Team Closure Language Proposal

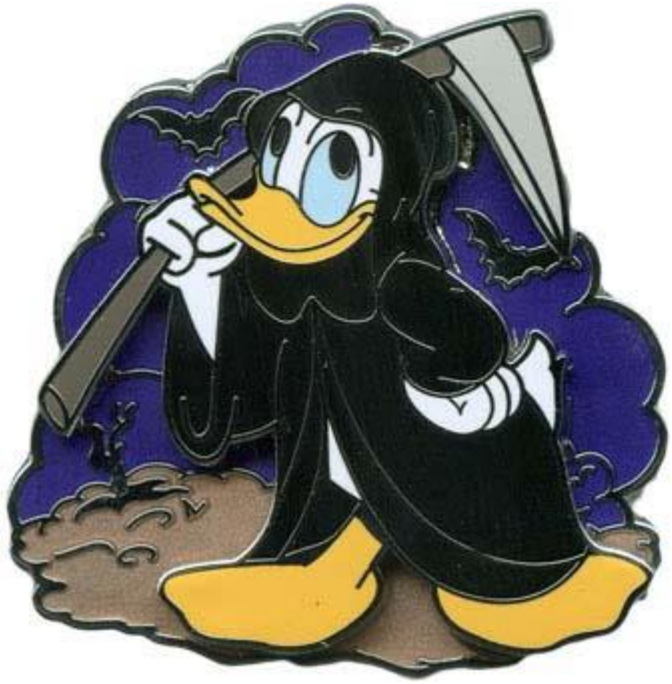
Authored By:

Edward Sutka

Mathew Mafia

Cameron Myer

The Death By Duck Language (DBD)



Picture provided from: http://www.yourwdwstore.net/assets/images/pins/2011pins/09sept/400004300487.jpg

I. Proposal

Intended Audience:

DBD is being developed with a targeted audience of duck lovers and academics alike. The intent of the language is to be lightweight and intuitive much like JavaScript or Haskell is. The framework of this langue should provide a great starting point for those interested in learning to code, or those who are generally interested in building light weight applications for personal use.

Dominate Paradigm:

The design of the Death by Duck (DBD) language is inspired by Cobalt, JavaScript, Java, Haskell, and C++. The inner workings of these languages provided the do's and don'ts for the DBD Dev

Team. The dominate paradigms observed in these languages are functional programming, object orientated programming, and procedural programing. After careful evaluation of all these languages the DBD Dev Team has chosen to implement DBD as a functional language that is dynamically typed, with an implicit free store.

Built-In Control Mechanisms:

• The Syntax of Death By Duck will provide the binding of variables using the "set" statement in both a local and global storage

• Death By Duck will implement conditional statements using the keywords "If" "then" and "else."

• Death By Duck will support user defined functions called a "func."

• Death By Duck will support user defined recursive functions called "ReFunc."

• Death By Duck will implement a "for" loops for its users.

• The Language will support an implicit lazy evaluation like call by need to evaluate expressions. This is so that code never called is never evaluated.

Data Types:

Expressed Values: IntVal, FloatVal, BoolVal, FuncVal, ListVal, StringVal

Denoted Values: Reference

Storable Values: StoVal(ExpVal)

Operators & Built-in Functions:

Integer and Float Based Operations	

sub(x , y)	x – y
add(x , y)	x + y
mul(x , y)	x * y
div(x , y)	x / y
pwr(x , y)	x ^ y
sqrt(x)	Returns the square root of a number
zero?(x)	Returns Boolean value based on if x is 0 or not
equal?(x , y)	Returns true if x and y are equal
greater?(x , y)	Returns true if x is greater then y
less?(x , y)	Returns true if x is less then y
greaterEqual?(x , y)	Returns true if x is equal to or greater then y
lessEqual?(x , y)	Returns true if x is equal to or less then y
List Based Operations	
emptyList()	Builds empty list
addListElem(x , y)	Adds element x to list y
getTail(y)	Gets the tail of elements in list y
getHead(y)	Gets first element in a list
getElemAt(x , y)	Gets element in list y at index x
null?(y)	Returns Boolean value based on if y is null or not
String Based Operations	
concat(x , y)	Combines string y into x

charAt(x , y)	Finds a character in string x at index y

Type System:

Death By Duck is a dynamically typed language with a type checker to stop errors at run time but do not inhibit the programmer from altering their data types at different times in their code.

1. All Integer and Float based operations may only receive IntVals or FloatVals as parameters for either operators or operands.

2. emptyList receives no parameters.

3. The getLast, getHead, and Null? List operations all receive one parameter that must be a ListVal.

4. The getListElem, accepts a ListVal as its first parameter and a IntVal as its second value.

5. The addListElem accepts a variable expression as its first value and only a ListVal as its second value.

6. Concat accepts only StringVals for both parameters.

7. charAt accepts only StringVals for its first parameter and only an IntVal as its second parameter.

