EPLAS Language Specification ver.1.0

AISE Lab., Saitama University
eplas@aise.ics.saitama-u.ac.jp

Last modified: Aug. 2010

Change Log

Version	Date	Maintainers	Comments
1.0	2010/08/15	Wencheng FANG <wencheng@aise>	Initial version

Contents

1 Preface

1.1 Introduction

EPLAS is a strongly dynamically typed language.

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

1.2 An Example Program

> There are six doors numbered 1 to 6, each of which leads to heaven or hell. In front of each door, there is a doorman. Three of them are liars and the door guarded by them are led to hell. Every doorman testified that "The door next to me is not leading to heaven". The doorman of the door 1 replied "No" to the question "Is door 4 leading to heaven?". The doorman of the door 2 replied "No" to the question "Is door 6 leading to heaven?". So which is the road to heaven?

The program written in EPLAS to solve the above logic puzzle "The Road to Heaven" by Reductio ad absurdum is shown below.

```
1  read();
2  assumptions = read();
3  // pick any 3 of 6 doormen
4  assumptions = { x | #x == 3 : x in \assumptions };
5  savepoint;
6  foreach x in assumptions do
7      restore;
8      me + x;
9      while me++ do
10         // get all possible conclusions
11     end;
12     if %me then
13         // if inconsistent, try next combination
14         continue;
15     else
16         print(me[]);
17         break;
18     end;
19 end;
```

Figure 1: Program to solve a logic puzzle

2 Grammar

2.1 The Lexical Grammar

EPLAS syntax is defined by context-free grammar.

Terminal symbols are shown in `fixed width` font in the productions of the lexical and syntactic grammars.

Nonterminal symbols are shown in italic type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example:

IfThenStatement:
`if` *Expression* `then`
Statement

2.1.1 Identifiers

An identifier is an unlimited-length sequence of letters and digits, the first of which must be a letter. An identifier cannot have the same spelling as a keyword.

2.1.2 Comments

A traditional comment: all the text from the ASCII characters /* text from the ASCII characters /* to the ASCII characters */ is ignored (as in C and C++).

A end-of-line comment: all the text from the ASCII characters // to the end of the line is ignored (as in C++)

2.1.3 Keywords

The following character sequences are reserved for use as keywords.

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

2.2 Literals

A literal is the source code representation of a value of a primitive type like integer `1` or string `"hello world"`;

2.2.1 Number Literals

A number literal may be expressed in decimal (base 10) or float-point.

For example:

```
123
-123
123.45
1.40e-45
```

2.2.2 String Literals

A string literal consists of zero or more characters enclosed in double quotes. For example:

```
""
"a"
"hello world!"
```

2.2.3 Boolean Literals

The boolean type has two values, represented by the boolean literals `true` and `false`.

2.2.4 Logic Formula Literals

A logic formula literal consists of characters enclosed in underscores. For example:

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

The grammar of well-formed formulas is described in subsection 3.3.

2.2.5 Set Literals

A set literal is written as a comma-separated list of expressions, enclosed by braces "{" and "}". The order of expressions will be ignored.

SetStatement:
{ *expressionList* }

For example:

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

2.2.6 List Literals

A list literal is written as a comma-separated list of expressions, enclosed by braces "<" and ">". By contrary to sets, the order of expression in lists will be preserved.

ListStatement:
< *expressionList* >

2.3 Operators

The following 37 tokens are the operators, formed from ASCII characters:

```
*, /, +, -

#, $, \\, **

&&, ||, !

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

$$, %%
```

3 Variables and Types

3.1 Variables and Assignments

AssignmentStatement:
variable = *expression*

The type of a variable is determined by the first value assigned to that variable without an explicit declaration. It is a runtime error if the value in the post assignments is not same as that type.

A variable, the first character of whose name is an uppercase, is a constant variable. A constant variable may only be assigned to once. For example:

```
1 ConstValue = "a␣constant␣string";
2 CONST1 = "another␣constant␣string";
3 CONST1 = "re-define␣constant"; // Runtime error
```

3.2 Primitive Types

The primitive types predefined in EPLAS is number type, string type, boolean type.

3.2.1 The Number Type

The number type represents 64bit integers and floating-points. The values of the integers are in from -9223372036854775808 to 9223372036854775807. The values of the floating-points follow double-precision 64-bit format IEEE 754 values.

EPLAS provides a number of operators that act on numeric values:

The numerical operators

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

The comparison operators

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

3.2.2 The String Type

The string type represent sequences of unicode characters.

Operations

Concatenation(+)

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

concatenate two string.

Length(#)

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

get the length of a string.

CharAt([n])

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

get the nth character of a string or the last one if n is omitted.

Substring([n..m])

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

get a string consists the characters from the nth to the mth in a string.

Conversion($)

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

convert any value of other type to a string.

3.2.3 The Boolean Type

The boolean type represents a logical quantity with two possible values, indicated by the literals true and false. The boolean operators are:

Equal(==)

`a == b`

a equals b.

