR （CONS （A B） C）） $V）

$VにはCONSが代入される（（A B）ではない）

（〈 var 〉 ：＝ 〈 form 〉）

〈 form 〉を評価し，評価値を変数〈 var 〉に代入する．その際，引数も評価される．

例1 －（$V ：＝ （CAR（CONS （A B） C））

$Vには（A B）が代入される．

例2 Lisp関数による<=の定義（OR，LESSP，EQUALはLisp関数でTまたはNILを返す）

```
(+($X <= $Y)
   -(REAL $X $Y)
   -($V := (OR(LESSP $X $Y)(EQUAL $X $Y)))
   -(EQ $V T))
```

(2) Lisp関数による手続きの定義

ADLでは，手続きをLisp関数で定義することができる．Lisp関数の呼出しは，ホーン節で書かれた手続きの呼出しと同じである．インタプリタとのインタフェースとして，Lisp関数は，成功，失敗のいずれかの評価値を返さねばならない．成功の場合は代入リスト（NIL又は（（変数 値）（変数 値）…））を，失敗の場合はアトムFAILを返す．FAILはバックトラックを引起す．

インタプリタからの手続き呼出しの方法は関数の属性によって異っている．すなわち，属性がEXPR，SUBRの場合は，環境のもとに評価された引数列が与えられるので，Lisp側では特別の処理を必要としない，FEXPR，FSUBRの場合は，実引数列とその環境が渡されるので，Lisp側で引数列を環境のもとに評価する必要がある．（システムに用意されている*COPY関数を用いる．）

Lisp関数もパッケージごとに管理されるが，実行時にはglobal な手続きと見なされるので注意を要する．以下にEXPRとFEXPR属性の定義例を示す．図中，／$は編集時のカーソル記号，*VARは引数か変数かをテストするLisp関数である．

EXPR属性の定義例：第1引数，第2引数の和を第3引数（変数）に代入する．

```
I EDIT(ADD EXAM)
   OK-ET
R PA
((EXPR 0 0) /$
            LAMBDA
            (N1 N2 N3)
            (COND
             ((AND (NUMBERP N1) (NUMBERP N2) (*VAR N3))
              (LIST (CONS N3 (PLUS N1 N2))))
             (T (QUOTE FAIL))))
R OK
  = ADD
```

FEXPR属性の定義例：引数列が実数かテストする．

```
I EDIT(REAL EXAM)
   OK-ET
R PA
((FEXPR 0 0) /$
             LAMBDA
             (ARGS ENV)
             (PROG (N)
                   (SETQ ARGS (*COPY ARGS ENV))
              LOOP (COND ((NULL ARGS) (RETURN NIL)))
                   (SETQ N (CAR ARGS))
                   (SETQ ARGS (CDR ARGS))
                   (COND
                    ((OR (*VAR N) (NOT (NUMBERP N)))
                     (RETURN (QUOTE FAIL))))
                   (GO LOOP)))
R OK
  = REAL
```

### 3. 6 FORTRANとの結合

FORTRANのロードモジュールとの結合を実現するために，対象に配列を拡張し，配列への値の代入，参照，サブルーチン，関数呼出しの述語が用意されている．

(ARRAY 〈 array 〉 〈 size 〉 〈 type 〉)

配列の宣言．〈 array 〉は配列名で変数でなければならない．〈 size 〉は配列の大きさを一次元で指定する．〈 type 〉はデータの型で，FIX，FLOAT，または（String 文字数）のいずれかでなければならない．

(ASET 〈 array 〉 〈 index 〉 〈 value 〉)

〈 array 〉は配列名で変数，一次元配列の〈 index 〉で指定された要素に〈 value 〉を代入する．

(AREF 〈 array 〉 〈 index 〉 〈 var 〉)

一次元配列の〈 index 〉で指定された要素の値を変数〈 var 〉に代入する．

(ASET 2 〈 array 〉〈 m 〉〈 n 〉〈 index 1 〉 〈 index 2 〉 〈 value 〉)

