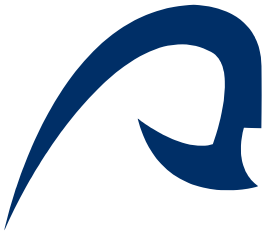

Universidad de Las Palmas de Gran Canaria

Procesadores de Lenguajes

Lenguaje Romaji

por Diego Sáinz de Medrano

27 de marzo de 2017

Índice

1. Introducción

El lenguaje Romaji está pensado para ser una suerte lenguaje C modificado y traducido al japonés. En su primer diseño, que es presentado en este documento, se quiere maximizar la utilidad del lenguaje al tiempo que se optimiza para la compilación en código Q.

2. Tipos de datos

La lista de los tipos definidos en el lenguaje Romaji, su tipo equivalente en formato similar a C++ y el tamaño en bytes que ocupa un literal de dicho tipo en memoria.

seisu	int	2
naga seisu	long int	4
nashi seisu	unsigned int	2
baito	char	1
mojiretsu	string	1+
furotingu	float	4
daburu	double	8

En el lenguaje Romaji se pueden declarar variables de todos estos tipos. A continuación se especifican las asignaciones permitidas sobre cada uno de ellos.

	Asignación
V	`seisu i <- 32767 (-32767)`
V	`seisu i <- 0xFFFF (0x0000)`
X	`seisu i <- 32768 (-32768)`
X	`seisu i <- 0xFEFEF`
X	`seisu i <- 0xAAA`
X	`seisu i <- i`
X	`seisu i <- 'i'`

	Asignación
V	`naga seisu l <- 65535 (-65535)`
V	`naga seisu l <- 0xFFFFFFFF (0x00000000)`
X	`naga seisu l <- 65536 (-65536)`
X	`naga seisu l <- 0xFFFFFFFFF`
X	`naga seisu l <- 0xFFFFFFF`

	Asignación
V	`nashi seisu u <- 65535 (0)`
V	`nashi seisu u <- 0xFFFF (0x0000)`
X	`nashi seisu u <- 65536 (-1)`
X	`nashi seisu u <- 0xFFFFF`
X	`nashi seisu u <- 0xFFF`

	Asignación
V	`baito b <- 255 (0)`
V	`baito b <- 0xFF (0x00)`
V	`baito b <- 'i'`
X	`baito b <- 256 (-1)`
X	`baito b <- 0xFEF`
X	`baito b <- 0xA`
X	`baito b <- i`
X	`baito b <- ''i''`

	Asignación
V	`furotingu f <- 1.0 (-1.0)`
V	`furotingu f <- 1 (-1)`
X	`furotingu f <- 0xFFFF`
X	`furotingu f <- 'a'`
X	`furotingu f <- ''a''`

	Asignación
V	`daburu d <- 1.0 (-1.0)`
V	`daburu d <- 1 (-1)`
X	`daburu d <- 0xFFFFFFFF`
X	`daburu d <- 'a'`
X	`daburu d <- ''a''`

	Asignación
V	`mojiretsu i <- ‘‘A string composed of ASCII characters.’’`
V	`mojiretsu i <- ‘‘’’`
X	`mojiretsu i <- 1234`
X	`mojiretsu i <- string_without_quotations`
X	`mojiretsu i <- 'string_with_single_quotations'`

Dentro de las ristras (`mojiretsu`) se aceptan caracteres ASCII, con la excepción del caracter nulo (`'0'` o NULL en la tabla ASCII). La aparición del caracter nulo señala el final de la cadena, como en C, por lo que no debe incluirse en la cadena. Se permite el uso de los códigos de escape especificados para C (ver referencia).

Además, están definidos dos tipos “especiales”, en el sentido de que no están pensados para ser declarados ni que se les asigne ningún valor, sino que se utilizan como medios de control para los retornos de las funciones y las operaciones lógicas.

shinri	boolean
kyo	void

El tipo `shinri` tiene dos valores, `shin` (equivalente a verdadero) y `nise` (equivalente a falso), mientras que el tipo `kyo` no toma ningún valor, sirviendo para especificar que las funciones no necesitan retornar nada al finalizar.

