

Manual del lenguaje r3

r3 es un lenguaje de programación derivado del ColorForth.

<https://github.com/phreda4/r3>

2023

Vladimir Gomez Perez

Bianca Cipollone Di Croce

Pablo H. Reda

Programar una computadora

La computadora es un mecanismo que necesita una programación para funcionar, o sea, una descripción de lo que tiene que hacer. Esta descripción está en un lenguaje que entiende la máquina. Pero el proceso de programar comienza antes.

Primero nace de una idea, por ejemplo, dibujar un círculo. Luego vamos a escribir el código para que la máquina dibuje un círculo en un lenguaje determinado, por ejemplo

```
:draw 1 circle ;  
: draw ;
```

La computadora, primero traduce este código en otro lenguaje llamado código de máquina, que es el único que realmente puede entender. Este proceso se llama compilación.

Una vez que el programa escrito en el código se compila, el resultado se ejecuta en la computadora.

<<robot dibuja circulo>>

Si el código que escribimos no es posible compilar, entonces tiene un error en el código.

```
:draw 1 cicle ;  
: draw ;
```

Cuando tratamos de compilar la computadora nos dice que hay un error y no ejecuta nada.

cicle??

Otra posibilidad es que el programa esté bien escrito, pero no hace lo que queremos, que es dibujar un círculo.

```
:draw 1 box ;  
: draw ;
```

<<robot dibuja un cuadrado>>

La equivocación está en el programador, no en la computadora.

Para hacer esto necesitamos:

1. Tener un problema o una tarea a resolver.
2. Tener una idea de como resolverlo.
3. Traducir la idea al lenguaje de programación, hacer el programa o código.

4. Compilar el código sin errores.
5. Ejecutar el código y que haga lo que nos imaginamos

Para el punto 2: solamente podemos programar lo que sabemos cómo resolverlo o probar alguna solución que inventemos, no podemos programar lo que no sabemos cómo funciona. Esto se puede entrenar empezando con problemas fáciles, es un proceso creativo y de estudio.

Para el punto 3: necesitamos habilidad para traducir la idea al código, aunque puede requerir práctica y estudio, generalmente esto no es creativo, es solo una habilidad que se puede adquirir.

El punto 4 es la indicación que hicimos bien el punto 3.

El punto 5 es la indicación que hicimos bien el punto 2.

La elección del lenguaje de programación va a modificar el trabajo necesario para completar del punto 3 y el punto 4, aunque no tendrán implicancias en el punto 2.

Existen más de 3000 lenguajes de programación, r3 es uno de ellos.

Lenguaje r3

Introducción

Programar una computadora significa construir un mecanismo que va a producir un comportamiento en la máquina, es hacer la “receta” de este comportamiento, esta RECETA se llama CÓDIGO FUENTE o PROGRAMA..

El programa se compone de 2 tipo de definiciones:

- DATO, que podemos llamar también MEMORIA, ESTADO o VARIABLE
- CÓDIGO, que también llamamos ORDEN, RUTINA, FUNCIÓN o ACCIÓN

Como DATO necesitamos representar lugares en la memoria donde guardar números, que pueden representar o ser usados como:

- CANTIDAD, por ejemplo: 3 vidas
- DIRECCION o UBICACION, por ejemplo: lugar 100 de la pantalla
- ESTADO, por ejemplo: saltando o cayendo

Como CÓDIGO necesitamos construir ACCIONES.

Podemos construir cualquier comportamientos con los siguientes elementos

- SECUENCIA: una orden sigue con la siguiente.
- CONDICIÓN: solo si se cumple una condición se sigue una orden.
- REPETICIÓN: repetir una orden de alguna manera indicada.
- RECURSIÓN: definir una acción haciendo referencia a la misma acción, lo menos usado.

Por lo tanto, se definen dos cosas: códigos y datos, algo que funciona y algo que recuerda. Lo que funciona solo pasa a través del tiempo (una secuencia de pasos con un orden específico), lo que se recuerda se guarda en un espacio de memoria.

Un programa parte de una IDEA de mecanismo para hacer algo. Esta idea es generalmente una secuencia de cálculos y acciones que funcionan cuando el programa está escrito y es EJECUTADO por la computadora. Como toda actividad creativa, dentro de las reglas de programación, las posibilidades son inmensas, encontrar la idea y transformarla en mecanismo es lo que necesitamos practicar para programar.

