EPLAS言語仕様　第1.0版

埼玉大学 先端情報システム工学研究室
eplas@aise.ics.saitama-u.ac.jp

最終更新: 2010年8月

変更履歴

版	日付	編集者	コメント
1.0	2010/08/15	Wencheng FANG <wencheng@aise>	初版

Contents

1 はしがき

1.1 概要

EPLAS は，強い動的型付け言語です.

```
s = "1"; // OK, s is a string.
s = s + 1; // NG
s = s + "1"; // OK "11"
```

1.2 プログラム例

> 1 から 6 の番号が振られた 6 つの扉があり, それぞれ天国または地獄へ通じている. そ れぞれの扉の前には番人がおり, そのうち 3 人だけ嘘つきである. 嘘つきの番人のいる 扉は, 地獄へとつながっている. 各々の番人は,「私の扉の隣の扉は天国へ通じる扉ではありません.」と証言した. 扉 1 の番人は,「4 の扉は天国へ通じていますか」という質問に,「いいえ」と答えた. 扉 2 の 番人は,「6 の扉は天国へ通じていますか」という質問に,「いいえ」と答えた. 天国への扉はどの扉か.

以上の「天国への道」という論理パズルを解くプログラムは、以下に示します.

```
read();
assumptions = read();
// pick any 3 of 6 doormen
assumptions = { x | #x == 3 : x in \assumptions };
savepoint;
foreach x in assumptions do
    restore;
    me + x;
    while me++ do
        // get all possible conclusions
    end;
    if %me then
        // if inconsistent, try next combination
        continue;
    else
        print(me[]);
        break;
    end;
end;
```

Figure 1: 背理法で論理パズルを解くプログラム

2　文法

2.1　字句構造

EPLAS の構文は，文脈自由文法によって定義されます.

終端記号は，字句文法及び構文文法の生成規則中では，固定幅 (fixed width) フォントで表示する。

非終端記号の定義は，その終端記号の名前及びその直後のコロンによって始まる。定義の中で非終端記号は イタリック (italic) で表示する。非終端記号の次の行以降には，非終端記号を代替する 一つ又は複数の右辺が現れる。

例：

IfThenStatement:
if *Expression* then
Statement

2.1.1　識別子

英文字から始まり，英文字または数字からなります. 識別子の長さに制限はありません.

2.1.2　コメント

二つのスラッシュ('//') から行末までをコメントと見なします.

また、'/*' から'*/' までをコメントと見なします.

2.1.3　予約語

true, false, if, then, else, end, while, break, continue, foreach, in, do, procedure, function, with, return, me, savepoint, restore

2.2　リテラル

数字の 1 や文字列"hello world"のようにプログラムの中に直接に記述できる値の事をリテラルといいます。

2.2.1 数値

十進法で表した整数と小数が数値として評価されます．例：

123
-123
123.45

2.2.2 文字列

文字列はダブルクォートに囲まれています．例：

”hello world!”