3. Estructuras básicas

3.1. Operaciones aritméticas

Las operaciones aritméticas en Romaji siguen el modelo de notación prefija.

```
seisu i <- + 1 2
```

En este sistema, las operaciones se describen primero con el operador, en este caso `+`, seguida de los literales, que se procesan en orden de aparición. Encadenar operaciones se realiza de la siguiente manera:

```
(1 + 2) * ((3 - 1) / 4)
```
se expresaría
```
* + 1 2 / - 3 1 4
```

Las operaciones aritméticas sólo pueden aplicarse a literales o variables de los tipos numéricos. Salvo casos especiales, las operaciones aritméticas deben recibir como argumentos tipos numéricos iguales y devolverán el mismo tipo numérico, o compatible*. El mismo operador es válido para cualquiera de los diferentes tipos.

* Los tipos compatibles quieren decir lo siguiente: existe una “jerarquía” de tipos, en el sentido de que unos tipos pueden ser contenidos en otros. Esto significa que si en una operación aparecen dos tipos compatibles diferentes, el resultado será del tipo “mayor”, haciendo el equivalente de un “casting” en C del tipo “menor” al “mayor” para realizar la operación. Ejemplo:

```
daburu d <- * 2 0.55
```

Sucederá que la multiplicación sea con los operandos 2.0 y 0.55, tratándolos ambos como `daburu`, y el resultado será de ese tipo también. La “jerarquía” mencionada sería la siguiente:

```
                / < naga seisu < daburu
baito < seisu {
                \ < nashi seisu < furotingu < daburu
```

Los operadores binarios definidos son:

- `+` suma.
- `-` resta.
- `*` multiplicación.
- `/` división.
- `%` módulo.
- `=` igualdad*.
- `<` mayor que*.
- `>` menor que*.
- `<= / >=` mayor/menor o igual que*.
- `|` o lógico**.
- `&` y lógico**.

*Los operadores de comparación devuelven un valor del tipo `shinri`.
**Estos operandos toman como operandos dos expresiones que devuelvan tipo `shinri`, ya sea un literal o una operación, y devuelven:

	`|`	`&`
shin	Una o las dos expresiones son ciertas	Las dos expresiones son ciertas
nise	Las dos expresiones son falsas	Una o las dos expresiones son falsas

Los unarios:

- `++` incremento.
- `--` decremento.
- `!` negación lógica*.

Estos operadores se utilizan de forma similar a los otros, escribiendo primero el operador y posteriormente el operando: `++ 1` . Estas operaciones pueden aplicarse a todos los datos numéricos (incluyendo los de coma flotante) y devuelven el valor del operando incrementado o decrementado en una unidad.

*El operando negación sólo toma como operando una expresión que devuelva tipo `shinri`, ya sea un literal o una operación, y devuelve el opuesto del resultado de la misma.

3.2. Estructura principal

Los programas escritos en Romaji siguen la siguiente estructura básica:

```
omo:[ret. type] <- [[type]:[arg1 name] [...] [type]:[argN name]]
{
    [code]
    [...]
    kisu [result] | shi
}
```

La palabra `omo` es el identificador de la función principal, el punto de entrada del programa. La palabra seguida después de los dos puntos es el tipo de dato que retorna el programa al terminar, y se permite cualquier tipo de dato exceptuando `shinri`. La flecha indica el comienzo de la declaración de los argumentos que recibe el programa al ser ejecutado (se especifica más en los argumentos en la sección 3.4).

El código (`[code]`) del programa debe estar rodeado de llaves, ya sean en la misma línea o en una nueva. En el código pueden aparecer alguna o ninguna de las siguientes expresiones, sin importar el orden:

- declaración de variable
- asignación de valor a una variable (ver sección 3.5)
- llamada a una función (ver sección 3.4)
- llamadas de entrada y salida (ver sección 3.3)
- bloques de código introducidos por instrucciones de control de flujo (ver sección 3.6)

Por último, el programa debe terminar con una de dos declaraciones: `shi`, en caso de que el tipo de retorno sea `kyo` o cuando quiera forzarse el término de la ejecución, o `kisu [result]` si se especifica algún tipo de retorno, siendo `result` una variable o un valor que pertenezca al tipo. Si bien estas llamadas pueden aparecer más de una vez en el código, por ejemplo, en un control de flujo, es requerido que aparezca al final para marcar el final del programa. (Nota: la función principal debe aparecer en el programa una única vez, siendo este el único requisito para que el programa sea válido, y aparecerá al final del programa, ya que como se verá más adelante, no tiene sentido definir funciones posteriormente al `omo`.)