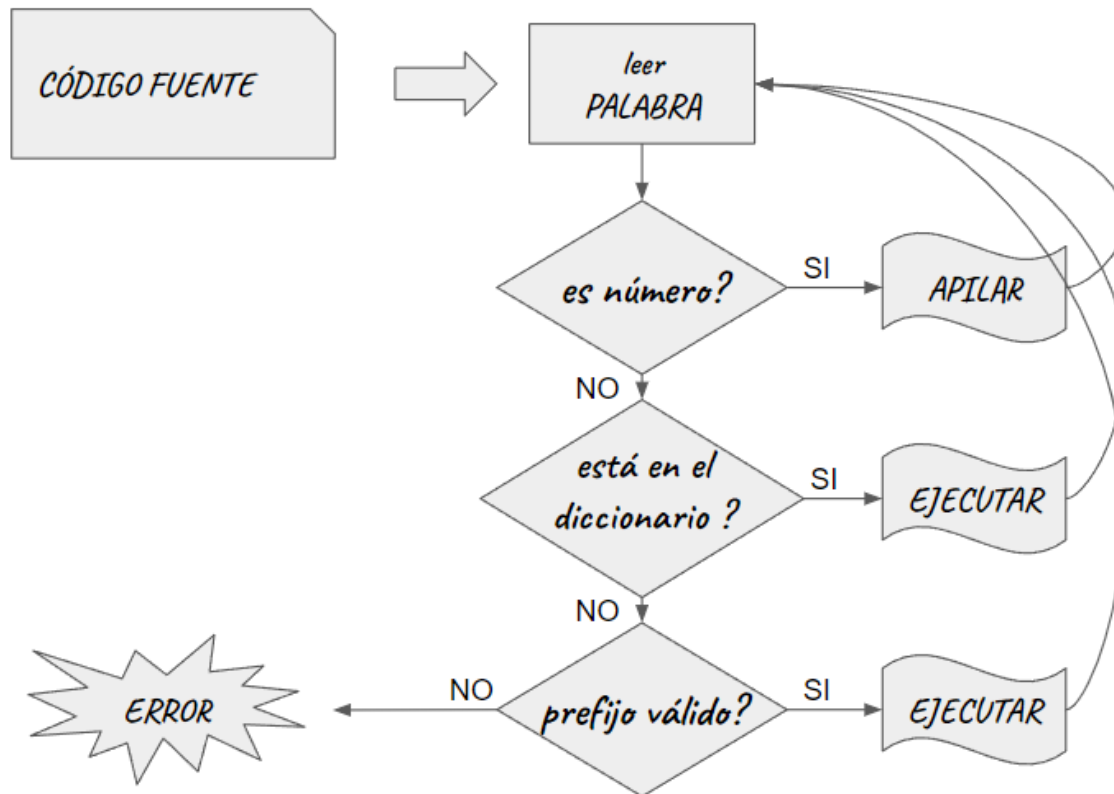
Conceptos de programación en r3

El programa es un archivo de texto, el CÓDIGO FUENTE, donde se separa en palabras, una PALABRA se define como una secuencia de letras, dígitos o caracteres, separadas por espacios, son palabras válidas:

VIDAS 134 -*saltar*- vidas

No se toma en cuenta si las letras son mayúsculas o minúsculas, por lo que VIDAS y vidas son iguales para el lenguaje.

Los espacios definen los límites de las palabras, muchas veces el error de un código es que falta un espacio. No se toma en cuenta la forma en que están separadas, es lo mismo si hay un espacio, varios espacios o está en la línea de abajo.



El código fuente se lee palabra por palabras

- Si la palabra es un número, se apila en la PILA DE DATOS, y se pasa a la siguiente palabra
- Si la palabra no es un número válido, se busca en el DICCIONARIO, si se encuentra se EJECUTA y se pasa a la siguiente palabra
- Si la palabra tiene alguno de los prefijos válidos, se interpreta según su significado y se pasa a la siguiente palabra
- Si no se encuentra dicha palabra se termina el programa indicando un **error** en ese lugar, se detiene el programa y se indica el error.

Se reconocen 8 prefijos en las palabras. Estos tienen un significado en el código que escribimos.

