Learning Language

Reference Manual

George Liao (gkl2104)
Joseanibal Colon Ramos (jc2373)
Stephen Robinson (sar2120)
Huabiao Xu(hx2104)

A. Introduction

Learning Language is a programming language designed to be accessible to students prior to entering high school. Students at this age traditionally have no exposure to computer programming, despite the fact that many of them have the maturity and education to write procedural logic. For these students, existing programming languages can be overwhelming due to their abstract syntax and the difficulty of implementing I/O operations. To that end, the language is designed with the needs and abilities of 10-14 year old children in mind.

B. Lexical Conventions

B1. Comments

The character # introduces a comment, which terminates at the next newline. Comments do not occur within string literals.

B2. Identifiers

An identifier is a sequence of letters and digits and underscores. The first character must be a letter. Upper and lower case letters are the same. Identifiers may have any length.

B3. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

string	return	called
number	function	listof
fraction	until	display
repeat	displayline	end
If	ifnot	times
then	end	open
read	readline	

B4. Constants

There are several kinds of constants.

Constant:

Number-constant
String-constant
Fraction-constant
List-constant

Number-Constant: A number constant consists of a sequence of digits which are taken to be an integer in decimal base. The sequence is optionally preceded by a '-' for negative.

String-Constant: A string constant is a sequence of one or more characters enclosed in double quotes, as in "foobar". In order to represent the " character, newlines, and certain other characters, the following escape sequences may be used:

Newline \n
Backslash\\
Double quote \"

Fraction-Constant: A fraction constant is two numbers separated by '//'.

List-Constant: A list constant is a comma separated sequence of number, fraction or string constants surrounded by braces. Example: {1,2,7}

C. Basic Types

There are three basic types of variables in Learning Language. They are number, fraction and string. The number type is a 32bit integer. The fraction type is two 32 bit integers, a numerator and a denominator. The string type is a dynamically allocated array of characters. The programmer does not need to specify the size of the string.

There is one derived type in Learning Language. It is the list. Lists can be of one of the three basic variable types or of another list. Memory is dynamically allocated to the list as needed. Users do not need to specify the size of their list.

D. Conversions

There is only one case where conversions occur in Learning Language. For binary arithmetic operators, if one operand is a number and the other a fraction, then the number is promoted to a fraction.

E. Expressions

E.1 Primary Expressions

Primary expressions are identifiers, constants, strings, or expressions in parenthesis.

Primary-expression:
Identifier
Constant
String
(expression)

E.2 Postfix Operators

The operators in postfix expressions group left to right.

Postfix-expression:
Primary-expression
Postfix-expression[expression]
Postfix-expression(argument-expression-list)
Postfix-expression()

Argument-expression-list:
Assignment-expression
Argument-expression-list, assignment-expression

List references: A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted list reference. The first of the two expressions must have type "Listof T" where T is some type, and the other must have Number type.

Function Calls: A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function. In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. Functions must be declared with explicit parameter types.

E.3 Unary Operators

Unary Minus Operator '-':

The operand of the unary minus operator must have type number or fraction. The result is the negative of the operand.
Example:

```
Number called b
b <- -4
b <- -b
```

Logical Negation Operator 'not':

The operand must be of type number. The result is 1 if its operand is equal to 0, and 0 otherwise. The result is of type number.
Example:

```
Number called b
b <- 1
a <- not b      #a = 0
```

Logical Magnitude Operator '|operand|':

The operand must be of type string or list. The result is the length of the string or the number of elements in the list. The result is of type number.
Example:

```
Number called a
string called b
listof number called c
b <- "the cat"
a <- |b|      #a = 7
c[5] <- 3
a <- |c|     #a = 5
```

Absolute Value Operator '|operand|':

The operand must be of type number or fraction. The result is the length of the string. The result is of type number.
Example:

```
Number called a
number called b
a <- -8
b <- |a|      #b = 8
```

E.4 Binary Arithmetic Operators

Exponential Operator '^':

The exponential operator groups left to right. The exponent must be of type number and positive. The left operand may be a fraction or a number.

Exponential-expression:
Exponential-expression ^ unary-expression
Unary-expression

Example:

```
Number called a
number called b
a <- 2
b <- 2 ^ a      #b = 4
```

Multiplicative Operators '*', '/', '%':

The multiplicative operators group left to right. The operands for '*' (multiplication) and '/' (division) must be numbers or fractions. The operands for '%' (remainder) must be numbers. If the second operand is zero for any of the above, the result is 0. If one operand is a fraction, the other is implicitly converted to a fraction

Multiplicative-expression:
*Multiplicative-expression * Exponential-expression*
Multiplicative-expression / Exponential-expression
Multiplicative-expression % Exponential-expression
Exponential-expression

Example:

```
Number called a
number called b
a <- 2 * 4 / 3  #a = 2
b <- a % 2      #b = 0
```

Additive Operators '+', '-' :

The additive operators group left to right. The operands for '+' (addition) and '-' (subtraction) must be numbers or fractions. If one operand is a fraction, the other is implicitly converted to a fraction

Additive-expression:
Additive-expression + Multiplicative-expression
Additive-expression - Multiplicative-expression
Multiplicative-expression

Example:

```
Number called a
a <- 2 + 4 - 3  #a = 3
```

E.5 String Operators

Concatenation Operator '+':

The '+' operator can be used to concatenate two strings. It groups left to right.

string-expression:
*string-expression * string*
string-expression / string
string-expression % string
string

Example:

```
string called a
a <- "cat"
a <- a + a   #a = "catcat"
```

E.6 Relational Operators

The relational operators are '<' (less than), '>' (greater than), '<=' (less than or equal), '>=' (greater than or equal), '=' (equals) and 'not=' (not equal to). These operators all yield 1 if the expression evaluates to true and 0 if it evaluates to false. When multiple relational operators are used in a single expression, the enclosed operands are duplicated so that the relational expression is evaluated correctly.