En Romaji, las diferentes instrucciones deben estar separadas por un salto de línea, exceptuando los casos en los que se deba agrupar el código en bloques. Se requiere esto cuando

- declaramos el código de una función, por ejemplo el código de la función `omo`.
- el código se ejecuta dentro de una estructura de control, como un bucle o un condicional (ver sección 3.6).

En estos casos, las llaves pueden considerarse saltos de línea, es decir, se puede escribir el código previo a la llave y el posterior en la misma línea. Sin embargo, la misma norma se aplica en las diferentes instrucciones dentro del bloque.

Los comentarios se deben escribir en secciones ignoradas del código. Estas zonas se definen después de una almohadilla (#) hasta el final de la línea.

3.3. Entrada y salida

Existen dos funciones propias del lenguaje Romaji para la interacción, la de impresión y la de lectura, y se invocan utilizando las palabras clave `tsutaeru` y `uketoru` respectivamente.

El esquema para imprimir es el siguiente:

```
tsutaeru [var, string or number] [...]
```

Se imprimirán en orden los valores que se especifican entre paréntesis. Ejemplo:

```
omo:kyo <-
{
    seisu i <- 7
    furotingu f <- -2.67
    mojiretsu s <- "plus"
    tsutaeru (i s f "equals" + 7 f "\n")
    shi
}
```

La ejecución del programa producirá en la terminal de salida este resultado:

```
7 plus -2.67 equals 4.33
```

Nótese que al introducir un operador (+), empieza una expresión, que devuelve el resultado de la operación aritmética, contando como un único parámetro para la función `tsutaeru`.

El esquema para escanear es el siguiente:

```
[variable] <- uketoru
```

La ejecución del programa se detendrá al llegar a la instrucción que contiene la llamada a la función de lectura, y esperará una entrada desde el teclado, que terminará cuando llegue un retorno de carro.

Ejemplo:

```
omo:kyo <-
{
    tsutaeru ("Choose an integer:")
    seisu i <- uketoru
    tsutaeru ("You selected number " i "\n")
    shi
}
```

La función de lectura puede utilizarse para cualquier variable asignable, pero la entrada debe estar en concordancia con el tipo de la misma, es decir: lo que se introduzca debe ser una asignación válida, o dicho de otra manera, debe introducirse el dato en un formato igual al que se aceptaría al asignar ese mismo valor en el código (con la excepción del tipo `mojiretsu`, que no requiere que el texto esté rodeado de comillas).

Imprimiendo `shinri`

Una nota sobre la especialidad de la función `tsutaeru` es en el momento de imprimir un literal o el resultado de una expresión de tipo `shinri`. Se imprimirá "shin" en caso de que el valor sea verdadero o "nise" en caso contrario.

3.4. Funciones

Para declarar una función en Romaji se sigue este esquema:

```
kansu [func name]:[ret type] <- [arguments]
{
    [code]
    kisu [result]
}
```

La palabra clave `kansu` indica el comienzo de la declaración, marcando el nombre de la función con la cadena antes de los dos puntos. Como con la función principal, después se especifica el tipo de retorno y los argumentos.

Argumentos

Se especifican de la siguiente manera: separados por espacios, se escribe el tipo al que pertenece, dos puntos, y el nombre que tomará en la función. Los argumentos se pueden utilizar como variables dentro del código de la función, aplicandose todas las reglas para variables (sección 3.5). En Romaji, los argumentos que vienen dados como variables se pasan siempre por referencia, de forma que al realizar operaciones que los modifiquen, estamos modificando la zona de memoria donde están almacenados. El resto de argumentos, que pueden ser literales, expresiones o llamadas a funciones (se toma el resultado de las expresiones y el retorno de la función como argumentos), se pasan por copia.