	Es un <u>comentario</u> , no se ejecuta, termina al final de la línea
^	<u>Incluye</u> el código del archivos indicado
“	Se define una cadena de <u>texto</u> , termina con ”
:	Define <u>acciones</u>
#	Define <u>datos</u>
\$	Define números <u>hexadecimales</u>
%	Define números <u>binarios</u>
'	Indica la <u>dirección</u> de una palabra, esta <u>dirección</u> se apila como un número

El prefijo dirección, necesita una palabra válida, de lo contrario es un **error**, las palabras del diccionario base no tienen dirección.

Diccionario

El lenguaje comienza con un diccionario ya definido, este diccionario tiene alrededor de 200 palabras que son las funciones básicas que realiza la computadora, puede verse en el Apéndice 1. A partir de aquí se definen nuevas palabras con los prefijos : y #, se agregan al diccionario, para ser usadas después.

Cuando el lenguaje busca una palabra en el diccionario, esta búsqueda se realiza desde la última a la primera, se pueden definir palabras con el mismo nombre pero solamente será ejecutada la última definida, después de esta definición.

El orden de búsqueda de las palabras hace que si se definen dos o más palabras con el mismo nombre, que está permitido, solamente se podrá invocar a la última definida.

Programar es definir nuevas palabras en este diccionario y por último llamar a la primera palabra que va a ejecutar todo el programa.

El prefijo de inclusión ^ va a tomar el archivo de texto indicado y va a agregar en el diccionario todas las palabras definidas en este texto que se marquen como exportadas, doble :: cuando es código y doble ## cuando es dato.

Las palabras que están en el diccionario base, no tienen dirección, el resto si. Esta dirección puede ser utilizada para guardar como dato una acción y ejecutarse después.

Programar es crear palabras

Programar es crear palabras que van a ser usadas por otras palabras hasta que se pueda describir el comportamiento que queramos programar.

Definimos acciones con el prefijo : o datos con el prefijo #, : solo es el inicio del programa, un programa completo en r3 puede ser el siguiente

1	#lado 5
2	:cuadrado dup * ;
3	
4	: lado cuadrado ;

La línea 1 define una variable con el valor 5.

La línea 2 define una palabra que duplica el tope de la pila y luego multiplica estos dos números.

La línea 4 es el lugar del comienzo del programa, apila una 5, que es el valor que contiene la variable lado, luego llama a cuadrado, aquí duplica este valor y luego multiplica los dos valores, retorna a la línea 4, donde había sido llamada y termina la ejecución del programa.

Al final del programa, la pila tendrá el número 25.

Cuando se incluye una biblioteca, con el prefijo ^, solamente irán al diccionario de quien la incluyó las palabras definidas como exportada, esto es, con ## o con ::, además si hay un comienzo de programa con :, estas palabras se ejecutarán al principio de la ejecución de quien la incluyó.

El punto y coma no indica que se terminó la definición de la palabra sino que termina la ejecución, por esto es posible que una palabra tenga varios punto de finalización e, inclusive, no que no tenga fin y continúe en la próxima palabra definida. Las palabras que son datos, es decir, las variables, están en un espacio de memoria distinto a las palabras que son código, por lo que la continuación antes mencionada nunca va a ocurrir con una variable definida después de un código sin punto y coma.

Diccionario base

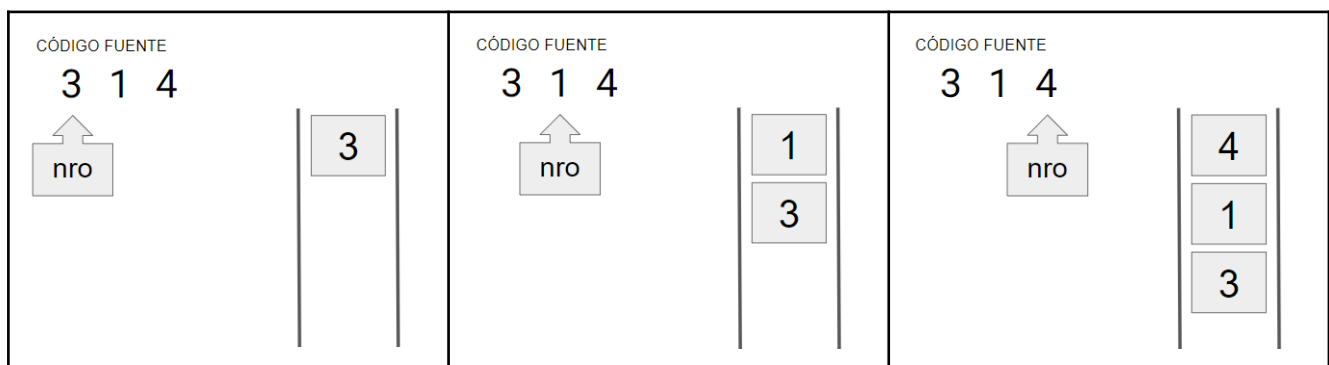
La palabra más utilizada va a ser el punto y coma, esta palabra no consume ningún valor de la pila y tampoco produce ningún valor.