For example, the expression:

```
A < b < c
```

is parsed as:

```
(A < b) * (b < c)
```

Relational-expression:
Additive-expression < Additive-expression
Additive-expression > Additive-expression

Additive-expression <= Additive-expression
Additive-expression >= Additive-expression
Additive-expression = Additive-expression
Additive-expression not= Additive-expression
Additive-expression

Compound-Relational-expression:
cre < Relational-expression
cre > Relational-expression
cre <= Relational-expression
cre >= Relational-expression
cre = Relational-expression
cre not= Relational-expression

cre:
cre < Additive-expression
cre > Additive-expression
cre <= Additive-expression
cre >= Additive-expression
cre = Additive-expression
cre not= Additive-expression
Additive-expression

E.7 Assignment Operator

The assignment operator, '<-', groups left to right. The operands must be of the same type with the exception of an integer being assigned to a fraction.

Assignment-expression:
primary-expression <- Assignment-expression
primary-expression <- sub-assignment-expression

sub-assignment-expression:
Relational-expression
Compound-Relational-expression

Example:

```
Number called a
a <- 2    #a = 2
```

F. Declarations

Declarations are used to add identifiers to the name space. A declaration also specifies the data type of an identifier.

F1. Variable Declarations

Learning language supports the following types: number, fraction, string. Furthermore, lists of any of the above types may be declared. Memory is dynamically allocated for the list. The user does not specify the size of the list. Declarations have the form:

Declaration:
Type called identifier
Listof Declaration

Type:
Number
Fraction
String
Listof Type

Learning Language does not support user defined types.

F2. Function Declarations

Functions cannot be declared in the main code file. Functions are defined in a separate file (typically *.lf) and can be included to be used in the main code. Functions should be included at the top of the main file but so long as they are included prior to being called, the program will compile correctly. The syntax for including a function file is:
`use filename`
where `filename` is the relative path to the file in double quotes (example: “func.lf”).

Function declarations have the form:

Function-declaration:
Function called identifier(Arg-list) -> return-type

Arg-list:
Arg-list, Declaration
Declaration

return-type:
Number
Fraction
String
Listof Type
Nothing

G. Statements

G1. Statements

statement:
expression-statement
compound-statement
selection-statement
iteration-statement

G2. Expression Statements

expression-statement:
expression
ε

G3. Compound Statements

A compound statement is a block of code. Examples of compound statements are the bodies of functions, conditionals and loops.

compound-statement:
declaration-list statement-list
declaration-list
statement-list

ε

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

G4. Selection Statements

Selection statements choose one of (up to) two flows of control. The expression in the if statement must have integer type and if it evaluates unequal to zero, then the first statement is executed. The second statement, if present, executes if the expression evaluates to zero.

selection-statement:
if *expression* then *statement* end
if *expression* then *statement* ifnot *statement* end

G5. Iteration Statements

iteration-statement:
repeat *expression* times *statement* end
repeat until (*expression) statement* end

In the first form of the loop, the expression is evaluated once and it must evaluate to a number. If that number is less than zero, then the loop will not execute. The statement is executed that number of times. The iteration count is held implicitly in the variable named nth, which is scoped within the loop. For nested loops, nth holds the iteration counter for the innermost loop.

In the second form of the loop, the expression must evaluate to a number, as in the first. If that expression evaluates unequal to zero, the statement executes and then the expression is again executed. The loop will continue until the expression evaluates to zero.

H. Input and Output

H1. Standard I/O

Standard output takes on the following form:

Output:
Display expression
Displayline expression

In both forms, the evaluation of expression is output to standard out. In the latter form, the newline character is appended to the output.

Standard input takes on the following form:

Input:
Identifier <- input

When types agree, the value input by the user is assigned to the identifier. For a string, the entire input is assigned. For a number, the first integer delimited by spaces is assigned. For fractions, input must be of the form "a // b" delimited by spaces.

H2. File I/O

To read or write a file, the file must first be opened:

```
Open "file.txt"
```

To read from the file:

```
someStr <- read "file.txt"
somestr <- readline "file.txt"
```

The former reads space delimited words and the latter reads until the next new line.

To write to a file:

```
Write string-expression
```

The above always appends to the file. When file I/O is complete, the file must be closed:

```
Close "file.txt"
```

H3. Default GUI

At this time, no customization is available for the default GUI. The default GUI will only provide a window for the program. As development progresses, we will review the feasibility of useful customization such as adding dropdown menus and buttons. The addition of such functionality is dependent on the successful implementation of our programming logic, as it is essential to our language whereas the GUI is primarily aesthetic.

I. External Declarations

I.1 Function Definitions

function-definition:
function called identifier (arg-list) -> return-type statement end

Return types differ from types by the inclusion of the keyword 'nothing'. While this option is available for functions that do not return a value, such functions are not very useful. All parameter passing in Learning Language is done by value and therefore the use of such functions is very limited.

I.2 External Declarations

In Learning Language, all functions are declared externally. To incorporate them into a program, they must be included in the main code file (typically *.ll). The syntax to include a function file is:
`use Filename`
where `Filename` is the relative path to the function file.

J. Scope and Linkage

Learning language does not support precompiled routines. All functions will be recompiled when the main code is compiled. For this reason, learning language is not very scalable.

Learning Language is statically scoped. All variables declared in functions are scoped only in that function. The same is true of variables declared inside conditional and iterative statements. Variables declared above a statement are available within the statement. Variable names may not be reused within the same scope.