LoopdeeDo

A simple expression-oriented language that
features a unique REPL loop tool

Document and Language designed and prepared by
Thomas Famularo, Michael Gutierrez, Kai Wong

Contribution Log for Document:

Kai

- Wrote rough draft for all the various required sections
- Wrote out grammatical specification for arithmetic expressions, and logical and relational operators.
- Wrote example programs for parts of language that I did (arithmetic, logical, relational)
    - Added more example programs for changed syntax, save, loop, run, variable assignment. Updated tokens
- Wrote out lexical specification for parts of language that I did that used them
- Wrote out type system and type rules for parts I contributed, also did type rules for Function, Do, Call
- Changed all semantic rules to reflect store passing

Thomas

- Wrote out grammatical specifications for do, fn and call
- Wrote example programs involving do, fn and call
- Wrote out information about Free Store

Michael

- Wrote out specification for Bool Expressions
    - Added example programs
    - Added Type Specifications
- Wrote out specification for If Expressions
    - Added example programs
    - Added Type Specifications
- Wrote out specification for reading from standard input
    - Added Example programs
    - Added Type Specifications
- Wrote out specification for print expressions
    - Added example programs
    - Added Type Specifications
- Wrote out specification for String Val
    - Added example programs
    - Added Type Specifications
- Wrote out specification for List expressions
    - Added Example programs
    - Added Type Specifications
- Wrote out specification for Cons, Head, & Tail expressions with Kai
    - Added Example programs
    - Added Type Specification for Cons, Head, & Tail

I. Proposal

A. Motivation

LoopdeeDo is based off of the LET language originally designed by Mitchell Wand. It makes improvements and changes to the LET language syntax, while also providing additional functions and operators that add more use value to the language. Some improvements include more logical and concise syntax. It contains unique features, such as a loop function in which the user can continuously enter parameter values and get function results, and being able to read in a program from a file.

B. Values

LoopdeeDo consists of several native types of expressed values:

- Integer (NumVal)
- Boolean (BoolVal)
- String (StringVal)
- Function (FunctionVal)
- List (ListVal)

We introduce a StoVal, which consists of expressed values.
All denoted values are references to addresses in the free store.
There is a global environment/store and a local environment/store, and they support scoping.

C. Built-in Operators / Functions

- Integer Arithmetic (+, -, *, /, %) (Right associative)
- Integer Predicate (zero?, <, >, <=, >=, ==)
- Boolean Predicate (&&, ||)
- List Operations (head, tail, cons)
    - cons edits its second argument in memory

D. Control Mechanisms

Variables can be declared and bound to a value in the environment using the “do” expression. In addition, if-then-else expressions provide conditional branching. Variables can be declared globally using the “bind” expression.

E. Type System

Currently, LoopdeeDo is an untyped language, in which variables can be assigned any kind of value and operations will fail at run time if given the wrong kind of data.

However, in the future, a type system and type rules will be implemented. The type system and type rules are as follows:

Let v be an expressed value

- v has type IntType iff v is a NumVal
- v has type BoolType iff v is a BoolVal
- v has type (t_1 -> t_2) iff v is a FunctionVal, which when passed an arg of type t_1 at call-time, either will (a) returns a value of type t_2, or (b) fail to terminate, or (c) terminate for a reason other than a type error
- v has type StringType iff v is a StringVal
- v has type ListType iff v is a ListVal

The type environment is represented as the greek letter tau (τ)

---

typeOf (ConstExp n) τ = IntType

---

typeOf (StringExp s) τ = StringType

---

typeOf (BoolExp b) τ = BoolType

---

typeOf (NegExp n) τ = IntType

---

typeOf (VarExp var) τ = τ (var)