2.2.3 真理値

真理値は “**true**” もしくは “**false**” のいずれかで表され，それぞれ真もしくは偽を表すデータです．

2.2.4 論理式

論理式は，アンダースコア (’_’) に囲まれています．例：

_ @x (P(x) -> Q(x)) _

論理式の文法は，subsection 3.3を参照してください．

2.2.5 集合

集合は，大括弧’{’ と’}’ に囲まれている式で表されます．

SetStatement:
{ *expressionList* }

例：

{ 1, 2, 3, 4 }
{ var1, var2, var3, var4, var5, var6 }

2.2.6 リスト

リストは，中括弧’<’ と’>’ に囲まれている式で表されます．順序のない集合に対して，リストでは要素のリストに入れ込まれたときの順序が保たれます．

ListStatement:
< *expressionList* >

2.3 演算子

演算子は，以下のトークンとします。

`*`, `/`, `+`, `-`

`#`, `$`, `\\`, `**`

`&&`, `||`, `!`

`==`, `!=`, `<`, `<=`, `>`, `>=`

`$$`, `%%`

3 変数及び型

3.1 変数及び代入文

AssignmentStatement:
varaible '=' *expression*

変数の型を明示的に宣言する必要はありませんが、一番最初に変数に代入された値の型をその変数の型と決定し、それ以降の式ではその型を使って型検査を行ないます。

アルファベット大文字で始まる識別子は，定数として扱われます．定数は，一度だけ値を代入することができます．

例：

```
1 CONST1 = "a␣const␣string";
2 CONST1 = "re-define␣const␣string"; // NG
3 ConstValue = "another␣const␣string"; // OK
```

3.2 基本型

基本型は、数値型、文字列型と真理値型とします。

3.2.1 数値型

EPLAS で扱える数値は，整数と浮動小数点数の 2 種類です．数値型は，整数または浮動小数点の値域を持つ型とします．整数の範囲は，64bit の long になります．浮動小数点の範囲は，64bit の double になります．整数的な値に作用する以下の演算子を提供します．

数値演算子

```
*, /, +, -
```

比較演算子

```
<, <=, >, >=
```

等価演算子

```
==, !=
```

3.2.2 文字列型

文字列型は，文字列の値を持つ型とします．

演算子

接続 (+)

```
"abc" + "de" => "abcde"
```

二つの文字列をつなぐ二項演算子．

文字数 (#)

```
#"ab" => 2
```

文字列の文字数を整数で返す単項演算子．

変換 ($)

```
$123 => "123"
```

文字列以外の型の値を文字列に変換する単項演算子．

文字 ([n])

```
"123"[1] => "1"
"123"[] => "3"
"123"[-1] => "2"
```

n 番目の文字を取り出す演算子．n が指定されない場合、最後の文字を返す．

文字 ([n..m])

```
"1234"[2..3] => "23"
```

n 番目から m 番目の文字を取り出す演算子．

3.2.3 真理値型

真理値型は，真理値を持つ型とします．

等価演算子

等価 (==)

`a == b`

a と b が同じ真理値．

不等価 (!=)

`a != b`

a と b が異なる真理値．

3.3 論理式型

Table 1: 論理式を形成する語彙

語彙	定義
項	EPLAS の識別子
関数	EPLAS の識別子
結合子	英文字と数字を除く符号
量化子	全称記号@及び存在記号#
補助記号	括弧 () 及びカンマ，

形成規則 1. 論理式

(1) 項は論理式である．

(2) P が n 項の関数，$t_1, \cdots, t_n$ が項であるならば，$P(t_1, \cdots, t_n)$ は論理式である．

(3) A と B が論理式，op が結合子ならば，`(A op B)` は論理式である．

(4) A を論理式，x を項とすると，`(@x(A))`, `(#x(A))` はともに論理式である．

(5) **(1)**，**(2)**，**(3)**，**(4)**，**(5)** によって論理式とされるもののみが論理式である．

プログラム中で利用する論理体系をカスタマイズできます．デフォルトでは，命題論理と一階述語論理と二階述語論理を対応し，結合子の否定 (!)，論理積 (&&)，含意 (->) が定義されています．例：

```
_ A -> B _
_ predicateA(ConstTerm1) _
_ @P(P(func(a))) -> #z(Q(z)) _
```

単項演算子

否定 (~)

```
~b
```

論理式 b の否定を取ります.

述語化 (*)

```
*func(b)
```

論理式 b に述語 func をかけます.

全称量化 (@)

```
@x(b,2)
```

論理式 b の二つ目の定項を束縛変数 x で全称量化子によって束縛します．二つ目の引数が省略された場合，最初の定項を量化します.

存在量化 (#)

```
#x(b,2)
```

論理式 b の二つ目の定項を束縛変数 x で存在量化子によって束縛します．二つ目の引数が省略された場合，最初の定項を量化します.

