CISC / CMPE 458 Course Project Winter 2020
The Qust Programming Language (UPDATED)

J.R. Cordy - January 2020
(Updated 9 February 2020)

The course project this year consists of implementing a compiler for the programming language *Qust.* Qust ("not Quite rUST") is an extended subset of Mozilla Corporation's *Rust* programming language, which parts of the Firefox web browser, the Thunderbird email client, and the Bugzilla bug-tracking system are all programmed in.

We will begin with the PT Pascal compiler, and modify each of its phases to handle Qust instead of PT, one at a time. For each phase, I will provide suggestions on how to make the necessary modifications to PT. However, you are not required to pay any attention to these suggestions, and you may choose to design modifications of your own. (Theorem: there are an infinite number of good ways to skin any given cat. Corollary: I don't know them all.) However, the Qust language that you implement must meet exactly the specifications below.

Teams

The project will be undertaken in teams of four students. You are expected to choose your teammates on your own and submit team sign-up sheets to me by the beginning of class on Wednesday, January 22. The sign-up sheet must include the full name and student number of each team member, and must be signed by all four. Students who have not chosen teammates by January 22 will be assigned teammates at random.

The project forms an integral part of the course and you will be examined on material which can only be learned by actively taking part in each phase. You should therefore take care that every member of the team has an opportunity to contribute to every phase of the project. Assigning each phase's work to one person is not an appropriate way to split the work, because the phases require different amounts of work, ranging from 15% of the work for the parser phase to 25% of the work for the semantic phase, and because you will not all learn about all the phases that way. Assigning each language feature to one person is also not the best way to split the work, because some of the extensions will involve a lot of work in some phases and none at all in others, so you will not learn about all the phases that way.

The best way to manage the team is to hold team meetings to design overall solutions for each phase, estimate the work required to do the modifications, and then split the work as evenly as possible between team members. Team meetings are essential to the success of the project, and you should be sure to take time to hold them before each phase.

The Qust Language

Qust is a modular programming language with features similar to other modern languages like Rust, Swift, Ruby, and Python. However, from our point of view, it is a modification and extension to PT Pascal. The main differences between PT and Qust are Rust syntax for programs, statements and declarations, the addition of Rust modules, the replacement of the ***char*** data type with the Rust ***str*** type, the addition of Rust's **else if** to **if** statements, the replacement of the PT **repeat** statement with Rust's general **loop** statement, and the replacement of PT's **case** statement with Rust's **match** 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 (nonterminals) not defined here are as defined in original PT Pascal.

1. Comments.

Qust changes to Rust-style comments, which consist of **//** to end of line and **/*** … ***/**.
PT Pascal **{** ... **}** and **(*** ... ***)** comments are deleted from Qust.

2. Programs.

A Qust *program* is :

mod main (identifier { **,** identifier } **) {**
    { declarationOrStatement }
}

Where *declaration* and *statement* are as described below.

Execution of the Qust program consists of initializing the declarations and executing the statements in the order that they appear.

3. Declarations.

Qust declarations are mostly similar to PT, except for the introduction of immutable variables, and changes to modern Rust-like syntax.

A *declaration* is one of :

a. constDeclaration
b. typeDeclaration
c. variableDeclaration
d. routine
e. module

A *constantDeclaration* is:

const identifier **=** constant { **,** identifier **=** constant } **;**

A *typeDeclaration* is:

type identifier **=** type **;**

A *variableDeclaration* is:

let [**mut**] identifier [**:** type] [**=** expression]
    { **,** [**mut**] identifier [**:** type] [**=** expression] } **;**

Where *constant* and *expression* are as described in the PT Pascal syntax.

Unlike PT, Qust has both mutable and immutable variables. An immutable variable (declared without **mut**) is read-only, and cannot be assigned to after its initial value. As in PT, **const** declarations take on the type of their value. If the type is omitted in a variable (**let**) declaration, the type is assumed to be integer (***int***), and the variable must have an initial value.

4. Types.

A *type* is:

a. simpleType
b. **[** simpleType **:** constant **]**
c. **file of** simpleType

Where *simpleType* is a named type (including ***int***, ***str***, ***bool***)**.** PT range types (i.e., constant .. constant) are deleted from Qust. Form (b) is an array type. Like PT, Qust arrays are 1-origin and are subscripted using square brackets **[ ]**. However, PT subscript ranges (i.e., constant .. constant) are deleted from Qust. In Qust, all arrays are indexed starting from 1.

5. Routines.

Qust routines (functions) are like PT Pascal routines except for the change to Rust-like syntax.

A *routine* is :

```
[ pub ] fn identifier ( [ [ mut ] identifier : type_identifier
                          { , [ mut ] identifier : type_identifier } ] ) {
        { declarationOrStatement }
}
```

The optional **pub** 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. Parameters declared using **mut** are *mutable* parameters, passed by reference, like **var** parameters in PT, and can be assigned to. Parameters declared without **mut** are *immutable* parameters, passed by value, like non-**var** parameters in PT, and cannot be assigned to. As in PT, arrays must always be passed by reference (i.e., using **mut**).

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. Qust solves this problem by adding Rust-like modules. Like “anonymous classes” in C++ and Java, Qust modules are single-instance.