---

typeOf exp_1 τ = IntType

---

typeOf (IsZeroExp exp_1) τ = BoolType

---

typeOf exp_1 τ = IntType
typeOf exp_2 τ = IntType

---

typeOf (DiffExp exp_1 exp_2) τ = IntType

typeOf exp_1 τ = IntType
typeOf exp_2 τ = IntType

---

typeOf (AddExp exp_1 exp_2) τ = IntType

typeOf exp_1 τ = IntType
typeOf exp_2 τ = IntType

---

typeOf (MultExp exp_1 exp_2) τ = IntType

typeOf exp_1 τ = IntType
typeOf exp_2 τ = IntType

---

typeOf (DivExp exp_1 exp_2) τ = IntType

typeOf exp_1 τ = IntType
typeOf exp_2 τ = IntType

---

typeOf (ModExp exp_1 exp_2) τ = IntType

typeOf exp_1 τ = IntType
typeOf exp_2 τ = IntType

---

typeOf (RelationalExp exp_1 exp_2) τ = BoolType

typeOf exp_1 τ = BoolType
typeOf exp_2 τ = BoolType

---

typeOf (LogicalExp exp_1 exp_2) τ = BoolType

typeOf exp_1 τ = (t_1 -> t_2)
typeOf exp_2 τ = t_1
typeOf (CallExp exp_1 exp_2) = t_2

---

typeOf (LoopExp exp_1) τ = StringType

---

typeOf (SaveExp exp_1) τ = StringType

typeOf exp_1 τ = (t_1 -> t_2)
typeOf exp_2 τ = t_1
typeOf (CallExp exp_1 exp_2) = t_2

---

typeOf (RunExp exp_1) τ = StringType

typeOf exp_1 τ = BoolType
typeOf exp_2 τ = t_1
typeOf exp_3 τ = t_1

---

typeOf (IfExp exp_1 exp_2 exp_3) τ = t_1

typeOf exp_1 = t
typeOf id τ = τ (var)
τ' = τ .extendTEnv(var, t)

---

typeOf (DoExp id exp_1 exp_2) τ = typeOf exp_2 τ'

τ' = extendTEnv param t_{arg} τ
typeOf body τ' = t_{res}

---

typeOf (FunctionExp param t_{arg} body) τ = (t_{arg} -> t_{res})

typeOf exp_1 τ = (t_{arg} -> t_{res})
typeOf exp_2 τ = t_{arg}

---

typeOf (CallExp exp_1 exp_1) τ = t_{res}

typeOf exp_1 τ = t_1

---

typeOf (PrintExp exp_1) τ = t_1

typeOf file τ = t_1

---

typeOf (ReadExp file) τ = t_1

---

typeOf (ListExp) τ = ListType

typeOf exp_1 τ = NumType

---

typeOf (ListExp exp_1) τ = ListType

typeOf exp_2 τ = ListType
typeOf exp_1 τ = IntType

---

typeOf (ListExp exp_1 exp_2) τ = ListType

typeOf(exp_1) τ = ListType

---

typeOf (HeadExp exp_1) τ = IntType

typeOf(exp_1) τ = ListType

---

typeOf (TailExp exp_1) τ = ListType

typeOf(exp_2) τ = ListType
typeOf(exp_1) τ = ListType

---

typeOf (ConsExp exp_1 exp_2) τ = ListType

F. Example Programs

Input	Output
`5`	5
`"hey"`	"hey"
`bind x = 5` `(x - 5)`	5 0
`zero?(4)`	false
`((5-5)-5)`	-5 (removing right associativity)
`(5*5-2)`	23
`((5-2)*3)`	9
`((5-19) < 8)`	true
`(5 >= 8)`	false
`(zero?(4)||false||zero?(0))`	true
`do (x - 3) x = 5`	2
`do (f,4) f = fn(x)zero?(x)`	false
`read input.txt`	Output varies with what program file contains
`print "hello"`	"hello"

`loop fn(x)(x-5)`
`3`
`5`
	"Program fn(x) (x-5) is running..."
	-2
	0

`save fn(x)(x*3)`
`run saved`
`3`
`5`
	"Function saved"
	"Program fn(x) (x*3) is running…"
	9
	15

`bind x = [1,2,3]` `head x`	[1,2,3] 1

`tail x`
`cons ([4],x)`
`x`
	[2,3]
	[1,2,3,4]
	[1,2,3,4]

`do x y = do cons([1],x) x = [1,2,3]`	(should fail. Good! Final call to x is out of scope)
`do x x = do cons([1],x) x = [5,6]`	[5,6,1]
`do do (y - x) y = do x x = 10 x = 5`	5

G. Notes about Syntax

- The “loop”, “save”, and “run saved” programs are standalone programs; they cannot be used as sub-expressions for other programs
- The input.txt file can contain a program
- There are order of operations for arithmetic and logical expressions
- Arithmetic expressions are right associative; use parentheses to combat this
- Arithmetic, logical, and relational expressions are surrounded with parentheses
- You may only perform the cons operation on values with ListType

II. Formal Syntax

A. Overview

Italics denote a non-terminal symbol. Kleene Star and Plus are used for repetition, and pipe for alternation. Parentheses indicate groupings except for TRParen and TLParen. Tilde indicates anything other than.