定項代入 ([t/x])

```
b["a"/x]
```

論理式 b の束縛変数 x に定項”a”を代入します.

文字列化 ($)

```
$b
```

論理式 b を文字列型として返します.

比較演算子

等価 (==)

```
a == b
```

式の符号と順序がまったく同じである.

不等価 (!=)

```
a != b
```

式は同じでない.

正規表現 (~=)

```
a ~= b
```

a が正規表現式 b と一致するかどうか.

矛盾 ($$)

```
a $$ b
```

a が b の導出元になっている．`a == not(b)` と同値.
注意 1 ：否定演算が定義されている論理体系のみ有効。

3.4 集合型

同一の型を持つ一つ以上の値または変数から成る集合を表す型です。

例：

```
s = {"ab", "bcd", "e"};
```

また、集合の内包で集合を表すこともできます。このような式を集合式と言います.

SetExpression:
{ *variable* | *booleanValue* : *inExpressionList* }

inExpressionList:
inExpression
inExpressionList : *inExpression*

inExpression:
variableList in *variable*
variableList in *setStatement*

例：

{ y | **true** : y **in** x : x **in** *set* }

上の集合式において，各変数は右から順に評価される．したがって，次のような記述は上と同じにはならない.

{ y | **true** : x **in** *set* : y **in** x }

集合演算子

和 (+)
`{"a"} + {"b"} => {"a", "b"}`
二つの集合の和集合を返す.

差 (-)
二つの集合の差集合を返す.

積 (*)
二つの集合の積集合を返す.

直積 (**)
`{"a","b"}**{"c","d"} => {{"a","c"},{"a","d"},{"b","c"},{"b","d"}}`
二つの集合の直積集合を返す.

べき (\)
`\{"a", "b"} => {{"a"}, {"b"}, {"a","b"}}`
集合のべき集合を演算する単項演算子

要素数 (#)

```
#{"a", "b"} => 2
```

集合の要素数を演算する単項演算子

存在 (in)

```
 a in {a, b, c, d} => true
```

要素が存在するかどうか.

集合演算式の値は，集合演算を行った結果を表すデータである．例えば，“{ 2 } + { 3 }”は，集合 {3} と集合 {2} の和集合 {3, 2} の値を持つ.

3.5 リスト型

同一の型を持つ一つ以上の値または変数から成るリストを表す型です。推論規則は，論理式型の値を一つ以上持つリストになります.

演算子

追加 (||)

```
"a" || "b" => < "a", "b" >
<"a", "b"> || "c" => < "a", "b", "c" >
```

リストの作成、要素の追加。

削除 (-)

```
 <1,2,3> - 3 => <1,2>
```

要素の削除。

要素数 (#)

```
#< 1, 2, 3 > => 3
```

要素数を返す。

要素 ([n])

```
<"a","b","c">[0] => "a"
<"a","b","c">[] => "c"
```

n 番の要素を返す。

サブリスト ([n..m])

```
 <a, b, c, d>[2..3] => <b, c>
```

n 番目から m 番目の要素を取り出す演算子.

存在 (in)

```
 a in <a, b, c, d> => true
```

要素が存在するかどうか.

3.6 認識状態型と認識過程型

認識状態が論理式の集合で、認識過程が認識状態のリストです．認識状態型が論理式型の拡張型で、認識過程が認識状態のリスト型の拡張型です．

me という認識過程型変数が組み込み変数として定義されています．認識過程型データを返す演算の結果が代入されていなければ，この変数に代入されます．

認識状態の演算子

前項 (<<n)

```
<a, b, c>[]<<2 => a
```

n 個前の認識状態．

後項 (>>n)

```
<a, b, c>[1]>>2 => c
```

n 個後の認識状態．

認識操作 (*)

適用された認識的操作を返す．

矛盾 (%)

