Connor Valerio & Ekrem Yurdakul

Programming Language Concepts - User Manual

---

1. Keywords

In order to understand the syntax of our language, you must first be informed of the keywords we have adopted. Listed below are our keywords (in bold) with their uses, as well as this, some examples have been outlined (in italic).

IN: This keyword is used to pass the previous environment forward to the next line of the program; this includes any external file input, variable assignments or function definitions.

VAR: This keyword is used when declaring a new variable. Note that when declaring a new variable, an identifier must be given to that variable; this identifier should start with a dollar sign ($) as do all identifiers in the language. Following the identifier should be an equals sign (=), the value being assigned to the variable, the 'IN' keyword and then the rest of the program.

```
VAR $myvar = 5 IN … (rest of program)
```

DEFINE: This keyword is used to define a new function within the program. Similarly to the variable, the function being defined will also need an identifier. One parameter is also expected for each function that is defined. The parameter should be surrounded with parenthesis and should be followed by an equals sign (=), the function body, the 'IN' keyword' and the rest of the program.

```
DEFINE $myfunction($parameter) =
        $parameter + 1 IN
$myfunction(9)
```

The above function definition defines a function that adds 1 to any parameter given to it, in this case it is then being applied to the number '9' and will therefore return '10'.

IF THEN ELSE: These keywords are used to define a conditional "if statement".

```
1)  VAR $myvar = 5 IN
    IF ($myvar = 5) THEN true ELSE false
```

```
2)  VAR $myvar = 5 IN
    IF ($myvar > 10) THEN true ELSE false
```

The first program will return true because 'myvar' is equal to 5 and the 'if statement' is checking to see if this condition is true. The second 'if statement' will return false, this is because 'myvar' is not greater than 10 (i.e. 5 is not greater than 10)

---

2. Basic Syntax

Types – This language is not strongly typed and therefore you do not need to declare variable types, however, the language itself will recognise the following types: Boolean, Int, String & CurlyList. Boolean types can be used through the keywords 'true' and 'false', Int types are recognised as whole numbers and String types are recognised as a sequence of lowercase alphabetical letters. Lastly, the CurlyList type is similar to a list you may see in other languages, however, its properties differ slightly. The CurlyList type can only hold String elements, it surrounds its elements with curly brackets and separates its elements with commas (e.g. {e, x, a, m, p, l, e}).

Function Names – As previously mentioned, these must begin with a dollar sign ($) and can contain lower case alphabetical characters, numbers or a mixture of both.

Variable Names – Variable names, much the same as function names, must also begin with a dollar sign ($) and should be made up of lower case alphabetical characters, numbers or both.

Case Sensitivity – The language we have developed is case sensitive and therefore when referring to built-in functions or keywords, it is important to make sure they are written in upper case alphabetical characters (with the exception of 'true' and 'false' which are lower case reserved keywords).

Whitespace Sensitivity - Our language is whitespace insensitive; all spaces, newline characters and tabs will be ignored when the program is run.

Mathematical Operators – The language includes the use of addition, subtraction, multiplication and division (+ - * /). As well as this, the language accepts the use of equals, greater than and less than symbols (= > <). It is worth noting, the language only accepts whole numbers and if a decimal number is produced by division it will automatically be rounded down (i.e. 6 / 5 = 1).

Getting Input from a File – In order to get input from a command line file, the function *READFILE* must be called at the beginning of the program; this prepares the program's environment for file input. Once *READFILE* has been called, lines within the file can be read into the program using: *CLISTREADIN, INTREADIN & STRINGREADIN*. Each of the three aforementioned functions will need to be provided with a line number in order to successfully read in the contents from the file. These contents can then be stored as variables, as shown in the box below.

CLISTREADIN – used when the line being read in from a file is of type *CurlyList*.

INTREADIN – used when the line being read in from a file is of type *Int*.

STRINGREADIN – used when the line being read in from a file is of type *String.*

```
READFILE IN
VAR $myclist = (CLISTREADIN 1) IN
VAR $myint = (INTREADIN 2) IN
VAR $mystring = (STRINGREADIN 3) IN
…
```

---

3. Library of Built-in Functions

Below are two tables of built-in functions that were added to the language for a programmer's convenience. Figure 1 shows a table of functions that are used on Lists, whereas Figure 2 is a table of general functions (although predominantly contains functions used for file reading).

