Pseudo: Language Reference Manual

Kristy Choi (eyc2120), Kevin Lin (kl2806),
Benjamin Low (bkl2115), Dennis Wei (dw2654), Raymond Xu (rx2125)

February 22, 2017

Contents

1 Introduction **2**

2 Lexical Conventions **3**

2.1 Comments . . . 3
2.2 Identifiers . . . 3
2.3 Keywords . . . 3
2.4 Constants . . . 4
2.4.1 Numerical Constants . . . 4
2.4.2 Escape Character Constants . . . 4
2.5 Strings . . . 4

3 Types **4**

3.1 Primitive Data Types . . . 4
3.1.1 `num` . . . 4
3.1.2 `bool` . . . 5
3.1.3 `string` . . . 5
3.2 Collections . . . 5
3.2.1 `Lists` . . . 5
3.2.2 `Maps` . . . 6
3.3 Objects . . . 6
3.3.1 Objects . . . 6
3.4 Type Inference . . . 7
3.5 Automatic Initialization . . . 8

4 Operators and Expressions **8**
4.1 Assignment 8
4.1.1 Assignment 8
4.2 Operators 9
4.2.1 Arithmetic Operators 9
4.2.2 Logical Operators 9
4.2.3 String Operators 9
4.2.4 Relational Operators 10
4.2.5 Collection Operators 10
4.3 Declarations 13

5 Control Flow **13**
5.1 Conditionals 13
5.2 For 13
5.3 While 13

6 Scope **14**

7 Functions **14**
7.1 Declarations 14
7.2 Usage 15

8 Additional Examples **15**

1 Introduction

Algorithmic thinking and analysis serve as the cornerstone of all application areas in computer science, equipping students and professionals alike with the ability to tackle challenging problems efficiently and effectively.

Traditional high-level programming languages such as Java and C++ are often unnecessarily verbose for algorithmic definition. Bulky language design complicates programmatic implementations of ubiquitous algorithms such as the graph-traversing `depth-first search` or `shortest paths`, forcing users to set up classes and infrastructure to implement simple ideas in code.

Our language will be perfect for rapidly prototyping algorithms, verifying the behavior of algorithms on given inputs, and for educational purposes, as the syntax is inspired by the easily-readable pseudocode from CLRS, the classic textbook for algorithmic analysis. Following an imperative programming paradigm and using type inference with intuitive keywords to maximize human readability, we hope to facilitate designing and implementing algorithms for all users.

2 Lexical Conventions

2.1 Comments

Single line comments begin with //. Multi-line comments begin with /* and end with */. Comments do not nest.

2.2 Identifiers

Identifiers, or names, are used to describe the various components of Pseudo. They are composed of a sequence of alphanumeric characters and/or the character _, where the first character must be alphabetic. Identifiers are case sensitive - uppercase and lowercase characters are considered distinct.

```
/* Valid identifiers */
my_int = 10
flag = True

/* Invalid identifiers */
_int = 10
1thousand = 1000
```

2.3 Keywords

The list of identifiers reserved as keywords are below:

and	for	or
assert	sort	return
break	if	None
continue	True	swap
def	in	union
else	is	while
elseif	not	
False	print	

We provide some examples of how these keywords can be used:

```
a = 2
b = 4
assert a == b // Exception
assert (a == b) == False // True
2 not in [1, 2, 3] // False
```

We will provide more examples specific to each keyword in later sections.

2.4 Constants

Pseudo allows for several types of constants.

2.4.1 Numerical Constants

Numerical constants consist of a sequence of digits from 0-9, including a hyphen (-) for negative numbers and a decimal point (.) for floating point numbers. Pseudo only supports decimal numbers - number systems in other bases (e.g. binary, hexadecimal) are not allowed.

2.4.2 Escape Character Constants

Escape character constants consist of two characters - a slash (\), and another character that designates the type of escape character it is. Below is a list of all the escape character constants available in Pseudo and their definitions:

\n - newline character
\t - tab character
\r - return character
\" - double-quote character
\\- backslash character

2.5 Strings

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

3 Types

3.1 Primitive Data Types

There are three primitives in pseudo: `num`, `bool`, and `string`.