認識状態に存在する矛盾の論理式の集合を返す．

認識過程の演算子

認識的演繹 (++)

```
me++
```

認識的拡張 (+)

```
me + a
```

論理式または論理式の集合 a を認識的拡張します．

認識的縮約 (-)

```
me - a
```

論理式または論理式の集合 a を認識的縮約します．

矛盾 (%)

認識過程のすべての認識状態に存在する矛盾の論理式の集合を返す．

例：

```
1 p = { _A_, _A -> B_ }; // 認識状態
2 me || { _A_, _A -> B_, _B->C_, _C->D_ }; // 認識過程
3 me++; // 認識的演繹
4 foreach b in me do
5   // { _A_, _A -> B_ }
```

```
6    // { _A_, _A -> B_, _B -> C_, _C -> D_ }
7    // { _A_, _A -> B_, _B -> C_, _C -> D_, _B_ }
8    print(b);
9  end;
10
11 // 最後まで演繹
12 while me++ do
13   // Modus Ponens only
14   // 1nd iteration: { _C_ }
15   // 2rd iteration: { _D_ }
16   print( me[] - me[-1] );
17 end;
18
19 // 認識的拡張
20 me = me + {_D->E_, _E->!A_, _E->F_, _F->G_};
21
22 // 矛盾が出るまで演繹
23 me++;
24 while #%me == 0 do
25   // 1st iteration: { _E_ }
26   print( me[] - me[-1] );
27   me++;
28 end;
29
30 // 2nd iteration: { _!A_, _F_ }
31 print( me[] - me[-1] );
32
33 foreach b in %me do
34   // { _A_, _!A_ }
35   print(b);
36   me - b; // 認識的縮約
37 end;
38
39 // {_A->B_,_B->C_,_C->D_,_B_,_C_,_D_,
40 // _D->E_,_E->!A_,E->F,_F->G_,_E_,_F_,_F}
41 print(me[]);
```

4　制御構造

EPLAS では，式が逐次的に実行されます．

制御構造は，条件分岐処理 **if** と反復処理 **while** と繰り返し処理 **foreach** があります．繰り

返し処理で実行する文の列中では，繰り返しを制御する **break** と **continue** がある．

4.1 if 文

if 文は，**else** ありとなしの 2 種類があります．

IfThenStatement:
if *Expression* then *Statement* end

IfThenElseStatement:
if *Expression* then *Statement* else *Statement* end

4.2 while 文

反復処理は，条件式を *conditional* とし，条件式の値が真でありつづけるとき，実行する文の列を *statements* とすると，

while *conditional* **do** *staetments* **end**

と記述される．

4.3 foreach 文

集合の各要素に対する繰り返し処理は，各要素を表す変数を *iterator* とし，各要素を持つ集合を *collection* とし，各要素に対し実行する文の列を *statements* とすると，

foreach *iterator* **in** *collection* **do** *statements* **end**

と記述される．

4.4 break 文

break は，反復処理や各構造型のデータの持つ各要素に対する繰り返し処理を終了し，次の処理に移る際に利用される．**break** は，終了させたい反復処理や各構造型のデータの持つ各要素に対する繰り返し処理の入れ子の数を N とすると，

break N

と記述される．N を省略すると，**break** 1 と同じであり，直近の反復処理や各構造型のデータの持つ各要素に対する繰り返し処理を終了させることを意味する．

4.5 continue 文

continue は，反復処理や各構造型のデータの持つ各要素に対する繰り返し処理を頭から，または次の要素に対して頭から処理する際に利用される．**continue** は，再度頭から処理したい反復処理や各構造型のデータの持つ各要素に対する繰り返し処理までの入れ子の数を N とすると，

continue N

と記述される．N を省略すると，**continue** 1 と同じであり，直近の反復処理や各構造型のデータの持つ各要素に対する繰り返し処理を再度頭から処理することを意味する．例えば，

```
while true
do
  while true
    ...
    continue 2;
    ...
  end;
end;
```