Figure 1 - List Functions

Name of Function	Parameters (Name : Type)	Description of Function	Example of Use
FILTER	• list : CurlyList • limiter : Integer	This function filters the output to contain the first 'x' elements of the supplied list, where 'x' is the value of the limiter.	*"FILTER {a, b, c, d, e} 3"* Result: **{a, b, c}**
APPEND	• element : String • list : CurlyList	Appends an element (string of alphabetical characters) to the end of a list.	*"APPEND e {a, b, c, d}"* Result: **{a, b, c, d, e}**
GET	• list : CurlyList • index : Integer	Returns the element (string) at a specific index of the list. The first element in the list is at index '1'.	*"GET {a, b, c} 2"* Result: **b**
LENGTH	• list : CurlyList	Returns the number of elements in a given list.	*"LENGTH {a, b}"* Result: **2**
SORT	• list : CurlyList	Returns a list where the elements are sorted into lexicographical order.	*"SORT {e, d, c, b, a}"* Result: **{a, b, c, d, e}**
SETIFY	• list : CurlyList	Returns a list where all of the duplicated elements have been removed (i.e. makes the list into a set).	*"SETIFY {a, b, b, a, c, c}"* Result: **{a, b, c}**

JOIN	• list1 : CurlyList • list2 : CurlyList	Returns a list made up of the elements found in 'list1' and 'list2'. Due to the nature of the function, the order will not be respected.	*"JOIN {a, b} {c, d}"* Result: **{d, c, a, b}**
PRINTLISTS	• list1 : CurlyList • list2 : CurlyList	Prints out 'list1' and 'list2' to standard output.	"PRINTLISTS {a, b, c} {d, e, f}"
CONTAINS	• list : CurlyList • element : String	Checks to see if the list contains a specific element. If it does it will return true, else it will return false.	*"CONTAINS {a, b} b"* Result: **true** *"CONTAINS {a, b} c"* Result: **false**
CAR	• list : CurlyList	Returns the first element of the list.	*"CAR {abc, def, ghi}"* Result: **abc**
CDR	• list : CurlyList	Returns a list containing every element in the original list, except the first element.	*"CDR {abc, def, ghi}"* Result: **{def, ghi}**

Figure 2 - General Functions

Name of Function	Parameters (Name : Type)	Description of Function	Example of Use
CONCAT	• string1 : String • string2 : String	Concatenates 'string1' and 'string2' and returns the new String.	*"CONCAT hello world"* Result: **helloworld**
READFILE	N/A	Prepares a program for file input from the command line. It should be called at the very start of the program and should be proceeded by the keyword 'IN' if file input is going to be used.	*READFILE IN* ---------- *Rest of program* ----------
INTREADIN	• linenumber : Int	Reads the line of the input file specified by the 'linenumber' parameter.	*"INTREADIN 1"*
STRINGREADIN	• linenumber : Int	Reads the line of the input file specified by the 'linenumber' parameter.	*"STRINGREADIN 2"*
CLISTREADIN	• linenumber : Int	Reads the line of the input file specified by the 'linenumber' parameter.	*"CLISTREADIN 3"*

4. Type Checking & Informative Error Messages

The language checks types for conditional if statements, mathematical operations and all built in functions that take arguments. Due to the nature of the language, the main exception that is thrown is "InvalidTypeError"; with this an informative error message is displayed to the user. Below are a list of error messages that can be displayed to the user and examples of when they occur.

*Note all error messages displayed in the table are of type "InvalidTypeError" exception.

Error Message	Example of when error message is displayed
"Int Parse Error (Single): Unable to parse the Int supplied."	Providing *GET* with a string value for the index: *"GET {a, b, c} d"*
"Mathematical Operation: Requires two Integer types"	Trying to add a string to an integer: *"a + 1"*
"CONCAT: Two Strings expected for this function."	Trying to *CONCAT* a string with a non-string: *"CONCAT b 1"*
"String Parse Error (Single): Unable to parse the Strings supplied."	Trying to *APPEND* a number to a CurlyList: *"APPEND 1 {a, b, c}"*
"CList Parse Error: Unable to parse the CList supplied."	No example with the language's built-in functions – would be displayed if the CurlyList supplied couldn't be parsed for some reason.
"IF STATEMENT: Conditional Statement is not of type Boolean."	Providing the IF statement with a non-conditional expression for its first expression: *"IF (1+1) THEN true ELSE false"*

There is one additional Exception thrown by the underlying OCaml code, this is the "IndexOutOfBounds" Exception we have implemented. It is thrown, for example, when trying to access an element in a CurlyList that isn't accessible, for example: *"GET {a, b, c, d, e} 0"* **or** *"GET {a, b, c, d, e} 6"*.