〈 array 〉を（〈 m 〉〈 n 〉）で宣言された二次元配列と見なし，二次元配列の（〈 index 1 〉，〈 index 2 〉）要素に値を代入する．

(AREF 2〈 array 〉〈 m 〉〈 n 〉〈 index 1 〉 〈 index 2 〉 〈 var 〉)

〈 array 〉を(〈 m 〉〈 n 〉)で宣言された2次元配列と見なし，二次元配列の(〈 index 1 〉，〈 index 2 〉)要素の値を変数〈 var 〉に代入する．

( ARRAYP 〈 var 〉)

〈 var 〉が配列名ならば成功．さもなければ失敗を返す．

( CALL (〈 subroutine name 〉 〈 argument 〉*))

引数列は，呼出し時には，配列又は数値定数でなければならない．実引数に変数( FORTRANの)を与える必要がある場合も1要素の配列として宣言しておかねばならない．

( INT_FUNC (〈 function name 〉 〈 argument 〉*) 〈 var 〉)

FORTRANの整数関数を呼出し，関数値を変数〈 var 〉に代入する．他はCALLと同じ．

( REAL_FUNC (〈 function name 〉 〈 argument 〉*) 〈 var 〉)

FORTRANの実数関数を呼出し，関数値を変数〈 var 〉に代入する．他はCALLと同じ．

例 連立一次方程式を解く手続きの一部

```
(+ (RENRITU ¥N (¥EQU . ¥EQUS))
   - (¥SIZE := ¥N * (¥N + 1))
   - (ARRAY ¥A ¥SIZE FLOAT)
   - (COEF_SET ¥A ¥N (¥EQU . ¥EQUS))
   - (ARRAY ¥ILL 1 FIX)
   - (¥K := ¥N + 1)
   - (CALL (GAUELS ¥A ¥N ¥N ¥K 4.999999E-06  ¥ILL))
   - (AREF ¥ILL 1 ¥COND)
   - (RETURNCHECK ¥COND)
   - (ANS_SET ¥A ¥N ¥K (¥EQU . ¥EQUS)))
```

## 4. デバッガ

デバッグモードによるプログラムの実行を行う．手続きのトレースや解探索の制御に用いる．以下，機能別に説明する．なおコマンド形式中の小文字は省略可能である．

### (1) デバッグが呼出し

DEBUG ( )

システムコマンド入力状態でDEBUGコマンドを入力するとデバッガモードに入る．引きつづき以下にあげるデバッガコマンドを入力する．下線のある部分が端末からの入力である．

```
I  DEBUG ( )
     DEBUG
R  TR (〈手続名〉〈パッケージ〉)  ┐
R  RUN(〈手続名〉〈パッケージ〉)  ├ デバッガコマンドの入力
    ⋮                          ┘
```

### (2) デバッガ呼出し点の設定

PAUse －Env

現実行環境に呼出し点を設定する．同一環境とは，top goal や導出された手続きの本体の部分である．現goalが(環境2 $-\ell_1-\ell_2-\ell_3-\ell_4$)(環境1 $-\ell_5-\ell_6$)とすると，$\ell_2$，$\ell_3$，

