JQER Reference Manual

Jiaxuan Pan (jp4131)
Qianjun Chen (qc2300)
Eurey Noguchi (yn2377)
Roger Lu (jl5822)

1. Introduction

JQER is a statically typed language with Python-like syntax for binary-tree data structures and operations on them. The main goal of JQER is to simplify the operations on binary-tree data structures such as searching and traversing. The language integrates a Tree as a built-in data type and their associated operating modules. Overall, our goal is to make tree data structures and algorithms easier to write and use.

2. Lexical Conventions

2.1 Line Structure

A JQER program is divided by a number of logical lines.

2.1.1 Logical Lines

The end of a logical line is denoted by the NEWLINE token. Logical lines cannot span multiple lines unless there is explicit line joining.

2.1.2 Blank lines

A logical line that only contains spaces, tabs, and comments are ignored.

2.1.3 Indentation

Whitespaces from tabs at the beginning of a logical line is used to determine the indentation level of the line. The indentation level is then used to resolve the grouping of statements as statement blocks. Here, the control flow keywords such as if, else, for, and while must be followed by a colon and the lines inside the control flow must have the same indentation level, unless there are further control flow keywords.

```
def square_even(x):
    if x % 2 == 0:
        return x * x
```

2.1.4 Whitespaces between tokens

Whitespaces except at the beginning of a logical line or in string literals are used to separate tokens. It is needed if the concatenation of two tokens could be interpreted as a different token.

2.2 Comments

JQER has both single and multi-line comments. A single line comment begins with a # (single pound symbol) and a multi-line comment is surrounded by “”” (three quotation marks).

```
# This is a single line comment.
'''
This is a multi-line comment
Spanning two lines
'''
```

2.3 Identifiers (Names)

An identifier starts with an ASCII letter and can include other ASCII letters, decimal digits, as well as underscores. It is case sensitive and cannot be one of the keywords of JQER.

2.4 Keywords

The following identifiers are reserved for the use as keywords of the language and cannot be used as normal identifiers.

```
int char string bool tree None
if else elif for while def return and or not continue break in range
True False
```

2.5 Literals

Literals represent constant values of built-in types.

2.5.1 Int Literal

An int literal consists of any sequence of integers between 0 and 9.

2.5.2 Bool Literal

A bool literal can either be True or False.

2.5.3 Char Literal

A char literal is one or two characters surrounded by single (‘) quotes. If two characters are used, they are used to represent escape sequence with the first character being a backslash, which are

```
\\ backslash
\' single quote
\" double quotes
\n new lines
\t tab
```

2.5.4 String Literal

A string literal is a sequence of characters surrounded by double (“) quotes.

2.5.5 Tree Literal

A tree literal consists of an ordered triple. The first of the triple describes the value and the second and third are the left and right child of the tree, which are also tree literals or None to specify the end. To initialize a tree, it is required to specify the data type (int, char or string) of value that the tree instance is allowed.

```
int [5 None None]
int [5 [2 None None] [3 [1 None None] None]]
char [‘a’ None None]
```

3. Data Types

3.1 Primitives

Primitives are series of bytes of some fixed length, and there are three primitive types in JQER:

int (4 bytes, 2’s complement)
char (1 byte, ASCII)
bool (1 byte, 00000001 for True, 00000000 for False).

3.2 Mutability

The primitive data types in JQER are immutable, including int, char, and bool. String and Tree are not primitives, but are also immutable. All operations which modify these objects actually return new objects.

3.3 Standard Library

JQER provides a number of built-in types, including String and Tree. These are implemented in JQER as built-in structures, and have associated methods and properties. String and Tree are immutable and support a rich variety of operations. They are fully memory safe.

3.4 None

None is a keyword that returns a reference to a predetermined object of a special type. There is only ever one object of this type.

4. Expression

4.1 Identifier

An identifier is a primary expression, its type is specified by its declaration. An identifier which is declared ‘‘function returning . . .’’, when used except in the function-name position of a call, is converted to ‘‘pointer to function returning . . .’’.

4.2 Constant

An int, char, bool, string, tree is a primary expression.

4.3 Tree

A tree is a primary expression. Its type is specified by “ [literal tree tree] “, where it stores one literal value which can be int, char, bool or string, and it contains two tree expressions that need to be of the same type. To initialize a tree, it is required to specify the data type (int, char or string) of value that the tree instance is allowed.