B. Lexical Specification

Digit ::= [0-9]
TNum ::= *Digit*+
Bool ::= true | false
Char ::= [a-z]
TVar ::= *Char* (*Char* | *Digit* | _ | *?* | ‘)*
TComma ::= ,
TRParen ::=)

TLParen ::= (
TLsqBracket ::= [
TRsqBracket ::=]
THead ::= head
TTail ::= tail
TCons ::= cons
TAssign ::= =
TElse ::= else
TThen ::= then
TIf ::= if
TDo ::= do
TBool ::= *Bool*
TPrint ::= print
TRead ::= read
TSave ::= save
TRun ::= run
TSaved ::= saved
TLoop = loop
TBind = bind
TTextFile ::= (< CHAR >)(< CHAR > | < DIGIT >)* .txt
TString ::= “ ("\\" ("\\" |"\r"| "\n" | "\"") | ~["\\","\n","\""])* ”
TFunction ::= fn
TIsZero ::= zero?
TMinus ::= -
TPlus ::= +
TTimes ::= *
TDivide ::= /
TMod ::= %
TGreaterEqual ::= >=
TLessEqual ::= <=
TGreater ::= >
TLess ::= <
TEquals ::= ==
TAnd ::= &&
TOr ::= ||

C. Grammatical Specification

Program ::= *Expression* | Pgm (exp)
::= *Loop* | Pgm (exp)
::= *SaveRun* | Pgm (exp)
Expression ::= *Const*
::= *Var*

::= *Bool*
::= *String*
::= *Neg*
::= *Do*
::= if *Exp* then *Exp* else *Exp*
::= *Print*
::= *Read*
::= (*Arithmetic)*
::= (*Relational)*
::= (*Logical)*
::= zero? (*Expression*) | IsZeroExp (rand)
::= *Function*
::=*Call*
::= *Assign*
::= *List*
::= *Head*
::= *Cons*

Relational ::= *RelationalTerm* >= *RelationalTerm* | GreaterEqualExp (e1 e2)
::= *RelationalTerm* <= *RelationalTerm* | LessEqualExp (e1 e2)
::= *RelationalTerm* > *RelationalTerm* | GreaterExp (e1 e2)
::= *RelationalTerm* < *RelationalTerm* | LessExp (e1 e2)
::= *RelationalTerm* == *RelationalTerm* | EqualsExp (e1 e2)

RelationalTerm ::= *Expression*

Logical ::= *LogicalSub*
::= *LogicalSub* && *Logical*

LogicalSub ::= *LogicalTerm*
::= *LogicalTerm* || *LogicalSub*

LogicalTerm ::= *Expression*

Arithmetic ::= *Multiplicative* - *Arithmetic* | DiffExp (e1 e2)
::= *Multiplicative* + *Arithmetic* | AddExp (e1 e2)
::= *Multiplicative*

Multiplicative ::= *Factor* * *Multiplicative* | MultExp (e1 e2)
::= *Factor* / *Multiplicative* | DivExp (e1 e2)
::= *Factor* % *Multiplicative* | ModExp (e1 e2)
::= *Factor*

Factor ::= *Expression*

Const ::= *Number* | ConstExp (num)

Neg ::= - *Number* | NegExp (num)

Var ::= *Identifier* | VarExp (id)

Bool ::= *Boolean* | BoolExp (bool)

String ::= *String* | StringExp (str)

Print ::= print *Exp* | PrintExp (exp)

Read ::= read *TextFile* | ReadExp (t)

Loop ::= loop *Function* | LoopExp (exp)
SaveRun ::= *Save*
::= *Run*
Save ::= save *Function* | SaveExp (exp)
Run ::= run saved | RunExp
Assign ::= bind *Identifier* = *Expression* | AssignExp (id exp)
Function ::= fn(*Identifier*) *Expression*
Call ::= (*Expression , Expression)*
List ::= [ ] | ListExp()
::= [*Const*] | ListExp (exp)
::= [*Const*, *List*] | ListExp (exp, list)
Do ::= do *Expression Identifier* = *Expression* | DoExp (id exp1 exp2)
Head ::= head *Expression* | HeadExp (exp)
Tail ::= tail *Expression* | TailExp (exp)
Cons ::= cons (*Expression* , *Expression*) | ConsExp (exp1, exp2)

III. Formal Semantics

A. Operational Semantics in Host-Specific Language (Java)

The inference rules below specify behavior of the LoopdeeDo language in terms of the Java language that is used as the host language for implementation. A global environment and store are denoted as env and sto below respectively. A local environment and store that is passed between expressions denoted as passedEnv and passedSto below respectively.