; | --

Esta no consume ni produce valores en la pila pero como acción lateral, marca el fin de la ejecución de la palabra es decir termina una palabra, no su definición, sino su ejecución.

Pila de datos

La pila es un concepto dentro de la informática, para representar valores dentro de un orden y funcionamiento determinado, el primer número en entrar a la pila es el último en salir de ella. Cuando se encuentran números en el código fuente, estos se irán apilando a medida que se ejecuta el código.

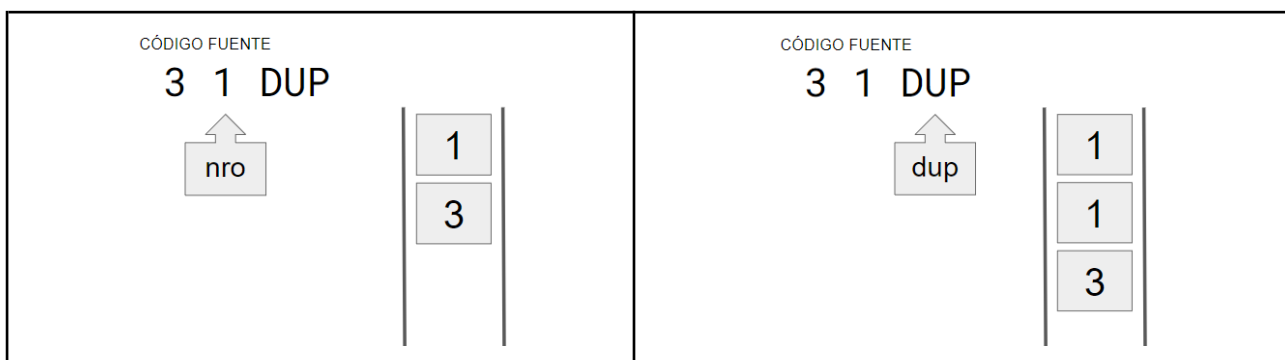


Este es un lugar donde se realizan todas las operaciones, aquí se almacenan los valores que van a ser usados para cálculos y para indicar parámetros o indicaciones a otras palabras.

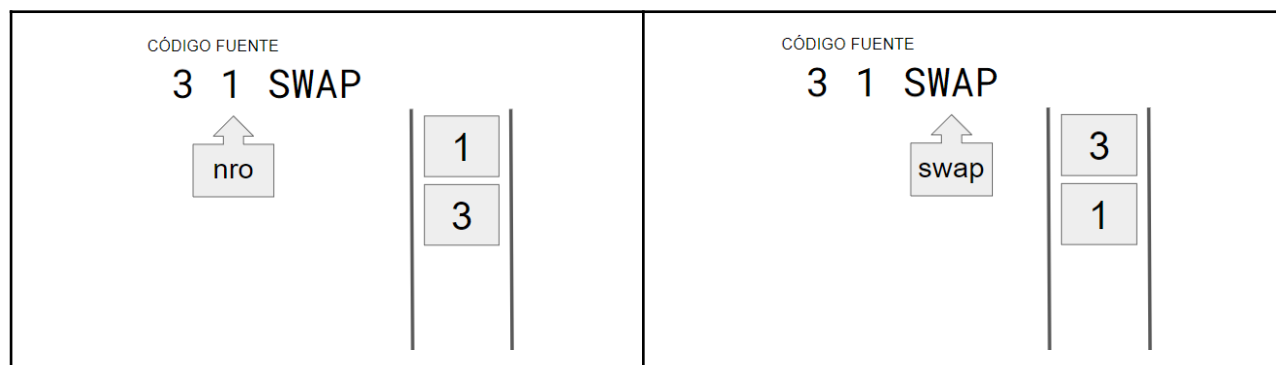
Cada palabra puede tomar y/o dejar valores de la pila de datos. Como ayuda al programador se pone un comentario (que no tiene incidencia en el código) con un diagrama de estado de la pila antes y después de la palabra, separados por 2 signos menos.