Not Equal(!=)

`a != b`

a is contrary to b.

3.3 The Logic Formula Type

Table 1: The vocabulary of logic formulas

Vocabulary	Definition
Atoms	Identifiers of EPLAS
Functions	Identifiers of EPLAS
Connectives	Symbols except letters and digits
Quantifiers	Universal`@` and Existential`#`
Separators	Braces`()` and Comma`,`

The grammar is shown below:

LogicFormula:

Atom
Function
LogicFormula LogicalConnective LogicFormula
`@` *identifier (LogicFormula)*
`#` *identifier (LogicFormula)*

Function:

Identifier (*identiferList*)

Atom:

Identifier

The production rule of well-formed formulas is customizable. EPLAS provides the second-order predicate logic as the default logic system. The negation(`~`), logical conjunction(`&`) and entailment(`=>`) are predefined. For example:

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

Unary Operators

Negation(~)

`~b`

the negation of formula b.

Predicate(*)

`*pre(b)`

predicate pre.

Universal(@)

`@x(b,2)`

universal quantify formula b by variable x.

Existential(#)

`#x(b,2)`

existential quantify formula b by variable x.

Assign(`[t/x]`)

`b["a"/x]`

assign "a" to variable x in formula b.

String($)

`$b`

take formula b as a string.

Comparison Operators

Equal(==)

`a == b`

formula a and b are same.

Equal(`!=`)

`a != b`

formula a and b are NOT same.

Regex(~=)

`a ~= b`

formula a matchs regular expression b.

Contradiction($$)

`a $$ b`

an alias of `a == ~b`.

3.4 The Set Type

A variable of set type holds literals or variables of same types.

Here is the declaration of a set of the string type.

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

A set can be created by a intensional expression.

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

inExpressionList:
inExpression
inExpressionList : *inExpression*

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

For example:

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

Operations

Sum(+)
`{"a"} + {"b"} => {"a", "b"}`
sum of two sets.

Difference(-)
difference of two sets.

Intersection(*)
intersection of two sets.

Cartesian production(**)
`{"a","b"}**{"c","d"} => {{"a","c"},{"a","d"},{"b","c"},{"b","d"}}`
cartesian production of two sets.

Power set(\)
`\{"a", "b"} => {{"a"}, {"b"}, {"a","b"}}`
power set of two sets.

Size(#)
`#{"a", "b"} => 2`
size of the set.

Existension(`in`)

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

determine the existension of a element or a subset.

Each operation of set introduces a new set value. For example, "`{ 2 } + { 3 }`" introduces a new set `{ 3, 2 }`.

3.5 The List Type

A variable of list type holds literals or variables of same types in which the order of elements are preserved.

Operations

Append(`||`)

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

append elements to a list.

Delete(-)

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

remove elements from a list.

Size(#)

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

size of a list.

Element(`[n]`)

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

get nth element.

Sublist(`[n..m]`)

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

get a sublist that consists elements from nth to mth

Existension(`in`)

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

determine the existension of a element or a sublist.

3.6 The Epistemic State Type and the Epistemic Process Type

A epistemic state is a set of logic formulas. A epistemic process is a list of epistemic states. EPLAS provides the epistemic state type as an extensional type of the set type and the epistemic process type as an extensional type of the list type.

The keyword `me` denotes a value that hold a epistemic process. A operation which returns a value of the epistemic process type is assigned to `me` if the operation is lack of the assignment.

Operations of the epistemic state type

Previous(<<n)
`<a, b, c>[]<<2 => a`
get the epistemic state *n*th before.

Afterward(>>n)
`<a, b, c>[1]>>2 => c`
get the epistemic state *n*th after.

Epistemic operation(*)
get the epistemic operations applied to.

Contradiction(%)
get a set of inconsistent logic formulas that included in the epistemic state.

Operations of the epistemic process type

Epistemic deduction(++)
`me++`

Epistemic expansion(+)
`me + a`
do epistemic expansion by formulas a.

Epistemic contraction(-)
`me - a`
do epistemic contraction by formulas a.

Contradiction(%)
get a set of inconsistent logic formulas that included in all epistemic states of the epistemic process.

For example:

```
1 p = { _A_, _A -> B_ }; // a epistemic state
2 me || { _A_, _A -> B_, _B->C_, _C->D_ }; // epistemic process 'me'
```

```
3  me++; // epistemic deduction
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 while me++ do
12   // Modus Ponens only
13   // 1nd iteration: { _C_ }
14   // 2rd iteration: { _D_ }
15   print( me[] - me[-1] );
16 end;
17
18 // epistemic expansion
19 me = me + {_D->E_, _E->!A_, _E->F_, _F->G_};
20
21 me++;
22 while #%me == 0 do
23   // 1st iteration: { _E_ }
24   print( me[] - me[-1] );
25   me++;
26 end;
27
28 // 2nd iteration: { _!A_, _F_ }
29 print( me[] - me[-1] );
30
31 foreach b in %me do
32   // { _A_, _!A_ }
33   print(b);
34   me - b; // epistemic contraction
35 end;
36
37 // {_A->B_,_B->C_,_C->D_,_B_,_C_,_D_,
38 // _D->E_,_E->!A_,E->F,_F->G_,_E_,_F_,_F}
39 print(me[]);
```

4 Control Flow Statements

In EPLAS, statements are executed sequentially.