4.4 Multiplicative operators

4.4.1 Expression * Expression

The binary * operator indicates multiplication. It only supports two int as operands.

4.4.2 Expression / Expression

The binary / operator indicates division. It only supports two int as operands.

4.4.3 Expression + Expression

The result is the sum or concatenation of the expressions. It supports two int as operands which is calculating the sum. It also supports string or chars as operands, which concatenate two

strings or chars and return a new one. If two operands are tree, the second tree will be added to the first tree at the most right pointer, the detail will be covered in section 9.

4.4.4 Expression - Expression

The result is the difference of the operands. It supports two int as operands.

4.5 Relational operators

The relational operators group left-to-right.
The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield False if the specified relation is False and True if it is true.

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

4.6 Equality operators

The == (equal to) and the != (not equal to). The comparison happens between two values.

```
expression == expression
expression != expression
```

4.7 Expression and Expression

The “and” operator returns True if both its operands are True, False otherwise. It guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is False. The operands need to be of type bool.

4.8 Expression or Expression

The “or” operator returns True if either of its operands is True, and False otherwise. It guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is True. The operands need to be of type bool.

4.9 not Expression

The result of the logical negation operator “not” is True if the value of the expression is False, False if the value of the expression is True. The type of the result is bool. This operator is applicable only to bool.

4.10 - Expression

The result is the negative of the expression, and has the same type. The type of the expression must be int.

4.11 identifier = expression

The assignment operator is grouped right-to-left. It requires an identifier as their left operand. The value is the value stored in the left operand after the assignment has taken place. The value of the expression replaces that of the object referred to by the identifier.

4.12 primary-identifier . member-of-structure

An identifier expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the identifier is assumed to have the same form as the structure containing the structure member. The result of the expression is an identifier appropriately offset from the origin of the given identifier whose type is that of the named structure member. The given identifier is not required to have any particular type.

5. Declarations

Declarations are to specify the interpretation which JQER gives to each identifier. They do not necessarily reserve storage associated with the identifier. Declarations have the form

```
declaration:
      type-specifier declarator-list-opt
```

5.1 Type specifiers

The type specifiers are

```
type-specifier:
      int
      char
      string
      bool
      tree
```

5.2 Declarators

The syntax of declarators:

```
declarator:
      Identifier
      def declarator( )
```

5.3 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type. Each declarator contains exactly one identifier. It is this identifier that is declared.
If a declarator is

```
def D( )
```

then the contained identifier has the type ‘‘function returning ...’’, where ‘‘ ... ’’ is the type which the identifier would have had if the declarator had been simply D.

See more in the function definition section below.

6. Statements

Excepted as indicated, statements are executed from left to right and from top to bottom.

6.1 Expression Statement

Some statements are expression statements, which have the form

```
expression
```

Usually expression statements are assignments or function calls.

6.2 Compound statement

The compound statement is in this form:

```
compound-statement:
      statement-list
statement-list:
      statement
      statement statement-list
```

Where a statement-list is a group of statements.

6.3 Conditional statement

The conditional statement has these forms:

```
if expression:
      statement

if expression:
      statement
(elif expression:
      statement)*
```

```
if expression:
      statement
(elif expression:
      statement)*
else statement
```

Where the notation (...)* means 0 or more of the same statements.

6.4 While statement

The while statement has this form:

```
while expression:
      statement-list
```

The statement is executed repeatedly until the value of the expression becomes False.

6.5 For statement

The for statement has this form:

```
for Identifier in range(int, int):
    statement-list
```

The identifier will be assigned an int value starting from the first int value, and increment by 1 in every iteration until it reaches the second int value - 1. The statement-list will be executed once each iteration until the value of the identifier reaches the int value.

6.6 Break statement

The break statement is:

```
break
```

This is used in while and for loop in order to terminate earlier.

6.7 Continue statement

The continue statement is:

```
continue
```

This is used in while and for loop in order to reach the end of the loop.

6.8 Return statement

The return statement can be:

```
return
return expression
```

In the first case, no value will be returned. In the second case, the value of the expression is returned to the caller of the function.

7. Function Definition

All function declarations are by default and the only type of external definitions for JQER, which means there must be an external definition for the given identifiers somewhere outside the function in which they are declared.