3.1.1 `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:

```
x = 3
y = 3.0
```

Because of this normalization, boolean operations ignore notation as well.

```
x = 3
y = 3.0
x == y // True
```

3.1.2 `bool`

`bool` represents a simple boolean value, either `True` or `False`. They can be declared as follows:

```
boolean_one = True
boolean_two = False
```

3.1.3 `string`

`string` is a primitive data type in Pseudo. They are denoted by enclosing the desired text in double quotes. The string datatype supports all ASCII characters. To insert the " character in a string, use \" to avoid ending the string.

```
str_one = "This is a \"string\"" // This is a "string"
str_two = "this Is %$# another %!@) \nstring"
```

3.2 Collections

There are two types of collections that are built into Pseudo: `Lists` and `Maps`. Both are strongly typed. Collections are typically named with a single uppercase letter such as `A, G, and M`, but this is only a recommended convention.

3.2.1 `Lists`

A `List` is represented by a sequence of comma-separated elements is 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.

```
A = [1, 2, 3, 4, 5]
A[0] = 4 // [4, 2, 3, 4, 5]
```

```
G.nodes = [a, b, c] // a list of nodes
a.adj = [b, c] // an adjacency list of neighboring nodes
```

3.2.2 Maps

Maps represent the traditional hash map that maps unique keys to values. They are represented by two curly braces {}, where each key is separated from its value by a colon :, and each key-value pair is separated by commas. All of the keys in a map must be of the same type as each other, and all of the values in a map must be of the same type as each other. Entries to Maps can be added by inserting new keys with their respective values using the square brackets [] for indexing the Map.

```
M = {"one": 1, "two": 2, "three": 3}
M["one"] // 1
M["four"] = 4 // assignment
// M = {"one": 1, "two": 2, "three": 3, "four": 4}
```

Entries in a Map can be overwritten by assigning a new value to an existing key. This data structure also supports deletion of (key, value) entries in addition to appending and access. The key values in the Map are not stored in any particular order.

3.3 Objects

Objects in Pseudo represent containers of data types. They can be used to represent more complicated structures such as nodes in a graph.

3.3.1 Objects

Objects are collections of data types that are accessed via the dot (.) operator. Consider a Person object which has the fields name and age. One can create and set the fields of such an object with the following code:

```
person.name = "John Doe"
person.age = 42
```

These values can be referenced later in the program as well. For example, the expressions

```
person.name == "John Doe"
person.age + 5
```

would return True and 47 respectively.

In addition to containing primitive data types, objects can also contain other objects. Consider the following declaration of a family:

```
family.size = 4
family.surname = "Doe"
family.mom.name = "Jane Doe"
family.mom.age = 43
family.dad.name = "John Doe"
family.dad.age = 42
child_one.name = "Tim Doe"
child_one.age = 12
child_two.name = "Tina Doe"
child_two.age = 9
family.children = [child_one, child_two]
```

Within this declaration of a Family object named `family`, we can see that it contains several data types. There are primitive fields `size` and `surname` of type `num` and `string`. However, there are also Person sub-objects of `mom` and `dad` each with fields of `name` and `age`. Lastly, there is even a list of Person objects `children` containing `child_one` and `child_two`. All types are inferred, with `family` having field types `int, string, object, object,` and `List<object>`.

Objects do not have in-built member functions. Functions on objects must be defined as a static function independent of the object itself.

3.4 Type Inference

Pseudo contains a robust type inference system. Given an expression, it will be determined at compile time what type each variable is an instance of. For example, given the expression

```
num_example = 3
```

it will be inferred that `num_example` is of type `num`.

This principle extends to more advanced data types as well. For example, the expressions

```
sample_list = [1, 2, 3, 4]
sample_map = {"one": 1, "two": 2}
```

will infer `sample_list` and `sample_map` to be of types `List<num>` and `Map<string:num>` respectively.