Constant Expressions

---

valueOf (ConstExp num) passedEnv passedSto = return new NumVal(Integer.parseInt(((ConstExp)exp).getToken().toString()));

Variable Expressions (supports scoping. Local shadows global)

---

valueOf (VarExp var) passedEnv passedSto = return (ExpVal) sto.deRef(env.applyEnv(t));

---

valueOf (VarExp var) passedEnv passedSto = return (ExpVal)
passedSto.deRef(passedEnv.applyEnv(t));

String Expressions

---

valueOf(StringExp exp) passedEnv passedSto = return new
StringVal(((StringExp)exp).getToken().toString());

Arithmetic Expressions

```
NumVal n1 = (NumVal)valueOf(((DiffExp)exp).getExp1(), passedEnv, passedSto)
NumVal n2 = (NumVal)valueOf(((DiffExp)exp).getExp2(), passedEnv, passedSto)
```

---

valueOf (DiffExp exp) passedEnv passedSto = return new NumVal(n1.getNumVal() -
n2.getNumVal());

```
NumVal n1 = (NumVal)valueOf(((AddExp)exp).getExp1(), passedEnv, passedSto)
NumVal n2 = (NumVal)valueOf(((AddExp)exp).getExp2(), passedEnv, passedSto)
```

---

valueOf (AddExp exp) passedEnv passedSto = return new NumVal(n1.getNumVal() +
n2.getNumVal());

```
NumVal n1 = (NumVal)valueOf(((MultExp)exp).getExp1(), passedEnv, passedSto)
NumVal n2 = (NumVal)valueOf(((MultExp)exp).getExp2(), passedEnv, passedSto)
```

---

valueOf (MultExp exp) passedEnv passedSto = return new NumVal(n1.getNumVal() *
n2.getNumVal());

```
NumVal n1 = (NumVal)valueOf(((DivExp)exp).getExp1(), passedEnv, passedSto)
NumVal n2 = (NumVal)valueOf(((DivExp)exp).getExp2(), passedEnv, passedSto)
```

---

valueOf (DivExp exp) passedEnv passedSto = return new NumVal(n1.getNumVal() /
n2.getNumVal());

```
NumVal n1 = (NumVal)valueOf(((ModExp)exp).getExp1(), passedEnv, passedSto)
NumVal n2 = (NumVal)valueOf(((ModExp)exp).getExp2(), passedEnv, passedSto)
```

```
valueOf (ModExp exp1 exp2) passedEnv passedSto =  return new NumVal(n1.getNumVal() %
n2.getNumVal());
```

Predicate Expressions

```
ExpVal e = valueOf (((IsZeroExp)exp).getExp(), passedEnv, passedSto)
```

```
valueOf (IsZeroExp exp) passedEnv passedSto = return new BoolVal(e.getNumVal() == 0);
```

```
ExpVal e1 = valueOf(((GreaterEqualExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((GreaterEqualExp)exp).getExp2(), passedEnv, passedSto)
```

```
valueOf(GreaterEqualExp exp) passedEnv passedSto = return new BoolVal(e1.getNumVal() >=
e2.getNumVal());
```

```
ExpVal e1 = valueOf(((LessEqualExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((LessEqualExp)exp).getExp2(), passedEnv, passedSto)
```

```
valueOf(LessEqualExp exp) passedEnv passedSto = return new BoolVal(e1.getNumVal() <=
e2.getNumVal());
ExpVal e1 = valueOf(((GreaterExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((GreaterExp)exp).getExp2(), passedEnv, passedSto)
```

```
valueOf(GreaterExp exp) passedEnv passedSto = return new BoolVal(e1.getNumVal() >
e2.getNumVal());
```

```
ExpVal e1 = valueOf(((LesslExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((LessExp)exp).getExp2(), passedEnv, passedSto)
```

```
valueOf(LessEqualExp exp) passedEnv passedSto = return new BoolVal(e1.getNumVal() <
e2.getNumVal());
```