$\ell_4$ の実行の直前にデバッガに入る．

PAUse（〈手続名〉〈パッケージ〉）

指定した手続きの評価直前にデバッガに入る．

PAUse Open※

オープンリストから保留中の goal を取り出す直前にデバッガに入る．

Continue - Env

Continue （〈手続名〉〈パッケージ〉）

Continue Open※

上記の Pause で指定した呼出し点を解除する．

**(3) インタプリタへの復帰**

Step

step by step の実行を行う．すなわち毎導出後にデバッガに入る．

Go

Panse 条件を満足するまで連続的に実行する．

Execute

現 goal の最初の手続呼出しの評価が終るとデバッガに入る．

**(4) トレースメッセージ制御**

TRace-All

すべての手続き呼出しに対し，次のようなトレースメッセージを表示する．

(a)最初の評価

（（〈レベル番号〉）CALL〈評価前の手続呼出し〉）

(b)評価に成功した場合

（（〈レベル番号〉）EXIT〈評価後の手続呼出し〉）

(c)評価に失敗した場合

（（〈レベル番号〉）FAIL〈評価前の手続呼出し〉）

(d)再評価の場合

（（〈レベル番号〉）REDO〈評価後の手続呼出し〉）

(e)実際に試行する手続の表示

（〈手続名〉〈手続番号〉〈他の選択枝の個数〉）

Lisp でかかれた手続の場合は（EVAL）と表示される．

TRace （〈手続名〉〈パッケージ〉）

指定した手続名の実行に対して，上記のトレースメッセージを表示する．

UNTRace-All

UNTRace （〈手続名〉〈パッケージ〉）

前述の trace 指定を解除する．

### (5) エディタ呼出し

EDit Goal

現goal を参照する.

EDit Procedure

現手続き呼出しと導出を予定している手続き集合を一時的に編集する. 解の探索を制御するのに用いる.

EDit Open ※

オープンリストを編集する. 解探索の制御に用いる.

EDit (手続名 パッケージ)

エディタの内部呼出し(5章エディタ参照)

EDit Stack

現ゴールに至るまでのバックトラック点を参照する.

### (6) 探索の制御

TRUE

現手続呼出しを評価することなく成功させる.

FAIL

現手続呼出しを評価することなく失敗させる.

/ 〈n〉

レベルnまでのバックトラック点を削除する.

// 〈n〉

レベルnのバックトラック点を削除する.

BACK 〈n〉

現ゴールの評価を取消しレベルnのゴールを復元する.

OPEN ※

残りのゴールをオープンリストへ変換する.

### (7) そ の 他

Help

デバッガのコマンド一覧の表示

Print

現手続呼出しを表示する.

EVAL 〈S式〉

LispのS式を評価する.

RUN(〈手続名〉 〈パッケージ〉)

インタプリタの内部呼出し. デバッグモードによる手続きの評価を行う.

実行が終了すると次のような実行状況が表示される.

(S=×× SU=×× FU=×× EV=×× B=××)

S：実行に成功した手続きの呼出回数　　SU：統一化に成功した回数

FU：統一化に失敗した回数　　EV：Lisp関数の評価回数　　B：バックトラックの回数

QまたはOKまたはEND

デバックモードを終了し，システムコマンドモードに移る．

## 5. エディタ

### (1) エデイタ呼出し

EDit　(〈手続名〉〈パッケージ〉)

パッケージに属する手続き(手続名アトムのパッケージ属性)を編集する．新しく定義する場合は手続きの型を入力する必要がある．すなわち，"NEW GENERATION WHAT TYPE？"が表示されるので，HORN，FRAME※，EXPR，FEXPR，LAPのいずれかを入力する．HORNの場合は，他のパッケージからの参照の可否(LOCAL，GLOBAL)と実行の優先順位とコストをリストにしたものを定義のcar部におく．省略値は(LOCAL 0 0)である．※ Lisp関数は，パッケージごとに管理されるが，実行時はglobalな手続きと見なされる．

EDit　(〈パッケージ〉 INFIX)

S式をインフイックス形式で記述するために，インフイックスオペレータをパッケージごとに定義しておく．インフイックス形式で記述されたS式は，順位文法を用いてプレフイックス形式(内部表現)に変換される．オペレータはリスト(〈文字アトム〉〈優先順位〉{LR/RL})で定義される．優先順位は整数で大きいほど優先度が高い．LR，RLは同順位のときの評価の順序を示す．LRは左から右，RLは右から左である．

例　インフイックスオペレータの定義と変換

((& 0 RL)(** 3 LR)(* 2 LR)(/ 2 LR)(+ 1 LR)
(- 1 LR))

(A*B*C) ⟷ (*(* A B)C)

($x$&$y$&z) ⟷ (& $x$(& $y$ $z$))