A *module* is :

```
mod identifier {
        { declarationOrStatement }
}
```

The purpose of a 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 as *public* (using **pub**) may be called.

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

7. Statements.

Qust changes the syntax of PT Pascal statements to Rust style, replacing the **begin**...**end** statement with explicit **{ }** brackets in each statement form. Qust replaces PT’s **repeat** statement with Rust's general **loop**, replaces PT's **case** statement with Rust's **match** statement, and adds Rust's **else if** form to **if** statements. Qust also adds the Qust short form assignments **+=** and **-=** for incrementing / decrementing simple variables.

A *statement* is one of the following :

```
a.   variable = expression ;
b.   variable_identifier [ += | -= ] expression ;
c.   routine_identifier ( [ mut ] expression { , [ mut ] expression } ) ;

d.   if expression {
             { declarationOrStatement }
     { } else if expression {
             { declarationOrStatement } }
     [ } else {
             { declarationOrStatement } ]
     }

e.   while expression {
             { declarationOrStatement }
     }

f.   loop {
             { declarationOrStatement }
             break if expression ;
             { declarationOrStatement }
     }

g.   match expression {
        | constant { | constant } =>
             { { declarationOrStatement } }
       { | constant { | constant } =>
             { { declarationOrStatement } } }
        | _ =>
             { { declarationOrStatement } }
     }

h.   ;
```

Where *variable*, *expression* and *constant* are as defined in the PT Pascal syntax.

Unlike PT, routine calls in Qust require an argument list ( ) even if there are no arguments. Arguments with **mut** are mutable (assignable, reference) arguments, and can only be passed mutable variables, to mutable (**mut**) formal parameters. Qust retains all of the PT Pascal input/output routines, but <u>renames</u> *write* to *print* and *writeln* to *println*.

8. Sequences of Declarations and Statements.

Unlike PT Pascal, Qust does not require declarations to precede statements, and they may be arbitrarily intermixed. The scope of declarations in a Qust statement sequence is from the declaration itself until the end of the enclosing **{ }** block.

A *declarationOrStatement* is one of :

a. declaration
b. statement

In Qust, semicolons are part of statements and declarations rather than separators between them as in PT. However, the null statement (statement form (h) above) allows additional semicolons to appear between declarations and statements even when not required.

9. Strings.

Text manipulation in standard Pascal is painful because of the necessity of shovelling characters around one-by-one. Modern languages like Rust solve this problem by providing a built-in varying-length string data type or library. Qust adds the ***str*** data type to PT (replacing the ***char*** data type, which is <u>deleted</u> from Qust).

A ***str*** 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 C. Each ***str*** variable is 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 and streams. 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 ***str*** cannot be subscripted, and can be assigned and compared as a whole. Besides assignment and comparison, there are four new operations on strings: concatenation, repetition, substring and length.

Concatenation of strings is denoted by the + operator. 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 (1,023, detected at run time).

Repetition of strings is denoted by the * operator. For example :

"hi " ***** 4 = "hihihihi"

The second operand must be an integer expression. It is an error to repeat strings if the result length exceeds the implementation-defined maximum (1,023, detected at run time).

The Qust substring operation is denoted by the **/** operator. The general form is :

expression **/** expression **:** expression

The first expression must be a ***str*** expression. The second and third expressions, which must be integer expressions, specify the (1 origin) first character position and last character position of the substring respectively. Example :

"Hi there" **/** 4 **:** 6 = "the"

The precedence of **/** (including the **:** part) is the same as *****, **/**, **%** and **&&**.

The string length operator has the form :

? expression

Where the expression must be a ***str*** expression. Example :

? "Jim" = 3

The precedence of **?** is the same as that of **!**.

10. Operators.

In Qust, all of the standard operators are changed to Rust notation instead of PT. In particular:

PT	Qust	PT	Qust
:=	=	**and**	&&
div	/	**or**	||
mod	%	**not**	!
=	==		
<>	!=		

11. Example Qust Program

```
// Primes: determines the primes up to maxprimes using the sieve method
mod main (output) {
   const maxprimes = 100;

   // Prime flags
   const prime = true;
   const notprime = false;

   mod flags {
      let mut flagvector: [bool : maxprimes];

      pub fn flagset (f: int, tf: bool) {
          flagvector [f] = tf
      }

      pub fn flagget (f: int, mut set: bool) {
          set = flagvector [f]
      }

      // Everything begins as prime
      let mut i = 1;
      while i <= maxprimes {
          flagvector [i] = prime;
          i += 1
      }
   }

   // Main program
   let mut isprime: bool;

   // Pick out multiples of each prime factor and set these to notprime
   let mut multiple = 2;
   let mut factor = 2;
   while factor <= maxprimes / 2 {

      // Set multiples of factor to notprime
      multiple = factor + factor;
      while multiple <= maxprimes {
         flagset (multiple, notprime);
         multiple += factor
      }

      // Set factor = next prime
      factor += 1;
      loop {
         flagget (factor, mut isprime);
         break if (factor > maxprimes / 2) || isprime;
         factor += 1
      }
   }

   // Now report the results
   print("The primes up to ", maxprimes:1, " are:");
   println();
   factor = 2;
   while factor <= maxprimes {
      flagget (factor, mut isprime);
      if isprime {
         print (factor:4)
      }
      factor += 1
   }
   println()
}
```