The Qust Programming Language (UPDATED)

J.R. Cordy - January 2020
(Updated 24 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.

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:

```rust
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:

```rust
const identifier = constant ( , identifier = constant ) ;
```

A typeDeclaration is:

```rust
type identifier = type ;
```

A variableDeclaration is:

```rust
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:

```rust
{ , [ 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`).


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:

```rust
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:

```rust
a. variable = expression ;
```

```rust
b. variable_identifier [ += ] expression ;
```

```rust
c. routine_identifier ( [ [ mut ] expression { , [ mut ] expression } ] );
```

```rust
d. if expression { 
{ declarationOrStatement } 
} else if expression { 
{ declarationOrStatement } 
} else { 
{ declarationOrStatement } 
}
```

```rust
e. while expression { 
{ declarationOrStatement } 
}
```

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

```rust
g. match expression { 
| constant ( I constant ) => 
{ { declarationOrStatement } } 
| I constant ( I constant ) => 
{ { declarationOrStatement } } 
| _ => 
{ { declarationOrStatement } } 
}
```

```rust
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 renames `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:

```rust
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 deleted 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:

```rust
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:

<table>
<thead>
<tr>
<th>PT</th>
<th>Qust</th>
<th>PT</th>
<th>Qust</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>=</td>
<td>:=</td>
<td>=</td>
</tr>
<tr>
<td>div</td>
<td>/</td>
<td>or</td>
<td></td>
</tr>
<tr>
<td>mod</td>
<td><code>%</code></td>
<td>not</td>
<td><code>!</code></td>
</tr>
<tr>
<td>=</td>
<td><code>==</code></td>
<td>&lt;=</td>
<td><code>!=</code></td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td><code>!=</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 11. Example Qust Program

// Primes: determines the primes up to maxprimes using the sieve method
```rust
tfn main (output) {  
  const maxprimes = 100;  
  // Prime flags  
  const prime = true;  
  const notprime = false;  
  mod flags {  
    let mut flagvector: [bool : maxprimes];  
    fn flagset (f: int, tf: bool) {  
      flagvector [f] = tf;  
    }  
    fn flagget (f: int, mut set: bool) {  
      set = flagvector [f];  
    }  
    // Everything begins as prime  
    let mut i = 1;  
    while i <= maxprimes / 2 {  
      // Set multiples of factor to notprime  
      let mut multiple = 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;  
      }  
    }  
  }  
  // 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 {  
    flagset (factor, mut isprime);  
    if isprime {  
      print (factor:4);  
    }  
    factor += 1;  
  }  
  println();  
}
```