Lastly, this type inference applies to objects as well. For example, consider the following sample program:

```
sample_object.name = "my name"
sample_object.value = 0
```

`sample_object` will be inferred to be of a type object with two fields, `name` which is of type `string` and `value` which is of type `num`.

Type inference applies to more complex objects that contain object fields as well. Consider the following program:

```
sample_object.name = "my name"
sample_object.sub_object.sub_value = 3
sample_object.sub_object.sub_list = [1, 2, 3]
```

`sample_object` is inferred to be an object with two fields, `name` of type `string` and `sub_object` of type `object`. Furthermore, `sub_object` is inferred to have fields `sub_value` of type `num` and `sub_list` of type `List<num>`.

If an object cannot be inferred when needed, a compilation error will occur.

```
print a // Compilation Error: Ambiguous Type
```

3.5 Automatic Initialization

All variables are automatically initialized to default values.

Primitives are automatically initialized to a default value depending on their type. Collections are automatically initialized to their empty states. Objects' fields are automatically initialized recursively to their default values.

Type	Default Value
num	0
bool	False
string	None
List	[]
Map	{}

4 Operators and Expressions

4.1 Assignment

4.1.1 Assignment

The = operator is used to assign the value of an expression to an identifier.

```
A = [1, 2, 3]
```

With type inference, the variable A 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
```

4.2 Operators

4.2.1 Arithmetic Operators

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

4.2.2 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.

4.2.3 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.

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

String concatenation is denoted by the binary `+` operator.

```
a = "Hello"
b = " world!"
c = a + b // "Hello world!"
```

```
a = True
b = False
print not a // False
print a and b // False
print a or b // True
```

4.2.4 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 `is` keyword is valid for simple objects, collections of primitives, and collections of simple objects. The `==` and `!=` operators are valid for primitives and lists containing primitives but not for objects or lists containing objects.

```
a = 1
b = 1
print a == b // True
print a is b // True

c.adj = b
b.adj = c
print b is c // False
print c is c // True
print b == c // Error

A = [1, 2, 3]
B = [1, 2, 3]
print A == B // True
```

Unfortunately, objects cannot be compared, and attempts to do so will result in a compilation error.

```
john.name = "John Doe"
john.age = 42
jane.name = "Jane Doe"
jane.age = 45
john == jane // Compiler Error: Object Comparison
```

However, the `is` command can be used with objects.

The following chart explains what can be compared with the `==` and `is` operators.

Type	`==`	`is`
primitive	✓	✓
primitive collection	✓	✓
object	X	✓
object collection	X	✓

4.2.5 Collection Operators

Collection operators allow for convenient manipulation of the Collection data types.

Lists

Lists support the following operations:

Length - returns the length of the list

```
a = [4, 5, 6]
a.length // 3
```

Access - returns the element at an index

```
a = [4, 5, 6]
a[0] // 4
```

Update - updates the element at an index

```
a = [4, 5, 6]
a[1] = 7
a[1] // 7
```

Insertion - inserts an element at an index and return it

```
a = [4, 5, 6]
a.insert(1, 8)
// a == [4, 8, 5, 6]
```

Removal - removes the element at an index and return it

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

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

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

Pop - removes the last element and returns it

```
a = [4, 5, 6]
a.pop() // 6
// a == [4, 5]
```

Dequeue - removes the first element and returns it

```
a = [4, 5, 6]
a.dequeue() // 4
// a == [5, 6]
```

For each `e` in `L` - iterate over a list

```
a = [4, 5, 6]
for elem in L:
    print a
// 4
// 5
// 6
```

Maps

Maps support the following operations:

Contains - returns whether or not a key is in the list

```
m = {"foo": 2, "bar": 3}
"foo" in m // True
```

Lookup - looks up the associated value for a key

```
m = {"foo": 2, "bar": 3}
m["foo"] // 2
```

Update - sets a new value for a key, overwriting its old value if it had one