4.1 The if Statement

The if statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

IfThenStatement:
`if` *Expression* `then` *Statement* `end`

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

4.2 The while Statement

The while statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is false.

WhileStatement:
`while` *Expression* `do` *Statement* `end`

4.3 The foreach Statement

A foreach statement is an iterative execution for sets or lists.

ForeachStatement:
`foreach` *Identifier* `in` *ListOrSetStatement* `do` *Statement* `end`

4.4 The break Statement

A break statement transfers control out of an enclosing statement and then immediately completes normally.

A break statement with a numeric value N attempts to transfer control N times to the outer enclosing statement in order, innermost to outermost.

A break statement with no numeric values attempts to transfer control to the innermost enclosing statement as if N is 1.

BreakStatement:
`break` *Number*

4.5 The continue Statement

A continue statement transfers control out of an enclosing statement and then immediately ends the current iteration and begins a new one.

A continue statement with a numeric value N attempts to transfer control N times to the outer enclosing statement in order, innermost to outermost.

A continue statement with no numeric values attempts to transfer control to the innermost enclosing statement as if N is 1.

ContinueStatement:
`continue` *Number*

For example:

```
1 while true
2 do
3   ...
4   while true
5     ...
6     continue 2; // start a new iteration from line 3
7   end;
8 end;
```

4.6 The return Statement

A return statement returns control to the invoker of a function or terminates the program if it is used in the main program.

ReturnStatement:
`return` *Expression*

5 Functions

5.1 Function Invocation Expressions

A function invocation expression is used to invoke a function. Variables and functions can share same identifers.

$$func()$$

or

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

5.2 Function Declarations

A function declares executable code that can be invoked, passing a fixed number of values as arguments. Two kinds of functions with or without return values are both defined in EPLAS. A function without return values is also called procedure.

5.2.1 Functions

FunctionDeclaration:
`function` *Identfier* `with` *ParameterList* `do` *Statement* `end`

It is a runtime error if a function does not return a value before the ends of function.

5.2.2 Procedures

ProcedureDeclaration:
`procedure` *Identfier* `with` *ParameterList* `do` *Statement* `end`

6 Expressions

Expressions in EPLAS are assignments, function invocations, function declarations and control flows. A EPLAS program is a list of expressions. Line terminator is a semicolon(';').

6.1 Evaluation Order

EPLAS guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

The intensional expression of the set type is evalutated from right to left. For example, the results of the following two expressions are different:

$\{\ y \mid \textbf{true} : y\ \textbf{in}\ x : x\ \textbf{in}\ set\ \}$

$\{\ y \mid \textbf{true} : x\ \textbf{in}\ set : y\ \textbf{in}\ x\ \}$

6.2 Type of an Expression

If an expression denotes a variable or a value, then the expression has a type known at compile time. For example, the expression `1 + 1` results a value of the number type.

7 Libraries

7.1 Code Re-use

To load extensional libraries, which include reusable functions or definitions of logic systems, EPLAS provide a predefined function `include(`*filename*`)`. The *filename* is a string to demand the library file to load. EPLAS tries to load the library, returning true if successful.

For example:

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

7.2 Predefined Procedures

print(i)

Outputs *i*, which is either a number or a string, to standard I/O.

print(wff)

Outputs *wff*, which is a logic formula, to standard I/O.

perform(func)

Invokes function *func*.

_addConnective(conn, func)

Declares a new connective, whose syntax is defined by *conn* and semantics is defined by *func*.

_removeConnective(conn)

Disables pre-defined connective *conn*.

savepoint

Saves the last epistemic state of `me`.

restore

Restores the last saved epistemic state to `me`.

7.3 Predefined Functions

read(filename)

Reads logic formulas from file named *filename*, returning a set of logic formulas.

read

Reads logic formulas from standard I/O, returning a set of logic formulas.

generalizeRule(pre, post)

Generalizes a inference rule $\{A_1, A_2, \cdots, A_n\} \vdash \{B_1, B_2, \cdots, B_n\}$ to a set of inference rules: { <$A_1, A_2, \cdots, A_n, B_1$>, <$A_1, A_2, \cdots, A_n, B_2$>, $\cdots$ <$A_1, A_2, \cdots, A_n, B_n$> } , in which *pre* is $\{A_1, A_2, \cdots, A_n\}$ and *post* is $\{B_1, B_2, \cdots, B_n\}$.

input_rule

Reads a inference rule from standard I/O.

input_number

Reads a integer from standard I/O.

input_real

Reads a floating-point from standard I/O.

input_string

Reads a string from standard I/O.

a.derivation

Returns a set of all logic formulas in the derivation tree of formula *a*.