8. If statements must have an operation that evaluates to a BoolVal in the *if* portion of the statement. The *then* and else *portions* should have equal types.

9. For loop statement accepts only a set expression for its first parameter, an expression that evaluates to a BoolVal for its second parameter, and an expression that evaluates to a IntVal or FloatVal for its final expression.

Examples:

Input	Result
777	777
777.777	777.777
Set x = 7	X = 7
Pwr(7 , 7)	49
Set x = 7 Return if equal(x , 7) Then 7 Else -7	7
getHead(addListElem(7, addListElem(6, addListElem(5, emptyList()))))	7
getLast(addListElem(7, addListElem(6, addListElem(5, emptyList()))))	5
getElemAt(2 ,addListElem(7, addListElem(6, addListElem(5, emptyList()))))	6
null(‘hello’)	Fails: type mismatch
isZero?(‘hello’)	Fails: type mismatch
Add(5, false)	Fails: type mismatch

II. Formal Syntax

Overview:

DBD source programs use ASCII character encoding. *Italics* denote a non-terminal symbol; **Bold** denotes a terminal symbol. The Kleene Star and Kleene Closure are used for repetition, and the pipe | is used for alternatives.

Lexical Specification:

In the context-free grammar below, the symbol *Space* is understood to be whitespace

Digit ::= [0-9]
Int ::= Digit$^+$
LowercaseAlpha ::= [a - z]
UppercaseAlpha ::= [A – Z]
Alpha ::= [*LowercaseAlpha* | *UppercaseAlpha*]$^+$
Special ::= , | (|) | .
Keyword ::= if | else | true | false | set | equal? | zero? | null? | greater?| less? | greaterEqual? | lessEqual? | sub | add | mul| div | pwr | sqrt | emptyList | addListElem | getLast | getHead | concat | charAt

Grammatical Specification:

In the following context-free grammar, suggested AST names accompany each production

Program* ::= *Expression	
***Expression* ::=**	
::= equal? (*Expression*, *Expression*)	assertEqual?(expr, expr)
::= set *Identifier* = *Expression* return *Expression*	set id = expr
::= if *Expression* then *Expression* else *Expression*	if boolVal then Expr else Expr
::= zero? (*Expression*)	isZero?(num)
::= null? (*ListExpression*)	isNull?(ListHead)

::= sub (***Expression, Expression*** **)**	sub(num, num)
::= add (***Expression, Expression*** **)**	add(num, num)
::= mul (***Expression, Expression*** **)**	multiply(num, num)
::= div (***Expression, Expression*** **)**	divide(num, num)
::= pwr (***Expression, Expression*** **)**	power(num, num)
::= emptyList	makeEmptyList
::= addListElem (***Expression, Expression*****)**	add(expr, listVal)
::= getLast(***Expression*** **)**	getLast(listVal)
::= getHead(***Expression*** **)**	getHead (listVal)
::= getElemAt (***Expression, Expression*****)**	getElementAt (intExp, listVal)
ListOfExp ::=	
::= ListEnd(Expression, Expression)	ListEnd Var, Exp)
::= ListPce(Expression, Expression, Expression)	ListPce (Var, Exp, ListExp)

III. Formal Semantics

Operational Semantics

Host-Specific / Haskell

The inference rules below specify the behavior of the LET language in terms of the Haskell language as a host platform for implementation.

Constant Expressions

value_of (ConstExp *Int*) *env* = (IntVal *Int*)

value_of (ConstExp *Float*) env = (FloatVal *Float*)

value_of (NegativeExp *Int*) env = (IntVal -*Int*)

value_of EmptyListExp env = ListVal [ ]

Variable Expressions

value_of (VarExp *var*) *env* = apply_env *env var*

Arithmetic Expressions

$$\frac{\text{value\_of } exp_1\ env = \text{IntVal } num_1 \quad \text{value\_of } exp_2\ env = \text{IntVal } num_2}{\text{value\_of (SubExp } exp_1\ exp_2 \text{) } env = \text{IntVal (} num_1 - num_2 \text{)}}$$

$$\frac{\text{value\_of } exp_1\ env = \text{IntVal } num_1 \quad \text{value\_of } exp_2\ env = \text{IntVal } num_2}{\text{value\_of (AddExp } exp_1\ exp_2 \text{) } env = \text{IntVal (} num_1 + num_2 \text{)}}$$

$$\frac{\text{value\_of } exp_1\ env = \text{IntVal } num_1 \quad \text{value\_of } exp_2\ env = \text{IntVal } num_2}{\text{value\_of (PwrExp } exp_1\ exp_2 \text{) } env = \text{IntVal (} num_1 * num_2 \text{)}}$$