(A*$x$**2+B*$x$+C)⟷(+(+(* A (** $x$ 2))(* B $x$)) C)

EDit　(〈パッケージ〉 CATA)

手続きの編集時に自動的にカタログ化されているカタログリスト(手続き名リスト)を編集する．

EDit (ADL PACKAGE)

システムに登録されているADLのパッケージ名を編集する．

### (2) 編集用コマンド

(2)－1　ポインタ移動コマンド

| | |
|---|---|
| F | ポインタをforwordに一つ進める． |
| FF 〈n〉 | ポインタをforwordにn個進める． |
| B | ポインタをbackwordに一つ戻す． |
| BB 〈n〉 | ポインタをbackwordにn個戻す． |

| | |
|---|---|
| DR | Down Right．ポインタを次のリストの左はしに移す． |
| DL | Down Left．ポインターを一つ前のリストの右はしに移す． |
| UR | Up Right．ポインタをリストの右外側に移す． |
| UL | Up Left．ポインタをリストの左外側に移す． |
| S 〈S式〉 | 〈S式〉と同じパターンを持つ式を forword にサーチしポインタを移す． |
| SA 〈S式〉 | Search All．サーチを forwordに繰返す．サーチしたS式を表示した後，入力状態となるので▼Q▼（quit）又は▼G▼（go）を入力する． |
| RS 〈S式〉 | Reverse Search．サーチを backword に行う． |
| RSA 〈S式〉 | Reverse Search All．サーチを backword に行う． |

(2)－2 書込みコマンド

| | |
|---|---|
| I 〈S式〉 | Insert．現在のポインタの位置にS式を挿入する． |
| W 〈S式〉…； | Write．；までのS式を書く． |
| R 〈S式1〉〈S式2〉 | Replace．次のS式の中のS式1をS式2で置き代える． |
| RA 〈S式1〉〈S式2〉 | Replace All．すべてのS式の中のS式1をS式2で置き代える． |
| D | Delete．次のS式を消去する． |
| K | Kill．次のS式を kill して kill スタックに保存． |
| Y | Yank．kill スタックのトップを挿入する． |
| YC | Yank Copy．Yank しかつkill スタックは元の状態のまゝおく． |
| BI 〈n〉 | Bracket In．ポインタの次からn個のS式をカッコに入れる． |
| BO | Bracket Out．次のS式の各要素をカッコの外に出す． |

(2)－3 表示コマンド

| | |
|---|---|
| DEP〈d〉 | DEPth．表示するリストの深さの上限を指定．default は10 |
| BEF〈b〉 | BEFore．ポインタより前のS式の数の上限を指定．default は1 |
| AFT〈a〉 | AFTer．ポインタより後のS式の数の上限を指定．default は3 |
| P | ポインタの前 b．後 a，深さ d をpretty print する． |
| PP 〈n〉〈m〉 | ポインタの前 n個，後 m個のS式を pretty print する． |
| PF 〈m〉 | ポインタの後 m個のS式のみを pretty print する． |
| PA | すべてのS式を pretty print |
| PPP | すべてのS式を構造の段落をつけずに表示する． |

(2)－4 その他

| | |
|---|---|
| CN 〈name〉 | Change Name．手続名を変更する． |
| CP 〈name〉 | Change Package．パッケージ名を変更する． |
| NAME | 現在編集中の手続名を表示する． |
| PACKAGE | 現在編集中のパッケージ名を表示する． |
| UNDO | 入力したコマンド列を取り消す． |
| Help | Help．エデイタのコマンド一覧を表示する． |

| | | |
|---|---|---|
| SAVE | (〈手続名〉〈パッケージ〉) | 現在編集中のものを指定したパッケージの指定した手続名として保存する．手続名アトムのパッケージ属性に定義される． |
| RESAVE | 〃 | すでに保存されている定義を更新する． |
| OK | | 編集中のものをSAVEした後，エディタを終了する． |
| Q | | SAVEをせずにエディタを終了する． |

