Cryptal Language Reference Manual

Carolina Almirola | Jaewan Bahk | Rahul Kapur | Michail Oikonomou | Sammy Tbeile
ca2636 | jb3621 | rk2749 | mo2617 | st2198

1 Types

Types are used to store data in different formats. A variable must contain a type and name. Names can be alphanumeric and can optionally contain the underscore character('_'). Names are also case sensitive. Reserved keywords cannot be used as names (see Keywords section for list)

1.1 Basic Data Types

The basic data types are similar to their C equivalents.

1.1.1 int

As in C, an int represents a 4-byte (32 bit) signed (two's complement) integer. It can take on values ranging from -2,147,483,648 to 2,147,483,647. If these values are exceeded, the int will overflow; this is undefined behavior.
ints can be assigned an initial value when they are declared, or can be assigned one at a later time. The value of an int is undefined until it is assigned a value.

```
int a;
int a = 2;
```

1.1.2 unsigned int

An unsigned int is the unsigned equivalent of an int. Values can range from 0 to 4,294,967,295. The same rules from an int regarding assignment and overflow apply to an unsigned int.

1.1.3 char

A char represents a 1-byte ASCII character. chars follow the naming convention delineated previously. Additionally, the value of the char is generally given in single quotes. However, a positive number can be assigned to a char provided that it is between 0 and 255.

```
char a;
a = 'a';
char b = 97;
a == b;    // true
```

1.1.4 void

void is the only basic data type that can't be assigned to. void exists solely to indicate when a function will return nothing

1.2 Cryptographic Types

1.2.1 gem

A gem is the foundation of this crystalline language. Values range from 0 to the modulus-1. It must be initialized with an immutable modulus. Numbers larger than the modulus will be treated as that number modulus the initial modulo. For example,

```
gem g = new gem(7);
g = 10;
printf(g); // returns 3
gem h = new gem(4)
h = 10
printf(g+h); // returns 5
g = h*g
printf(g); // returns 6
h = h+g
printf(h); // returns 0
```

1.2.2 lattice

A big integer apt for handling the manipulation of large prime numbers, which is essential for the current cryptographic system, where it banks on the product of large prime numbers computationally infeasible to determine.

1.3 Grouping

Cryptal supports both pointers and arrays for grouping similar data values similar to how they are used in C.

1.3.1 pointer

Pointers represent a memory address using 8 bytes. Pointers are generally referred to by the type of the data that the programmer would like them to reference. They can be incremented or decremented to move them to the next or previous block of memory (blocks are grouped based on the pointer's type). Dereferencing a pointer (*) will return the value at the memory address it points to. Similarly, referencing a variable (&) will return a pointer to it.

```
char a = 'a';
char *foo = &a; //foo now points to a
*foo = 'A' //a now stores 'A'
```

1.3.2 array

Arrays are used to group together multiple values of the same type. An array is declared by appending square brackets to the type in a declaration. The user should declare a size for the array by appending square brackets to the end of a declaration, with a number inside the brackets indicating the size.

2 Operators

2.1 Unary Operators

These work as in C

- * expression: Dereferencing a pointer. This returns the object which the pointer points to. (See Data Types: Basic Data Types: pointer for more information).
- & expression: Returns a pointer to the memory address at which the object is stored.
- ! expression: Logical negation. Returns 0 if expression is non-zero, non-null and 1 otherwise.

2.2 Mathematical Operators

These work similar to C, with special behavior in a modular environment.

- *expression ˆexpression: The **exponential operator** represents an exponential operation. It requires that each expression be an integer or a gem. If the left expression is a gem, the result will be a gem. In all other cases, the result will be an integer. If both the left expression and the right expression are gems, the mod value of the left expression will be kept.
- expression * expression: The **multiplication operator** performs multiplication between the left and right expressions, where expression must be of type int or gem. If both the left expression and the right expression are gem, but they do not have the same mod value the result will be an integer. If only the left expression is a gem, the result will be a gem. In all other cases the result will be an integer.
- expression / expression: The **division operator** performs division between the left and right expressions, where expression must be of type int.
- expression + expression: The **addition operator** performs the addition between both expressions, which must of type int or gem. If both the left expression and the right expression are gem, AND they have the same mod value, the result will be a gem of that value. In this case, it will first calculate the integer addition, and then cast as a gem. In all other cases the result will be an integer.
- expression - expression: The **subtraction operator** performs the subtraction between both expressions, which must of type int or gem. If both the left expression and the right expression are gem, AND they have the same mod value, the result will be a gem of that value. In this case, it will first calculate the gem of the right expression and then perform the subtraction. In all other cases the result will be an integer.

2.3 Relational Operators

These work as in C

- expression > expression
- expression < expression
- expression >= expression
- expression <= expression
- expression == expression
- expression ! = expression

These perform comparisons between the left and right expression, where expression is an int, gem, or pointer. If expression is an int or a gem, the numerical value is used as a comparison. Otherwise, the value stored at the memory location to which the expression evaluates to is used. These operators return 0 or 1, depending on the result of the comparison.

```
int a = 5;
modint b = a mod 3;        /* 2 */
modint c = 6 mod 4;        /* 2 */
int *p = &b;

a > b;             /* 0 */
b == c;            /* 1 */
*p == b;           /* 1 */
```

2.4 Logical Operators

- expression && expression: Evaluates to 1 if both expressions evaluate to 1 and 0 otherwise.
- expression || expression: Evaluates to 1 if one or both expressions evaluate to 1 and 0 otherwise.