Tenemos algunas palabras para modificar el estado de la pila, además de agregar valores necesitamos copiarlos, borrarlos y cambiarlos de orden.

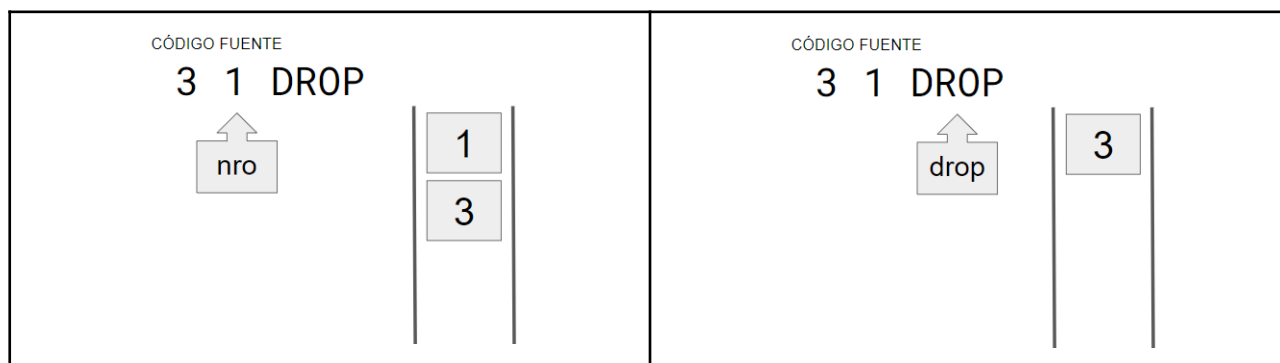
DUP | a -- a a



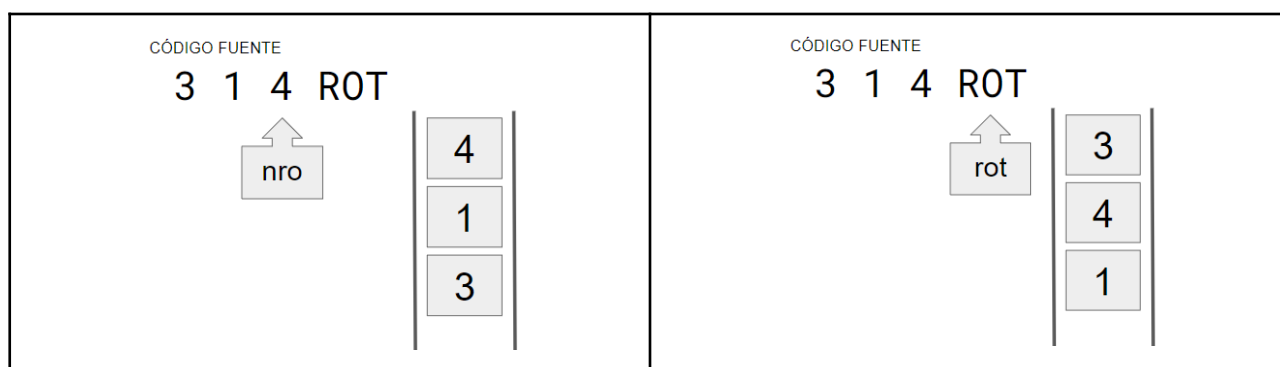
SWAP | a b -- b a



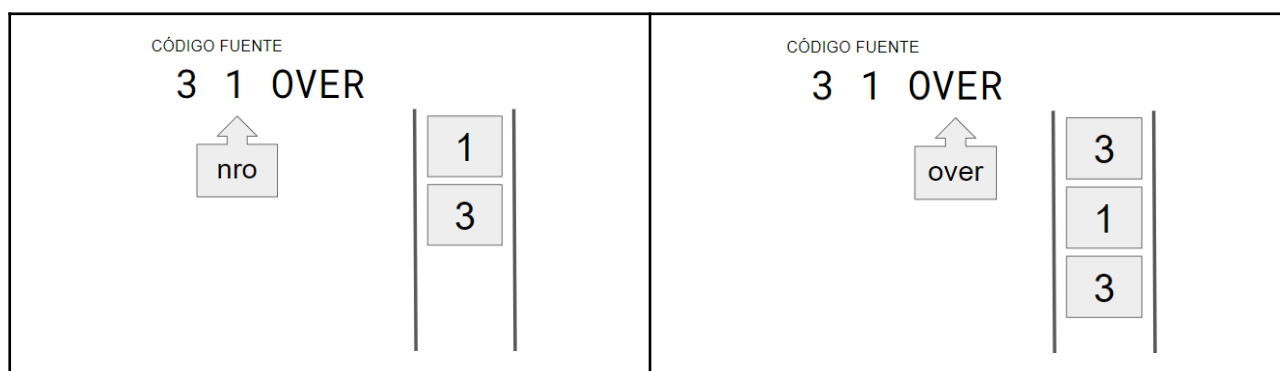
DROP | a --



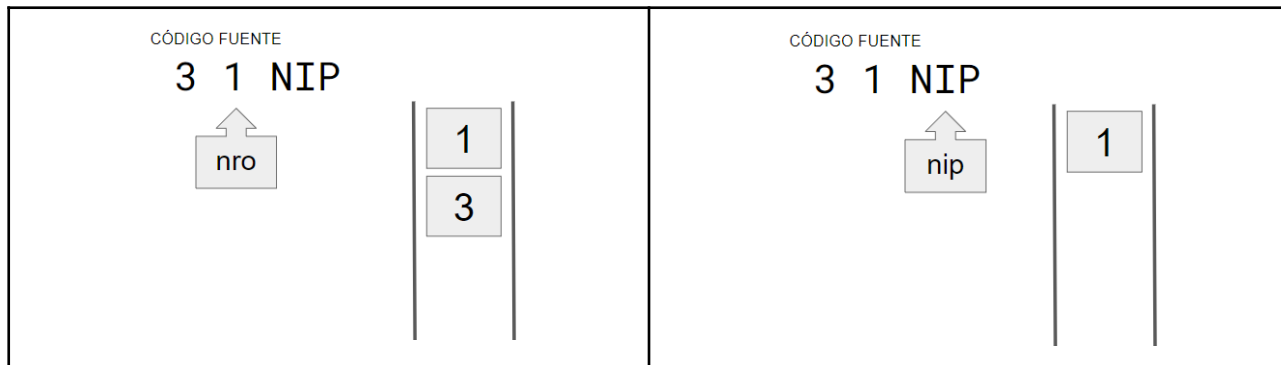
ROT | a b c -- b c a



OVER | a b -- a b a



NIP | a b -- b

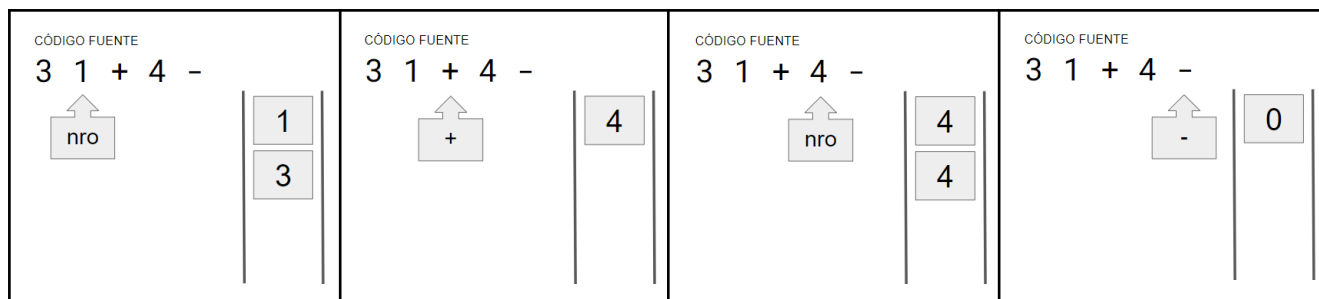