Function definitions have the form:

```
function-definition:
      type-specifier "def" function-declarator ":" function-body "\t"
```

Type-specifier is the same as in 5.1 and represents the return value of the function.

```
function-declarator:
      declarator (parameter-list)
```

Parameter-list is optional.

```
parameterlist:
      parameter , parameterlist

parameter:
      type-specifier identifier
```

Thus the parameters would be declared in function-declarator.

```
function-body:
      declaration-list-opt statement-list

declaration-list-opt:
      declaration "\t" , declaration-list-opt

statement-list:
      statement "\t" , statement-list
```

A simple example of a complete function definition is below, with indentation as tab:

```
int def max(int a, int b, int c):
      int m
      if a > b:
            m = a
      else m = b
      if m > c:
            return m
      else return c
```

8. Scope Rules

JQER has not yet supported or been supported by any external files or libraries, mainly because of the lack of a reliable I/O system. Accordingly, all declarations, definitions and operations would be done as a unit of a single file. All declarations and definitions at the top level of the file would be considered “global” thus extending themselves from their definition through the end of the file in which they appear.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

Since we use tab indentations to represent the beginning of three cases: a function definition, a conditional flow and a loop; with cancelling the tab indentation to represent the end of them, we consider JQER a block-structured language. All declarations and definitions when one or more indentations exist are only valid (considered declared as “local”) within the current level block, which is represented by matching tabs before them. Multiple declarations on the same identifier is allowed between two parallel blocks on the same level or blocks within them with a lower level, but redeclare identifiers already declared in a higher level is an error.

9. Built-in Module

9.1 Tree

Tree is a data type implemented in JQER, it is supported by a variety of useful methods for handling tree-related operations. Since Tree is immutable, any operation on an existing tree will return a new tree.

9.1.1 tree1 + tree2
Returns the combination of tree1 and tree2. tree2 is added to the most-right pointer of the tree1. And two operands must contain the same data type value.
For example:

```
int [2 [1 None None] [3 None None] ]  +  int [4 None None]
```

Returns:

```
[2 [1 None None] [3 None [4 None None] ] ]
```

9.1.1 tree . module(tree1)
There are built-in modules to support operations on tree, they are designed in tree library. There are:

tree.depth (tree1)	Returns the depth of the tree1

tree.bal (tree1)	Take tree1, and return a balanced tree
tree.contains (tree1, int/str/char)	Return true if there is a variable(can be string, char, or char) in tree1, otherwise return false.
tree.left (tree1)	Return the left child of the tree1, it can be None or a sub-tree.
tree.right (tree1)	Return the right child of the tree1, it can be None or a sub-tree.
tree.value (tree1)	Return the value of the tree1, it can be string, int, or char.
tree.inorder (tree1, func1)	Takes a tree and a function and traverses each node inorder applying the function, then return the new tree.
tree.preorder (tree1, func1)	Takes a tree and a function and traverses each node preorder applying the function, then return the new tree.
tree.postorder (tree1, func1)	Takes a tree and a function and traverses each node postorder applying the function, then return the new tree.
tree.dfs (tree1, func1)	Takes a tree and a function and traverses each node with depth first search applying the function, then return the new tree.
tree.bfs (tree1, func1)	Takes a tree and a function and traverses each node with breadth first search applying the function, then return the new tree.

9.2 print(var)

print(var) is used to print the string representation of the variable to standard output. The nature of that representation depends on the type of variable. If variable is one of the primitive types, there is a built-in print for it. For example:

```
ran = range(1, 5)
for i in range(1, 5):
      print(i) # prints 1 2 3 4
```

9.3 Casting

The user can explicitly cast objects from one type to another. These are built-in type casting:

```
str(int) -> str
str(tree) -> str
str(char) -> str
char(str) -> char
```

10. Examples

10.1 Fibonacci

This function returns the integer value of n-th element of the Fibonacci Sequence.

```
int def fib_recur(int n):
    if n <= 1:
        return n
    return fib_recur(n-1) + fib_recur(n-2)
```

10.2 Printing a Pedigree Genealogical Chart

```
family = ["Me" ["Mom" ["MomGM" None None] [MomGD None None]] ["Dad"
["DadGM" None None] [DadGD None None]]]

tree.bfs(family, print) # print my family in breadth first search
```