とすると，一番外側の反復処理を頭から再度行う．

5 関数及びライブラリ

5.1 関数呼び出し

関数は単項式である．関数は，

$$func$$

または，

$$func(exp_1, exp_2, \cdots, exp_i, \cdots, exp_n)$$

と表される．ただし，ここで $func$ は英字から始まる英数字の記号列である．前者は引数無しの関数である．後者は，$exp_i (0 \leq i \leq n)$ の値を引数とする関数である．関数の値は，定義された関数を引数があれば引数を与えて実行して返されるデータである．

5.2 関数定義

EPLAS では、関数（function）と手続き（procedure）が定義されています。関数は，決まった数の値を実引数として渡しながら呼び出すことができる実行可能コードを言います。手続きは、戻り値を持たない関数のことを言います。

5.2.1 関数

関数 $func$ を宣言し，文の列 $statements$ として定義する場合，

function $func$ [**with** $param_1, param_2, ...$] **do** $statements$ **end**

と記述する．“[ ]” は省略可能であり，任意個の引数をとれる．このときの $param_1, param_2, ...$ は，$statements$ で利用できる仮引数である．$statements$ 中では，かならず制御文 **return** が実行され，関数の値が返されなければならない．

関数中では，関数内の処理を終了し値を返す **return** がある．**return** は，返す値を表す式を exp とすると，

return exp

と記述される．

5.2.2 手続き

手続き $proc$ を宣言し，文の列 $statements$ として定義する場合，

procedure $proc$ [**with** $param_1, param_2, ...$] **do** $statements$ **end**

と記述する．“[ ]” は省略可能であり，任意個の引数をとれる．このときの $param_1, param_2, ...$ は，$statements$ で利用できる仮引数である．

6 式

式には、代入、制御構造、関数定義、関数呼び出しがあります。EPLAS プログラムは、これらの式を並べたものです。行終端子は、セミコロン (';') です.

6.1 評価

集合式の *inExpressionList* は，右から左へ評価します．それ以外の式はすべて，左から右へ評価します.

6.2 式の型

もし式が変数又は値を指示するならば，その式はコンパイル時に知らされる型をもつ。

式の型は，演算子の種類と関数の戻り値によって決定されます．例えば，`1 + 1` は整数型になります。

6.3 ライブラリ引用

外部プログラムファイルに定義されている定数や関数を利用するために，外部プログラムファイルをライブラリとして引用できます．各論理体系のライブラリを引用することで論理体系の選択ができます．ライブラリ引用の組み込み関数 `include`(*filename*) が定義されています．

例：

```
include(”abduction.ep”);
include(”model_logic.ep”);
```

7 組み込みライブラリ

7.1 組み込み手続き

print(i)

整数または実数または文字列またはを引数にとり，その値を標準 IO に出力する．

print(wff)

信念の整論理式 wff を引数にとり，その信念を標準 IO に出力します．

perform

手続きを引数としてとり，その手続きを実行する．

_addConnective(conn, func)

結合子と手続きを引数としてとり，新たな結合子を定義します．

_removeConnective(conn)

既存の結合子を無効化にします．

savepoint

`me` の最後の認識状態をセーブポイントとします．

restore

もっとも最近セーブした認識状態を me に追加します.

7.2 組み込み関数

read(filename)

ファイルから信念を読み込み，信念集合を返す.

read

標準入力から入力された信念を返す.

generalizeRule(pre, post)

$A_1, A_2, \cdots, A_n \vdash B_1, B_2, \cdots, B_n$ 推論規則を $\{\ A_1, A_2, \cdots, A_n \vdash B_1,\ A_1, A_2, \cdots, A_n \vdash B_2,\ \cdots\ A_1, A_2, \cdots, A_n \vdash B_n\ \}$ に汎化する.

input_rule

標準入力から入力された推論規則を返す.

input_number

標準入力から入力された整数を返す.

input_real

標準入力から入力された実数を返す.

input_string

標準入力から入力された文字列を返す.

a.derivation

信念 a の導出木中に含まれる全ての信念を返す.