La definición del código de la función viene integrada con la declaración, no pueden separarse ni redefinirse en el mismo código. Como con las variables, las funciones son accesibles para el código posterior a su declaración; es decir, que no se puede hacer una llamada a una función que esté declarada posteriormente a la llamada.

Para llamar a una función en el código seguimos el esquema de notación prefija:

```
[function name] ( [arg1] ... [argN] )
```

Deben pasarse los parámetros que la función reciba en orden, ya sea utilizando una variable o un valor literal, entre paréntesis. Las funciones pueden ser llamadas en asignaciones o en "solitario", siempre que estén dentro del flujo del programa, es decir, dentro del código de la función principal o de otra función.

Ejemplo de definición y llamada:

```
kansu multiply:seisu <- seisu:a seisu:b {
    kisu * a b
}

omo:kyo <- {
    seisu a <- 1
    seisu a_squared
    a_squared <- multiply (a a)
    shi
}
```

Ejemplo de como no se debe realizar una llamada:

```
kansu function:seisu <- seisu:a seisu:b { [...] }

function (1 2)    # nunca se pasa por esta linea

omo:kyo <- { [...] }
```

Nótese que una función con retorno puede ser llamada sin que este se capture en una asignación o se utilice como operando en alguna operación.

3.5. Variables

Para declarar una variable en Romaji se sigue este esquema:

```
[type] [var name] [<- initial value]
```

La declaración de una variable puede situarse en cualquier lugar del código fuente, pero su localización tiene impacto en el ámbito de la misma. Una variable declarada fuera de una función tiene ámbito global en el código sucesivo, mientras que una variable declarada dentro de un bloque de código solo tiene presencia en lo sucesivo del mismo.

Ejemplo:

```
seisu global_counter <- 0

omo:kyo <- {
    naga seisu local_counter
    local_counter <- ++ global_counter  # global_counter es accesible
}

kansu function:seisu <- seisu:a {
  local_counter <- a  # instruccion ilegal
}
```

Las variables aparecen en tres ocasiones: en la declaración, en asignaciones y cuando se quiere tomar su valor. En este último caso (que también puede ser al tiempo declaración), se pueden aplicar todas las operaciones a las variables de tipo numérico siempre que estas retornen el tipo que se requiere. Las variables de tipo `mojiretsu` no admiten operaciones aritméticas, solo de comparación, que se aplican al contenido de la cadena en el caso del operador `=` y a la longitud de la misma en el caso del resto de los operadores.

Normativa de nomenclatura

Tanto las variables como las funciones deben seguir unas normas en cuanto a su nombre. El nombre debe tener al menos un caracter de longitud, y ha de estar compuesto de caracteres alfanuméricos y/o "_", excluyendo cualquier otro símbolo, pero no pueden comenzar con un número. Además, no pueden ser iguales a ninguna palabra reservada.

3.6. Control de flujo

3.6.1. Condicionales

Para insertar un bloque de código que se quiera ejecutar en unas condiciones específicas, existen varias estructuras disponibles.

Si deseamos ejecutar un bloque de código bajo una única condición:

```
to [condition] {
    [code]
}
```

Donde `condition` es una expresión que de como resultado un valor booleano, ya sea una expresión de comparación o una llamada a una función con retorno booleano. En el programa se evaluará la condición, y de ser cierta, se ejecutará el bloque de código.

Para indicar que se debe ejecutar un bloque de código si una condición es cierta y otro distinto si no lo es:

```
to [condition] {
    [code if true]
}
ta {
    [code if false]
}
```

Recordatorio: las expresiones booleanas en Romaji se evalúan en `shin` o `nise`.

3.6.2. Bucles

Podemos escribir un bloque de código que se ejecute repetidamente si se cumple una condición.

```
naka [condition] {
    [code]
}
```

La condición se expresa igual que en las instrucciones condicionales. En el programa, se evaluará la condición antes de entrar en el bloque de código; de ser verdadera se ejecutará y en caso contrario saltará al final del bloque de código para continuar con la siguiente instrucción.