2.5 Assignment Operators

- lvalue = expression: The expression is evaluated and stored in the lvalue.

3 Keywords

- expression mod expression = The keyword mod takes the left expression, which can be of type int or gem, and creates a gem value with modular value equal to the right expression, which must only be of type int. The result is a gem with value left expression (mod right expression).

3.1 Control Flow

Here we specify the order in which computation is performed.

3.2 Statements and Blocks

Any expression can become a statement if it is followed by a semicolon. In other words, the semicolon is a statement terminator. Examples are:

```
score = 100;
grade = "A+";
```

Additionally, curly braces ... define compound statements or **blocks**. The latter are groups of declarations and statements, which will be syntactically treated as a single statement. We will see their use bellow in the fundamental control flow schemas.

3.3 If-Else

To execute statements conditionally, we use the keywords, **if** and **else**. Their syntax follows the rule:

```
if (expression)
        statement1
else
        statement2
```

The *else* part of the syntax is optional. First the expression is evaluated. If it is true, meaning that it evaluates to a non-zero value, then statement1 is executed. Otherwise, if the expression evaluates to zero (false) and an else clause is defined, statement2 is executed. If-Else statements can nest, and because else is optional, an else part is associated with the closest previous else-less if, just like in C. To associate an else to an if further away, blocks can be used, as in the example below:

```
if (x == y) {
        if (x > 0)
                a = 0;
}
else
        a = 1;
```

3.4 While Loop

The syntax of a while loop is:

```
while (expression)
        statement
```

First the *expression* is evaluated and if it is true the *statement* is executed. Then we repeat until the *expression* evaluates to false (0).

3.5 For Loop

The syntax for a for statement is:

```
for (expression1; expression2; expression3)
        statement
```

Which is equivalent to:

```
{
        expression1;
        while (expression2) {
                statement
                expression3;
        }
}
```

The equivalence is not exact for the use of the continue statement, which we will see later. Any of the expression1, expression2 or expression3 may be missing but not the semicolons. Last, the **for** loop:

```
for (;;)
```

is equivalent to the while loop:

```
while (true)
```

3.6 Break and Continue

A *break* statement, which can be included to the body of a loop, exits the loop early. Also, the rest of the body after the break statement will not be executed.
A *continue* statement will skip the rest of the statements of the body of the current iteration. Note, that for for loops specifically, expression3 will still be evaluated before the next iteration of the loop.

3.7 Mod Blocks

A mod block is a convenience mechanism the defines the modulo of the variables of type gem that are declared within that block. It has the following syntax:

```
mod (integer_expression)
        statement
```

The integer_expression, as its name hints, has to evaluate to some integer, which will be the modulo base of any gem declared in statement. For example, if we have the following piece of Cryptal code:

```
int x = 5;
int comparison
mod (x) {
        gem a = 4;
        gem b = 6;
        comparison = b > a;
}
...
```

after the *mod* block, comparison will evaluate to false (0), because in mod 5, a will be 4, but b will be 1.

4 Program Structure

A Cryptal program had global variables and functions that can communicate to each other through those global variables as well as parameters passed to them and/or values returned by them.

The only function that is required in a Cryptal program is the **main** function that returns an int. This is the entry point of the program. Any function called from within the *main* will need to be declared before main, but can be defined after. All other functions may have any other of our data types as a return type. Functions that return void will simply return a 0.

4.1 Examples

```
int main(arg1, arg2) { .... } // returns int
gem func(arg1, arg2) { .... } //returns gem
void func(arg1, arg2) { .... } // returns void
```

Using our construct of modular blocks must fall within the scope of a declared and defined function prior to main or within the scope of main itself.
Incorrect:

```
mod (5) {
        // some modular arithmetic
}

int main(arg1, arg2) {
        /* ...... */
}
```

Correct:

```
int main(arg1, arg2) {
        mod (5) {
        // some modular arithmetic
        }
}
```

Additionally, by default modular blocks return 0. But they can be made to return additional information using the keyword *return*. Furthermore modular blocks can also be nested to create allow for further modular arithmetic. Nested blocks can pass information by using the return keyword in each of their scope

```
int main(arg1, arg2) {
        mod (5) {
                // mod 5
                mod(3) {
                        //mod 3
                        gem a = 4;
                        gem b = 6;
                        return b > a;
                }
        }
}
```

Other than mod blocks we will also support a simple shorthand way to run hashing functions on arbitrary data. Using our native hash function call. For instance:

```
Hash.256(data); // SHA 256
Hash.DiffieHellman(data) // Diffie Hellman
```

This concept could be further extrapolated to having blocks of code run certain hashing algorithms. The structure of these blocks would follow in line with our existing concept of modular blocks.

5 File I/O

Through its IO library, cryptal supports input and output through 3 streams. Those are standard input (stdin), standard output (stdout) and standard error (stderr). The library includes 2 functions; one for reading and one for writing respectively:

```
int read (char *buf, int size)
int write (intf sn, char *buf, int size)
```

where:

- *sn* is the stream number corresponding to the stream number (2 for stdout and 3 for stderr)
- *buf* pointer to the data buffer
- *size* is the size of the meaningful data to be read/written.

Both functions return the number of bytes actually read/written, or -1 on error.