The Quby Language Specification

Quby is a new modern compiled language that borrows features from Ruby and Turing. However, from our point of view, it is a modification and extension to PT Pascal. The main differences between PT and Quby are: allowing the intermixing of declarations and statements, the change to Turing-style comments, Ruby-like syntax for programs and statements, Turing-like syntax for declarations, removal of the PT end-of-statement semicolon, the addition of Ruby **modules**, the replacement of the **char** data type with the Turing **string** type, the addition of Ruby's **else** to **case** statements, the addition of Ruby's **elsif** to **if** statements, the addition of Ruby's **unless** statement, and the replacement of the PT **repeat** statement with a Ruby/Turing-like general **do** statement.

In the following,
[item] means the item is optional, and
{ item } means zero or more repeated occurrences of the item.
Keywords are shown in **bold face**, and predefined identifiers in ***bold italics***.
Syntactic forms (non-terminals) not defined here are as defined in original PT Pascal.

1. Comments
Quby changes to Turing-like comments, which consist of **%** to end of line. PT Pascal **{** ... **}** and **(*** ... ***)** comments are deleted from Quby.

2. Programs
A Quby program is :
using identifier { , identifier }
{ *declarationOrStatement* }

Where declaration and statement are as described below. Note that unlike PT, Quby allows the intermixing of declarations and statements.

A *declarationOrStatement* is one of:
a. declaration
b. statement

3. Declarations
Quby declarations are similar to PT, except for changes to modern Turing-like syntax.

A declaration is one of :
a. *constantDeclaration*
b. *typeDeclaration*
c. *variableDeclaration*
d. *routine*
e. *module*
f. *;*

A *constantDeclaration* is:

val identifier = constant

A *typeDeclaration* is:

type identifier : type

A *variableDeclaration* is:

var identifier { , identifier } : type

Where constant and type are as described in the PT Pascal syntax.
Note that Quby has no semicolons between or at the end of declarations, since they are redundant. However, semicolons are accepted as a null declaration.

4. Statements
Quby changes the syntax of PT Pascal statements to Ruby style, removing the **begin**...**end** statement by using Ruby statement sequences instead. Statements in Quby have no semicolons. Quby adds Ruby's **else** case to **case** statements, adds Ruby's **elsif** to **if** statements, adds Ruby **unless** statements, and replaces the PT **repeat** statement with a Turing-like general **do** statement.

A statement is one of the following:

a. variable = expression
b. routine_identifier [(expression { , expression })]
c. **if** expression **then**
{ *declarationOrStatement* }
{ **elsif** expression **then**
{ *declarationOrStatement* } }
[**else**
{ *declarationOrStatement* }]
end
d. **unless** expression **then**
{ *declarationOrStatement* }
[**else**
{ *declarationOrStatement* }]
end
e. **case** expression
{ **when** constant { , constant } **then**
{ *declarationOrStatement* } }
[**else**
{ *declarationOrStatement* }]
end
f. **while** expression **do**
{ *declarationOrStatement* }
end
g. **do**
{ *declarationOrStatement* }
break if expression
{ *declarationOrStatement* }
end
h. ;

Where expression and constant are as defined in the PT Pascal syntax. Note that Quby has no semicolons between or at the end of statements, since they are redundant. Quby statements simply end where the next statement begins, or at end of file. However, semicolons are accepted as a null statement (which does nothing).
Quby retains PT Pascal input/output routines (read, write, writeln, etc.)

5. Routines
Quby routines (methods) are like PT routines except for the change to Ruby-like syntax. A routine is:

def [*****] identifier ([[**var**] identifier : type_identifier
{ , [**var**] identifier : type_identifier }])
{ *declarationOrStatement* }
end

The optional ' * ' before the routine name indicates a public routine (one that can be called from outside of the module it is declared in). See "Modules" below.

6. Modules
One of the most powerful modern software engineering tools is the concept of information hiding. Modern programming languages include special syntactic forms for information hiding such as classes or modules. One of the weaknesses of PT Pascal is its lack of such a feature. Quby solves this problem by adding the Ruby module construct. Like “anonymous classes” in C++ and Java, Quby modules are single-instance. A module is:

module identifier
{ *declarationOrStatement* }
end

The purpose of the module is to hide the internal declarations, data structures and routines from the rest of the program. Outside of the module, the module’s internal data cannot be accessed, and only those routines declared to be public (using ' ***** ') may be called.

The purpose of the statements in the module is to initialize the module's internal data. During execution, the statements of each module are executed to initialize the module before commencing execution of the statements of the main program.

6. Strings

Text manipulation in standard Pascal is painful because of the necessity of shovelling characters around one-by-one. Modern languages like Turing, Ruby, Javascript and Swift solve this problem by providing a built-in varying-length string data type or module. Quby adds the string data type to PT (replacing the char data type, which is deleted from Quby).

A string literal is any sequence of characters except the double quote enclosed between double quotes. Example:

```
"This is a string"
```

String variables take on the length of the last value that they were assigned. Example:

```
s = “hi"          # length of s is now 2
s = "there"       # length of s is now 5
```

There is an implementation-defined maximum length (1,023 characters) for string values. Varying length strings are implemented by storing the string value with an extra trailing character (ASCII NUL, the byte 0) marking the end of the string, as in Turing and C. Each ***string*** variable is actually allocated a fixed amount of storage (1,024 bytes) within which the string value is stored.

Strings can be input and output to/from text files. On input, the string read consists of the characters from the next input character to the end of the input line. Unlike arrays of ***char*** in PT Pascal, values of type ***string*** can be assigned and compared as a whole. Besides assignment and comparison, there are three new operations on strings: concatenation, substring and length.

Concatenation of strings is denoted by the **+** operator as in Turing. For example:

```
"hi " + "there" == "hi there"
```

It is an error to concatenate two strings if the sum of their lengths exceeds the implementation defined maximum (detected at run time).

The substring operation is denoted by the ' **$** ' operator:

```
expression $ expression .. expression
```

The first expression must be a string expression. The second and third expressions, which must be integer expressions, specify the (one origin) first character position and last character position of the substring respectively. Example:

```
"Hi there" $ 4 .. 6 == "the"
```

The precedence of the $ operator (including the .. portion) is higher (more binding) than that of *****, **div** and **mod** and lower than that of **not**.

The string length operator has the form:

```
# expression
```

Where the expression must be a string expression. Example:

```
# "Karim" == 5
```

The precedence of **#** is the same as that of **not**.

The string index operator has the form:

```
expression ? expression
```

Where each expression must be a string expression. The result is the (one-origin) index of the first occurrence of the second string expression in the first. If there is no such occurrence, the result is 0. Example:

```
"Hello there" ? "the" == 7
```

The precedence of the **?** operator is the same as that of *****, **div** and **mod**.

7. Operators.

Quby changes to Ruby operator notation for assignment, equality and inequality. Thus PT Pascal's **:=** becomes **=**, **=** becomes **==**, **not** becomes **!**, and **<>** becomes **!=**.