## 6. パッケージ管理

本システムでは，プログラムはパッケージ化して管理される．パッケージは，カタログリスト，インフィックスオペレータの定義(無いこともある)，手続きの定義より成る．以下にパッケージの管理のためのシステムコマンドを掲げる．

LOAD (〈データセット名〉$^{+}$)

データセットに保存されているソース形式の手続きをLispシステムのセル領域にロードする．〈データセット名〉は，外部のデータセット名に対応するアトムである．データセット名がLispの区切り記号を含む場合は$$アトムとする．

例 LOAD($$$USER.TEXT$)

LOAD($$$▼LIB.ADL.EXAM(DCG)▼$)

SAVE (〈データセット名〉〈パッケージ名〉 {NEW/[OLD]})

パッケージのカタログ，インフィックオペレータの定義，手続き(ソース形式)を指定したデータセットへ保存する．新規に作成する場合はNEWを，すでにデータセットが存在する場合はOLD(省略可)を指定する．

CLEARPACKAGE(〈パッケージ名〉)

指定したパッケージをADLの作業領域から消去する．

CLEARPROC(〈手続名〉〈パッケージ名〉)

指定したパッケージの中の指定した手続きを消去する．

*COMPILE(〈パッケージ名〉)

パッケージ内に定義されているすべてのLisp関数をコンパイルする．

*LAP(〈パッケージ名〉)

パッケージ内のすべてのLAP関数をアセンブルする．

## 7. おわりに

本稿では，Prolog系言語ADLの基本部分の仕様を述べた．Prologとして使用する場合は，今回公開した部分だけ使用していただけば良い．ADLは設計支援を主な対象領域として発展中であり，拘束条件解法，対象指向プログラミングを実装している．この部分は仕様が安定した段階で公開する予定である．

システムの障害や機能に関して問題点があれば筆者へ御連絡いただきたい．

［謝辞］ 本システムの一部として，本学大型計算機センター研究開発部，篠原 武氏ら作成のLisp構造エデイタを使用させていただいた．またDCGの例題は本学電子工学科・吉村賢治氏が提供されたものである．両氏に感謝する．

**参考文献**

1. Kowalski, R., Predicate Logic as Programming Language, Proc. of IFIP Congress 74, 1974.
2. W. F. Clocksin,C. S. Mellish, Programming in Prolog, Springer-Verlag, 1981. 中村克彦訳 Prologプログラミング，日本コンピュータ協会，1982.
3. 中島秀之 Prologとその処理系，情報処理，23，11，1982.
4. 中島秀之 Prolog入門，bit，14，5～7，1982.
5. 横井俊夫，淵一博 推論機構を内蔵した述語論理型言語Prolog，日経エレクトロニクス，9，27，1982.
6. Pereira, L., Pereira, F. and Warren, D., User's Guide to DEC System-10 PROLOG, Dept. AI Research, Edinburgh Univ., 1978.
7. Warren, D. H. D, WARPLAN:A System for Generating Plans, Memo 76, Dept. of Computational Logic, University of Edinburgh, 1974.
8. Fernando C. N. Pereira and David H. D. Warren, Definite Clause Grammers for Language Analysis-A Survay of the Formalism and a Comparison with Augumented Transition Networks, Artiftcial Intelligence, 13, 3, 1980.
9. 計算機マニュアル FACOM OS IV LISP手引書(64SP－3190)，富士通㈱
10. 篠原武，有川節夫 LISP関数の編集及び保存復元について，九州大学大型計算機センター広報，14，2，1981.

