NEWBIE LANGUAGE REFERENCE MANUAL

Braxton Gunter *(beg2119)* - Tester
Clyde Bazile (*cb3150*) - Language Guru
John Anukem (*jea2161)* - Systems Architect
Sebastien Siclait (*srs2232*) - Tester
Terence Jacobs (*tj2316*) - Project Manager

COMS 4115

Table of Contents

1. Introduction

Traditional high-level programming languages are often too cryptic and difficult for new users to understand. The goal with Newbie is to create a pseudo-code like programming language aimed to simplify the programming experience for beginner developers. This will allow new coders the ability to design, implement and better understand common algorithms without the frustration of learning specific programming syntax. Our standard library will specifically allow for easy implementation of basic algorithms involving linked lists, graphs, and trees.

2. Lexical Conventions

This will describe how the lexical analyzer breaks a file into tokens.

2.1 Character Set

Newbie uses the 7-bit ASCII character set. If an 8-bit character set is recognized Newbie will throw an error.

2.2 Line Terminators

2.2.1 Physical Lines

Programs are divided into lines by recognizing line terminators. Line terminators are any of the standard platform line terminations:

- Unix form, ASCII LF (newline)
- Macintosh form, ASCII CR (return)
- Windows form, ASCII CR followed by the ASCII LF

The two characters CR immediately followed by LF are counted as one line terminator and all three terminator sequences can be used interchangeably.

2.2.2 Logical Lines

The end of a logical is represented by the NEWLINE token. Statements cannot cross logical line boundaries except for when using specified explicit line joining rules.

2.2.3 Explicit Line Joining

Two or more physical lines may be joined into logical lines using a single backslash (\) character per line. The backslash must not be in a string literal or comment. Blank lines, lines without whitespace or a comment terminates multi-line statements.

2.2.4 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which determines the the grouping of statements.

Tabs are replaced by four spaces and the total number of characters up to and including the replacement characters must be a multiple of four even if a mixture of tabs and spaces are used. The total number of spaces preceding the first non-blank character determines the level of indentation. Indentation cannot be split over multiple physical lines. The whitespace up to the first backslash determines the indentation.

The indentation level of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack as follows:

> Zero is pushed to the stack before any line is read and will not be popped off. The numbers pushed onto the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, the indentation level is pushed to the top of the stack and one INDENT token is generated. If it is smaller, the indentation level must be one of the numbers occurring on the stack. All larger indentation levels are popped off and a DEDENT token is generated for each. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

***Correctly formatted example, *=space, <tab>=tab*

```
define foo with params bar
* * * * if len(bar) equals 1
* * * * * * * * return bar
<tab>set new_bar to new [ ]
* * * * for i from 0 to len(bar)
* * * * <tab> for j from 0 to len(bar)
* * * * * * * * * * * * new_bar = new_bar + bar[ i : j ]
* * * * * * * * <tab>new_bar = new_bar + bar[ j : ]
* * * * * * * * new_bar = new_bar + bar[ i : ]
<tab>return new_bar
```

2.3. Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Spaces, tabs, and newlines can be used interchangeably to separate tokens. Some whitespace is always required to separate tokens.

2.4. Comments

Single line comments will be signified by two backslashes like //. Multiline comments will be enclosed by the following characters /* … */. Comments will not nest, and they will not occur within string or character literals. Comments are ignored by the syntax; they are not tokens.

2.5. Keywords

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

and	*If*	*While*	*Params*
or	*break*	*not*	*no*
class	*continue*	*equals*	*each*
return	*null*	*print*	*in*
true	*def*	*for*	*to*
False	*Else*	*With*	*from*
Greater	*less*	*than*	

2.6. Strings

Strings are represented by a sequence of characters surrounded by double quotes (”) and are immutable.

3. Syntax and Semantics

3.1 Statements

Newbie supports expression, declaration, control flow and loop statements.

3.1.1 Expression Statements

An expression statement is a statement which must be evaluated.

3.1.2 Declaration Statements

Variables in Newbie utilize type-inference (4.3) and are statically-typed. Must assign some variable to any given new variable.

```
set team_size to 5
```

3.1.3 Control Flow Statements

The if statement is used to execute the block of statements in the if-clause when a specified condition is met. If the specified condition is not met, the statement is skipped over until any of the condition is met. If none of the condition is met, the expressions in the else clause (when specified) will be evaluated. Keywords are in bold.

if *expr*
   statement
else if *expr*
   statement
else
   statement

3.1.4 Loop Statements
The while statement is used to execute a block of code continuously in a loop until the specified condition is no longer met. If the condition is not met upon initially reaching the while loop, the code is never executed. 'for each' can only be used on iterable structures, i.e., strings and list. Example:

while *expression*
      statement

for each *variable* **in** *list*
      statement

for *idx* **from** 1 **to** 100
      statement

4. Types

4.1. Primitive Data Types

Newbie will have 4 primitive data types: *bool, char, num, string*

4.1.1 ***bool***

bool represents a simple boolean value, either true or false. They can be declared as follows:

set *PLT_is_great* **to true**
set *Edwards_is_boring* **to false**

4.1.2 ***char***

char are single ASCII characters or an escape sequence followed by a character contained in single quotes. They can be declared as follows:
set p **to** 'P'
set l **to** 'l'
set t **to** 't'

It is a compile-time error for the character following the single character or escape sequence to be anything other than a '.