```
m = {"foo": 2, "bar": 3}
m["foo"] = 5
```

Removal - deletes a key from the map

```
m = {"foo": 2, "bar": 3}
m.remove("foo")
m["foo"] // Exception
```

Iteration - iterate over each key in the map

```
m = {"foo": 2, "bar": 3}
for k in m:
    print k
// foo
// bar
```

4.3 Declarations

5 Control Flow

All statements in the language are executed in sequence.

5.1 Conditionals

The `if` statement is used for conditional execution of a series of expressions. Each statement is separated by a colon (:) to signal the end of a clause. The `elseif` keyword catches conditions that are skipped by `if`, and the `else` keyword catches all other cases.

```
if boolean-expression:
    expression
elseif boolean-expression:
    expression
else:
   expression
```

5.2 For

The `for` statement is used to iterate over the elements in a sequence (or some other iterable), allowing the user to repeatedly execute statements nested inside the loop. The `to` keyword can be used to specify a range to iterate over a starting and ending `num` type. The starting `num` is inclusive and the ending `num` is not inclusive.

```
for expression in expression_list:
    expression

for i = 0 to 10: // prints 0 to 9
    print i
```

The `break` keyword can be used to exit early, as is the case for the `while` loop. The `continue` keyword is used to go to the end of the loop.

5.3 While

The `while` statement is another way to continuously execute a statement so long as the value of the boolean expression evaluates to True. This expression is evaluated prior to execution of the nested statement.

```
while boolean-expression:
    statement
```

6 Scope

The lexical scope of variables follows from the structure of the program. Variables declared at the outermost level extends from their definition through the end of the file in which they appear. Function definitions, conditionals and loops create their own local scope. If a variable is a defined in a higher scope then assignment to that variable changes that variable.

```
a = 10
if True:
    a = 20
    b = 20
print a // 20
print b // error
```

7 Functions

7.1 Declarations

Functions are declared with the **def** keyword, the name of the function, and the parameters being passed into the function surrounded by parentheses, followed by a colon. The naming convention follows that of identifiers - it must begin with an alphabetic character and may consist of any combination of alphanumeric characters and _. If there are multiple parameters, they are separated by commas.

```
def ADD(a, b):
    return a + b
```

Return values and parameters of the function will determined based on type inference, whether from a previous initialization/usage of the variable, or from the operations performed on it in the function.

```
def INCREMENT(a):
    return a + 1 // returns num a + 1
```

The scope of a function is determined by its indentation. All lines that are one tab greater than the function declaration's indentation are considered part of the function. Pseudo knows when a function's scope has ended once it finds a line with the same indentation as the function declaration or a line that has less tabs.

7.2 Usage

User-defined functions may be called simply by providing the function name and the required parameters.

```
a = 3
b = 2
print ADD(3, 2)
```

In-built functions for a data type may be called by providing the variable of that data type, a period, and then the function name and its required parameters.

```
a.append(2) // a is a List
```

Parameters are passed into functions by reference. This means that any non-primitive type variable passed into a function (e.g Lists, Maps, Objects) can have its contents modified even outside the function's scope. For example:

```
a = [1,2,3,4,5]

def modify(a):
    a[0] = 0
    return a

modify(a) // [0,2,3,4,5]
```

Primitives cannot be passed by reference. A primitive variable declared outside a function will always retain its value unless reassigned within its scope. Any function that the variable is passed into will not modify the value of the original variable.

8 Additional Examples

```
def BFS-CONNECTIVITY(G, s, t):
    for u in s.adj:
        Q.push(u)
    while Q.length > 0:
        v = Q.pop()
        if v is t:
            return True
        visited.append(v)
        for u in v.adj:
            if u not in visited:
                Q.push(u)
```

```
    return False

def MAIN():
    a.adj = [b, c]
    b.adj = [a, d]
    c.adj = [a, b]
    d.adj = [b]
    G = [a, b, c, d]
    print BFS-CONNECTIVITY(G, a, d)


$ pseudo bfs.clrs
$ ./bfs
>> true
```