**Languages Requirements Flexible**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date Data Type** | | | |
| Date birthDt; | Represented as a String of 10 characters.  Note that Java's built-in date validation functions will not find these problems | Valid:  "2017-01-31"  "2016-02-29"  Invalid:  "17-01-31"  "2017-1-31"  "2017-02-29"  "2017-02-288" | Dates work as character strings for string operations.  I will provide some C functions that you might like. |
| Date Functions | **dateDiff**(date1, date2) returns an Int representing the difference in days | difference in days (date1 - date2)  dateDiff("2017-03-01", "2016-03-01")  returns 365  dateDiff("2017-02-01", "2016-02-01")  returns 366 | if using prefix:  (dateDiff date1 date2) |
|  | **dateAdj**(date, days)  returns a date | dateAdj("2017-03-01", -365)  returns "2016-03-01"  dateAdj("2017-03-01", 365)  returns "2018-03-01" | If using prefix:  (dateAdj date days) |
|  | **dateAge**(date1, date2)  returns an Int representing the years between the dates | difference in years (date1 - date2)  dateAge("2017-03-01", "2016-03-01")  returns 1  dateAge("2017-02-01", "2016-02-01")  returns 1  dateAge("2017-02-01", "1957-12-04")  returns 59 | if using prefix:  (dateAge date1 date2) |
| **Infix vs Prefix** | | | |
| Infix expressions are an option in the flexible requirements. | **We also must recognize unary minus.** | Infix:  tax = amount \* 1.15 + prime\*0.02;  result = 5 + func(-x+y, z-7) \* 5;  Prefix:  tax = (+ (\* amount 1.15)  (\* prime 0.02));  result = (+ (\*(func (+(- x) y)  (- z 7) ) 5)); |  |
| **Initializations** | | | |
| *datatype varNm =*  *value;*  *datatype varNm*  [*numElem*] =  *valueList*; |  | Int i=5;  String name = "I. M. Code";  Float gradeM [20] = 90.5, 50.0  , 60.0, 85.5;  String suits[4] = "Hearts"  , "Clubs", "Diamonds", "Spades"; |  |
| **Unbounded Arrays** | | | |
| *datatype varNm*  [] =  *valueList*; | **Size of the array can grow.** | Float gradeM [] = 90.5, 50.0  , 60.0, 85.5;  gradeM[30] = 100.0; | This also impacts array to array assignments. |
| **Additional Numeric Assignment Operators** | | | |
| +=  -= |  | x += 10;  x += 25.5;  y -= 1; |  |
| **IN and NOTIN operators** | | | |
| *value* **IN** *valueList* | Returns T of the value is in the valueList.  The data type is based on the value to the left of the operand. | fruit IN "apple", "orange", "clark"  gradePt IN 4, 3, 2, 1, 0 | if using prefix:  (IN value valueList)  (IN fruit "apple" "orange" "maynard") |
| *value* **NOTIN** *valueList* | Returns T of the value is not in the valueList.  The data type is based on the value to the left of the operand. | fruit NOTIN "apple", "orange", "clark"  gradePt NOTIN 4, 3, 2, 1, 0 | if using prefix:  (NOTIN value valueList)  (NOTIN fruit "apple" "orange" "maynard") |
| **slices** | | | |
| string[lb:ub] in expressions | returns the string that begins with the string[lb] and ends with the character before the ub. | Example:  str = "goodbye";  str[0:4] returns "good"  str[:4] returns "good"  str[4:] returns "bye" | This also affects finding a colon marking the end of an if or while condition.  You must checks for out of bounds.  You do not have to support negative values for the lb or ub. |
| array[lb:ub] in expressions | returns an array that begins with the element at array[lb] and ends with the element before array[ub] |  | This also affects finding a colon marking the end of an if or while condition.  You must checks for out of bounds.  You do not have to support negative values for the lb or ub. |
| Assigning slices of arrays and strings | Assigning to a slice can cause the target string or array value to grow or shrink. | str = "tacobell";  str[:4] = "school";  print(str); // prints schoolbell  str[0:6] = ""; // changes it from  // schoolbell to bell |  |
| **tokenizing a string** | | | |
| for *stringCV* from *string* by *delimiter*: | If not defining, *stringCV* implicitly is a string type. It is assigned each piece of the specified string based on the delimiter. | str = "apple,orange,clark"  for fruit from str by ",":  print (fruit);  endfor; |  |
| **break and continue** | | | |
| break | exit immediately surrounding for or while |  | Since our select statement doesn't use break, it doesn't impact a break. |
| continue | branch to the bottom of the for or while and continue the loop |  |  |
| **select when default endselect** | | | |
| select *value*:  when *valueList1*:  *statements1*;  when *valueList2*:  *statements2*;  …  default:  statementsN;  endselect; | The value can be an Int or a String (dates are also allowed).  Note that we can have nested select statements. | select grade:  when "A", "B":  print ("doing well");  when "C":  print ("average");  when "D":  print ("warning");  when "F":  print ("oh noooo");  default:  print("unknown grade:"  , grade);  endselect; | Compiler: do not use a case statement since C doesn't support strings. Also, it would impact the Havabol break. |
| **programmer-defined functions** | | | |
| def *type funcName*  (*parameterList*):  *statements*;  return *value*;  enddef; | Define a programmer-defined function. | def Void minMax (Ref Float[] fArrayM  , Ref Float fMax, Ref Float fMin):  fMax = 0.0; // arbitrary low value  fMin = 1000.0; // arbitrary high value  for fItem in fArrayM:  if fItem > fMax:  fMax = fItem;  endif;  if fItem < fMin:  fMin = fItem;  endif;  endfor;  enddef; | Return data types can be Void, String, Int, Bool, Float or Date (if supported).  Functions must support at least one parameter passing approach (by value or by ref). Additional points for doing both.  Must be able to pass primitive, strings, and arrays. If supporting slices, must be able to pass slices. Note that slices and expressions are copied to a temporary location that can be passed when passing by Ref. |
| Function definitions can be nested. | Non-local references use static scope. |  |  |
|  |  |  |  |
|  |  |  |  |