ExpVal e1 = valueOf(((EqualsExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((EqualsExp)exp).getExp2(), passedEnv, passedSto)

---

valueOf(EqualsExp exp) passedEnv passedSto = return new BoolVal(e1.getNumVal() == e2.getNumVal());

ExpVal e1 = valueOf(((AndExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((AndExp)exp).getExp2(), passedEnv, passedSto)
e1.getBoolVal() && e2.getBoolVal() == true

---

valueOf(AndExp exp) passedEnv passedSto = return new BoolVal(true);

ExpVal e1 = valueOf(((AndExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((AndExp)exp).getExp2(), passedEnv, passedSto)
e1.getBoolVal() && e2.getBoolVal() == false

---

valueOf(AndExp exp) passedEnv passedSto = return new BoolVal(false);

ExpVal e1 = valueOf(((OrExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((OrExp)exp).getExp2(), passedEnv, passedSto)
e1.getBoolVal() || e2.getBoolVal() == true

---

valueOf(OrExp exp) passedEnv passedSto = return new BoolVal(true);

ExpVal e1 = valueOf(((OrExp)exp).getExp1(), passedEnv, passedSto)
ExpVal e2 = valueOf(((OrExp)exp).getExp2(), passedEnv, passedSto)
e1.getBoolVal() || e2.getBoolVal() == false

---

valueOf(OrExp exp) passedEnv passedSto = return new BoolVal(false);

Conditional Expressions

BoolVal e1 = (BoolVal)valueOf(((IfExp)exp).getExp1(), passedEnv, passedSto);
e1 == true;

---

valueOf(IfExp exp1 exp2 exp3) passedEnv passedSto = return valueOf(((IfExp)exp).getExp2(), passedEnv, passedSto);

BoolVal e1 = (BoolVal)valueOf(((IfExp)exp).getExp1(), passedEnv, passedSto);
e1 == false;

---

valueOf(IfExp exp1 exp2 exp3) passedEnv passedSto = return valueOf(((IfExp)exp).getExp3(), passedEnv);

List Expressions

---

valueOf(ListExp) passedEnv passedSto = return new ListlVal();

NumVal n1 = (NumVal) valueOf(((ListExp)exp).getExp1(), passedEnv, passedSto);

---

valueOf(ListExp exp1) passedEnv passedSto = return new ListlVal(n1);

ListVal templist = (ListVal)valueOf(((ListExp)exp).getExp2(), passedEnv, passedSto);
NumVal n1 = (NumVal) valueOf(((ListExp)exp).getExp1(), passedEnv, passedSto);
exp1 != null && exp2 != null

---

valueOf(ListExp exp1 exp2) passedEnv passedSto =
return new ListVal(n1, list.getListVal().addAll(templist.getListVal()));

ListVal templist = (ListVal)valueOf(((HeadExp)exp).getExp1(), passedEnv, passedSto);
LinkedList list = (LinkedList)templist.getListVal();
list.size() != 0;

---

valueOf(HeadExp exp) passedEnv passedSto = return (NumVal)templist.getListVal().get(0);

ListVal templist = (ListVal)valueOf(((HeadExp)exp).getExp1(), passedEnv, passedSto);
LinkedList list = (LinkedList)templist.getListVal();
list.size() == 0;