$$\frac{\text{value\_of } \mathit{exp}_1 \ \mathit{env} = \text{IntVal } \mathit{num}_1 \qquad \text{value\_of } \mathit{exp}_2 \ \mathit{env} = \text{IntVal } \mathit{num}_2}{\text{value\_of (MulExp } \mathit{exp}_1 \ \mathit{exp}_2 \text{) } \mathit{env} = \text{IntVal (} \mathit{num}_1 * \mathit{num}_2 \text{)}}$$

$$\frac{\text{value\_of } \mathit{exp}_1 \ \mathit{env} = \text{IntVal } \mathit{num}_1 \qquad \text{value\_of } \mathit{exp}_2 \ \mathit{env} = \text{IntVal } \mathit{num}_2}{\text{value\_of (DivExp } \mathit{exp}_1 \ \mathit{exp}_2 \text{) } \mathit{env} = \text{IntVal (} \mathit{num}_1 \ / \ \mathit{num}_2 \text{)}}$$

Predicate Expressions

$$\frac{\text{value\_of } \mathit{exp}_1 \ \mathit{env} = \text{IntVal } \mathit{num}}{\text{value\_of (IsZeroExp } \mathit{exp}_1\text{) } \mathit{env} = \text{BoolVal (} \mathit{exp}_1 == \mathit{0} \text{)}}$$

$$\frac{\text{value\_of } \mathit{exp}_1 \ \mathit{env} = \text{IntVal } \mathit{num}_1 \qquad \text{value\_of } \mathit{exp}_2 = \text{IntVal } \mathit{num}_2}{\text{value\_of (IsEqualExp } \mathit{exp}_1 \ \mathit{exp}_2\text{) } \mathit{env} = \text{BoolVal (} \mathit{exp}_1 == \mathit{exp}_2 \text{)}}$$

$$\frac{\text{value\_of } \mathit{exp}_1 \ \mathit{env} = \text{IntVal } \mathit{num}_1 \qquad \text{value\_of } \mathit{exp}_2 = \text{IntVal } \mathit{num}_2}{\text{value\_of (IsGreaterExp } \mathit{exp}_1 \ \mathit{exp}_2\text{) } \mathit{env} = \text{BoolVal (} \mathit{exp}_1 > \mathit{exp}_2 \text{)}}$$

$$\frac{\text{value\_of } \mathit{exp}_1 \ \mathit{env} = \text{IntVal } \mathit{num}_1 \qquad \text{value\_of } \mathit{exp}_2 = \text{IntVal } \mathit{num}_2}{\text{value\_of (IsLessExp } \mathit{exp}_1 \ \mathit{exp}_2\text{) } \mathit{env} = \text{BoolVal (} \mathit{exp}_1 < \mathit{exp}_2 \text{)}}$$

$$\frac{\text{value\_of } exp_1\ env = \text{IntVal } num_1 \quad \text{value\_of } exp_2 = \text{IntVal } num_2}{\text{value\_of (IsGreaterEqualExp } exp_1\ exp_2)\ env = \text{BoolVal (} exp_1 >= exp_2 \text{)}}$$

$$\frac{\text{value\_of } exp_1\ env = \text{IntVal } num_1 \quad \text{value\_of } exp_2 = \text{IntVal } num_2}{\text{value\_of (IsLessEqualExp } exp_1\ exp_2)\ env = \text{BoolVal (} exp_1 <= exp_2 \text{)}}$$

$$\frac{\text{value\_of } exp_1\ env = \text{ListVal } n}{\text{value\_of (IsNullExp } exp_1)\ env = \text{BoolVal (} null\ n == True\text{)}}$$

Conditional Expressions

$$\frac{\text{value\_of } exp_1\ env = \text{BoolVal } test}{\begin{array}{l}\text{value\_of (IfExp } exp_1\ exp_2\ exp_3 \text{) } env \\ \quad \mid test == \text{True} = \text{value\_of } exp_2\ env \\ \quad \mid \text{otherwise} \quad = \text{value\_of } exp_3\ env\end{array}}$$

Lexical Binding Expressions

$$\frac{\text{value\_of } exp_1\ env = val \qquad \text{value\_of } exp_2 = \text{(ListVal } vals\text{)}}{\text{value\_of (AddListElemExp } exp_1\ exp_2)\ env = \text{ListVal (} val : vals\text{)}}$$

$$\frac{\text{value\_of } exp_1\ env = \text{(ListVal } l_1\text{)}}{\text{value\_of (GetTailExp } exp_1 \text{) } env = \text{ListVal (tail } l_1 \text{)}}$$

$$\frac{\text{value\_of } exp_1 \ env = (\text{ListVal } vals)}{\text{value\_of } (\text{GetHeadExp } exp_1) \ env = \text{head } vals}$$