4.1.3 *num*

num represents both integers and floating point numbers. num types are 32-bits and follow IEEE 754 standard. Because there is no distinguishing factor between integers and floating point numbers, it is acceptable to declare numerics in a variety of ways:

set *plt* **to** 4115
set *edwards_age* **to** 57
set *grade* **to** 57.0

Given that *num* represents both integers and floats, boolean operations will ignore decimals as well. For example:

grade **equals** *edwards_age* // true

If at least one of the operands are floating point then the result will also be a floating point.

4.1.4 *string*

A *string* consists of a collection of characters enclosed in double quotes, such as ". . . ". It is possible to index through a statically declared string, but it is not possible to manipulate the data contained in the string. The string datatype supports all ASCII characters. To insert the " character in a string, use \" to avoid ending the string. Strings are iterable.

set *plt* **to** “COMS 4115”
set *hello* **to** “Hello world, PLT is a \”great\” class”

4.2. Lists

In newbie, we will have only one type of collection: Lists. A List is represented by a sequence of comma-separated elements enclosed in two square brackets [...]. Elements can be accessed by their positions in the list, beginning with the zero index. The List is a mutable data structure, which means that it supports functions to append, remove, or update its values. Lists can contain primitives or objects, but not a mix of both. Within a List of primitives, each element must be of the same type – for example, a List may not hold a collection of both *num* and *string* elements. Within a List of objects, all elements must be of the same type.

set L **to** [‘N’,’E’, ‘W’, ‘B’]

set L **to** L + ‘I’ // [‘N’, ‘E’, ‘W’, ‘B’, ‘I’]
set L[4] **to** ‘E’ // [‘N’, ‘E’, ‘W’, ‘B’, ‘E’]
L[7] // error

4.3. Type Inference
Newbie will contain a robust type inference system. Given an expression, it will be determine variable type at compile time. This will make it easier for users as they no longer need to declare parameter or variable types. As such, if we declare:

set coms **to** 4115
set class **to** “plt”
set class_is_great **to true**
set grade **to** ‘p’

the compiler will interpret these variables as a *num*, *string*, *bool*, and *char* respectively. This will extend to more advanced data types,such as Lists, as well. For example, in the expression:

set team **to** [“Brax”, “Clyde”, “John”, “Sebas”, “TJ”]

team will be interpreted as a list of strings. This type inference will be done using the hindley milner method with a standardized notation for common data types.

4.4 Automatic Initialization
During compile-time, we will be identify all variables and their corresponding types. These variables will be automatically initialized to predictable default values. This means that variables do not need to be explicitly declared or initialized. Primitives are automatically initialized to a default value depending on their type. Lists are automatically initialized to their empty states.

Type	Default Value
bool	False
char	null
num	0
string	null
List	[ ]

5. Operators and Expressions

5.1 Assignment

The = operator or (to) can be used to assign the value of an expression to an identifier.

```
set x = 5
set x to 5 // same as above
```

With type inference, the variable x is automatically declared without having to declare the type. Assignment is right associative, allowing for assignment chaining.

```
a = b = 10 // Set both a and b to 10
set a to b to 10 // same as above
```

5.2 Operators

5.2.1 Arithmetic Operators

The arithmetic operators consist of +, - ,*, /, ^, and %. The order of precedence from highest to lowest is the ^ exponentiation operator, the unary - followed by the binary * and / followed by the binary + and -.

5.2.2 Logical Operators

The logical operators consist of the keywords *and*, *or*, and *not*. The negation operator *not* keyword inverts true to false and vice versa. The logical operators can only be applied to boolean operands. The *and* keyword joins two boolean expressions and evaluates to true when both are true. The *or* keyword joins two boolean expressions and evaluates to true when both are true.

5.2.3 String Operators

String access is denoted by square brackets enclosing an integer in the range of the length string. It returns the String indexed by the integer.

```
set a to "Hello world!"
print a[0] // prints "H"
```

String concatenation is denoted by the binary + operator.

```
set a to "Hello"
set b to " world!"
set c to a + b // "Hello world!"
```

5.2.4 Relational Operators

Relational operators consist of >, <, >=, <=, == and != which have the same precedence. For primitive types, the equality comparison compares by value. == compare structurally while the is keyword compares physically. The == and != operators are valid for primitives and lists containing primitives.

```
set a = 1
set b = 1
set print a == b // True
set print a < b // False
set print a >= b //True
```

The above could be written as:

```
set a to 1
set b to 1
print a equals b
print a less than b
print a greater than or equal to b
```

5.2.5 List Operators

Lists support the following operations:

Length - returns the length of the list

```
set a to [4, 5, 6]
length(a) // 3
```

Access - returns the element at an index

```
set a to [4, 5, 6]
a[0] // 4
```

Update - updates the element at an index

```
set a to [4, 5, 6]
set a[1] to 7
a[1] // 7
```

Insertion - inserts an element at an index and return it

```
**set** a **to** [4, 5, 6]
insert(a, 1, 8) // a == [4, 8, 5, 6]
```

Removal - removes the element at an index and return it

```
set a to [4, 5, 6]
remove(a, 0) // a == [5, 6]
```

Push/Enqueue - inserts an element at the end of the list and return it

```
set a to [4, 5, 6]
push(a, 7)  // a == [4, 5, 6, 7]
enqueue(a, 8) // a == [4, 5, 6, 7, 8]
```

Pop/Dequeue - removes the last element and returns it

```
set a to [4, 5, 6]
pop(a) // a == [4, 5]
dequeue(a) // a == [5]
```