## 付録1. ＡＤＬの構文

ADLの構文をBNF記法で表わす．図中，右肩に付された‘＋’は1個以上の，‘＊’は0個以上の並びを表わす．

〈 goal 〉：：＝(〈 procedure calls 〉)

〈 procedures 〉：：＝(〈 procedure 〉$^{+}$)

〈 procedure 〉：：＝(〈 procedure head 〉) |
(〈 procedure head 〉〈 procedure body 〉)

〈 procedure head 〉：：＝＋(〈 procedure name 〉〈 S－exp 〉＊)※1 |
＋(〈 term 〉〈 procedure name 〉〈 term 〉)※2

〈 procedure body 〉：：＝〈 procedure calls 〉

〈 procedure calls 〉：：＝－〈 procedure call 〉 |
－〈 procedure call 〉〈 procedure calls 〉

〈 procedure call 〉：：＝(〈 procedure name 〉〈 S－exp 〉＊)※1 |
(〈 term 〉〈 procedure name 〉〈 term 〉)※2 |
〈 variable 〉※3 | 〈 special primitive 〉※4

〈 term 〉：：＝〈 S－exp 〉 | 〈 term 〉〈 infix operator 〉〈 term 〉※5

〈 S－exp 〉：：＝〈 constant 〉 | 〈 variable 〉 | (〈 S－exp 〉＊) |
(〈 S－exp 〉$^{+}$.〈 S－exp 〉)※6 |
(〈 term 〉)

〈 variable 〉：：＝$〈 Lisp literal atom 〉 | ？〈 Lisp literal atom 〉 |
％〈 Lisp literal atom 〉〈 Lisp literal atom 〉？ | _※7

〈 constant 〉：：＝〈変数以外のLisp atom 〉

〈 psocedure name 〉：：＝〈 Lisp literal atom 〉

---

※1 引数の数は手続名によって決る．

※2 手続名がインフィックス・オペレータの宣言がしてあるとき．

※3 実行時には(手続名 引数…)が代入されていなければならない．

※4 初版では，／(カット記号)，FAIL，その他システムが内部的に使用するものがある．

※5 インフィックス・オペレータを含むＳ式は順位文法に従って解釈される．(本文3.2参照)

※6 点記法(dot notation)(本文3.1参照)

※7 (本文3.1参照)

## 付録2. プログラム例

例題を参照する場合は，

LOAD(＄＄＄LIB・ADL・EXAM(〈パッケージ名〉)＄)を入力し，プログラムをロードする．

例1. 階乗の計算

```
I EDIT(FACT EXAM)
    OK-ET
R PA
    ((LOCAL O O)
    /$
    (+ (FACT O 1) - /)
    (+ (FACT ?X ¥Y)
       - (¥X1 := ¥X - 1)
       - (FACT ¥X1 ¥Y1)
       - (¥Y := ¥X * ¥Y1))
    )
R OK
  = FACT
  TIME      25MS.
I EDIT(G1 EXAM)
    OK-ET
    ((LOCAL O O)
    /$
    (- (FACT 5 ¥Y) - (ANS ¥Y))
    )
R OK
  = G1
  TIME      15MS.
I RUN(G1 EXAM)
      ADL (VERSION A-2) START
    (120)
    (S   =  36 SU =  26 FU =  5 EV =  12 B  =  0)
  = NIL
  TIME       7MS.
```

例2. insertion sort

```
I EDIT(INSORT EXAM)
    OK-ET
R PA
    ((LOCAL O O)
    /$
    (+ (INSORT NIL NIL))
    (+ (INSORT (¥X . ¥LO) ¥L)
       - (INSORT ¥LO ¥L1)
       - (INSERT ¥X ¥L1 ¥L))
    )
R OK
  =INSORT
  TIME      28MS.
I EDIT(INSERT EXAM)
    OK-ET
R PA
    ((LOCAL O O)
    /$
    (+ (INSERT ¥X (¥Y . ¥LO) (¥Y . ¥L))
       - (ORDER ¥Y ¥X)
       - /
       - (INSERT ¥X ¥LO ¥L))
    (+ (INSERT ¥X ¥LO (¥X . ¥LO)))
    )
R OK
  = INSERT
  TIME      34MS.
```

```
I EDIT(G2 EXAM)
    OK-ET
R PA
    ((LOCAL O O)
    /$
    (- (INSORT (D 5 G A 1 9 B K S X P) ¥L) - (ANS ¥L))
    )
R OK
  = G2
  TIME      21MS.
I RUN(G2 EXAM)
      ADL (VERSION A-2) START
    ((A B D G K P S X 1 5 9))
    (S  =  95 SU =  55 FU =  14 EV =  57 B  =  8)
  = NIL
  TIME      15MS.
```