---

valueOf(HeadExp exp) passedEnv passedSto = return new StringVal("Head of empty list is nothing");

```
ListVal list = new ListVal();
ListVal templist = (ListVal)valueOf(((TailExp)exp).getExp1(), passedEnv, passedSto);
list.getListVal().addAll(templist.getListVal());
list.size() == 0;
```

```
valueOf(TailExp exp) passedEnv passedSto = return list.getListVal().remove(0);
```

```
ListVal list = new ListVal();
ListVal templist = (ListVal)valueOf(((TailExp)exp).getExp1(), passedEnv, passedSto);
list.getListVal().addAll(templist.getListVal());
list.size() == 0;
```

```
valueOf(TailExp exp) passedEnv passedSto = return new StringVal("Tail of empty list is
nothing");
```

```
ListVal list = new ListVal();
ListVal templist1 = (ListVal)valueOf(((ConsExp)exp).getExp1(), passedEnv, passedSto);
ListVal templist2 = (ListVal)valueOf(((ConsExp)exp).getExp2(), passedEnv, passedSto);
list.getListVal().addAll(templist2.getListVal());
list.getListVal().addAll(templist1.getListVal());
((ConsExp)exp).getExp2() instanceof VarExp;
VarExp v = (VarExp)((ConsExp)exp).getExp2();
Integer addr = passedEnv.applyEnv(v.getVar());
passedSto.setRef(addr, list);
```

```
valueOf(ConsExp exp) passedEnv passedSto = return list;
```

```
ListVal list = new ListVal();
ListVal templist1 = (ListVal)valueOf(((ConsExp)exp).getExp1(), passedEnv, passedSto);
ListVal templist2 = (ListVal)valueOf(((ConsExp)exp).getExp2(), passedEnv, passedSto);
list.getListVal().addAll(templist2.getListVal());
list.getListVal().addAll(templist1.getListVal());
((ConsExp)exp).getExp2() instanceof VarExp;
VarExp v = (VarExp)((ConsExp)exp).getExp2();
Integer addr = env.applyEnv(v.getVar());
sto.setRef(addr, list);
```

```
valueOf(ConsExp exp) passedEnv passedSto = return list;
```

```
ListVal list = new ListVal();
ListVal templist1 = (ListVal)valueOf(((ConsExp)exp).getExp1(), passedEnv, passedSto);
ListVal templist2 = (ListVal)valueOf(((ConsExp)exp).getExp2(), passedEnv, passedSto);
list.getListVal().addAll(templist2.getListVal());
list.getListVal().addAll(templist1.getListVal());
```

---

valueOf(ConsExp exp) passedEnv passedSto = return list;

Function Expressions

```
VarExp e1 = (VarExp) ((FunctionExp) exp).getExp1();
Exp e2 = ((FunctionExp) exp).getExp2();
```

---

valueOf(FunctionExp exp) passedEnv passedSto = return new FunctionVal(e1, e2, passedEnv, passedSto)

```
FunctionVal e1 = (FunctionVal) valueOf(((CallExp) exp).getExp1(), passedEnv);
ExpVal e2 = valueOf(((CallExp) exp).getExp2(), passedEnv);
```

---

valueOf(CallExp exp1, exp2) passedEnv passedSto = applyFunction(e1, e2)

where applyFunction is defined as such:

```
VarExp FE1 = p1.getE1();
Exp FE2 = p1.getE2();
Environment env = p1.getEnv();
Store passedSto = p1.getStore();
Integer addr = passedSto.newRef(e1);
Environment newEnv = env.extendEnv(FE1.getVar(), addr);
return valueOf(FE2,newEnv, passedSto);
```

Lexical Binding Expressions

```
ExpVal e1 = valueOf(((DoExp)exp).getExp1(), passedEnv, passedSto);
Integer addr = passedSto.newRef(e1);
Environment newEnv = passedEnv.extendEnv(((DoExp)exp).getToken(), addr);
```

---

valueOf(DoExp exp1, exp2) passedEnv passedSto = valueOf((exp2) newEnv passedSto);

```
Token t = ((AssignExp)exp).getVar();
ExpVal e = valueOf(((AssignExp)exp).getExp(), passedEnv, passedSto);
Integer addr = sto.newRef(e);
env = env.extendEnv(t,addr);
```

---

valueOf(AssignExp exp) passedEnv passedSto = return (ExpVal)sto.deRef(env.applyEnv(t));

I/O Expressions

---

valueOf(PrintExp exp) passedEnv passedSto = return new StringVal(((PrintExp)exp).getExp().toString());

---

valueOf(ReadExp exp) passedEnv passedSto = return new ExpVal(((ReadExp)exp).getToken().toString());

Unique REPL Loop Expression

```
ExpVal e = valueOf((CallExp exp userInput), passedEnv, passedSto);
```

---

valueOf(LoopExp exp) passedEnv passedSto = return new StringVal(“Exiting…”);

---

valueOf(SaveExp exp) passedEnv passedSto = return new StringVal(“Function saved”);

```
ExpVal e = valueOf((CallExp exp userInput), passedEnv, passedSto);
```

---

valueOf(RunExp exp) passedEnv passedSto = return new StringVal(“Exiting…”);