例3. n－QUEENS

n個のクィーンをチェス盤面上に互いに同じ行，列になく，互いに右対角線上，左対角線上にないように置く問題で可能な解をすべて求める．

プログラムはパッケージEXAMを参照．

```
I EDIT(QUEENS_FAST EXAM)
    OK-ET
R PA
    ((LOCAL O O)
    /$
    (- (EVAL (READ) ¥N)
       - (MKSEQ ¥N ¥SEQ)
       - (QUEENS2 ¥SEQ NIL ¥Q)
       - (QUERY ¥Q))
    )
R OK
  = QUEENS_FAST
  TIME      26MS.
I RUN(QUEENS_FAST EXAM)
      ADL (VERSION A-2) START
R 5
    ((4 2 5 3 1))
    ((3 5 2 4 1))
    ((5 3 1 4 2))
    ((4 1 3 5 2))
    ((5 2 4 1 3))
    ((1 4 2 5 3))
    ((2 5 3 1 4))
    ((1 3 5 2 4))
    ((3 1 4 2 5))
    ((2 4 1 3 5))
    (S  =  162 SU =  1954 FU =  686 EV =  550 B  =  454)
  = FAIL
  TIME      390MS.
```

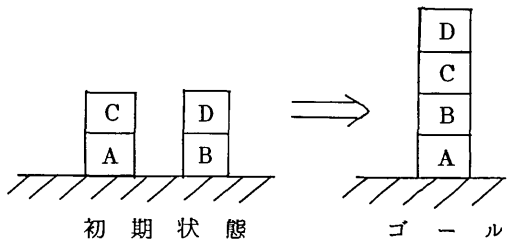
例4. 英文の解析と合成

プログラムはパッケージDCGを参照

```
I EDIT(PARSE DCG)
    OK-ET
R PA
    ((LOCAL 0 0)
    /$
    (- (READS ¥X)
       - (SENTENCE ¥S ¥X NIL)
       - (ANS ¥S)
       - (SENTENCE ¥S ¥Y NIL)
       - (ANS ¥Y))
    )
R OK
  = PARSE
  TIME       28MS.
I RUN(PARSE DCG)
      ADL (VERSION A-2) START
R (MARY WAS LIKED BY JOHN)
    ((S DCL (VOICE PASSIVE) (NP (NPR JOHN)) (TNS PAST) (VP (V LIKE)
    (NP (NPR MARY)))))
    ((MARY (WAS (LIKED (BY (JOHN NIL))))))
    (S  =  42 SU =  289 FU =  519 EV =  3 B  =  122)
  = NIL
  TIME       80MS.
```

例5. WARPLAN

プログラムはパッケージWARPLANを参照




```
I EDIT(G5 WARPLAN)
    OK-ET
R PA
    ((LOCAL 0 0)
    /$
    (- (PLANS
        ((ON A FLOOR) & (ON B FLOOR) & (ON C A) & (ON D B) &
         (CLEAR C)& (CLEAR D))
        ((ON B A) & (ON C B) & (ON D C) & (ON A FLOOR) & (CLEAR D))))
    )
R OK
  = G5
  TIME       67MS.
I RUN(G5 WARPLAN)
      ADL (VERSION A-2) START
    ((((ON A FLOOR) & (ON B FLOOR) & (ON C A) & (ON D B) & (CLEAR C) &
    (CLEAR D)) . (MOVE D B FLOOR) . (MOVE C A FLOOR) . (MOVE B FLOOR A)
     (MOVE C FLOOR B) . (MOVE D FLOOR C)))
    (S  =  318 SU =  2746 FU =  3944 EV =  374 B  =  1683)
  = NIL
  TIME      627MS.
```