Podemos ver que algunas palabras PRODUCEN números, por ejemplo DUP producirá un número (duplicando el tope de la pila), otras palabras CONSUMEN números, por ejemplo DROP elimina el tope de la pila. También algunas palabras no afectan a la pila, o consumen y producen a la vez. Cuando se ejecuta el código la pila va pasando de un estado a otro, este comportamiento no se ve en el texto que escribimos y esta es una de las dificultades para entender cómo funciona. En el caso que nuestra definición haga crecer la pila indefinidamente o consuma todos los números que estén ahí el programa tiene un error, ya que se está construyendo un mecanismo que no tiene sentido. Esto no se puede detectar antes de ejecutarse y lo que ocurrirá es que el programa quedará congelado o se detendrá.

Como estamos trabajando con números, se usan comúnmente las operaciones aritméticas.

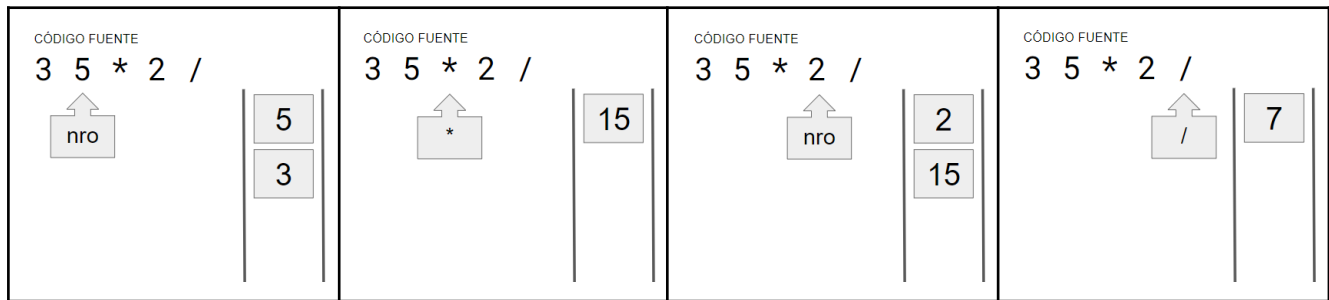
SUMA Y RESTA

+ | a b -- c c=a+b
 - | a b -- c c=a-b



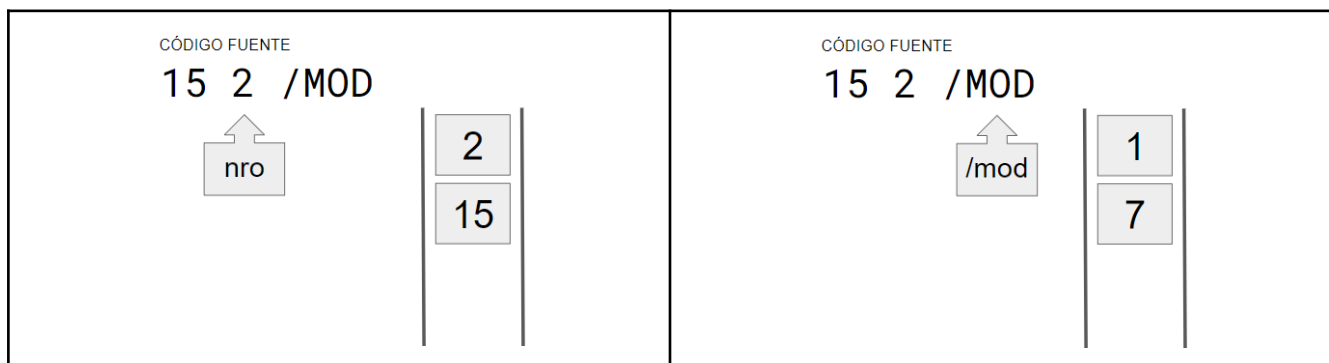
MULTIPLICACIÓN y DIVISIÓN

* | a b -- c c=a*b
 / | a b -- c c=a/b



DIVISIÓN y MÓDULO

MOD | a b -- c c=a mod b
 /MOD | a b -- c dc=a/b d=a mod b



DESPLAZAMIENTO DE BITS

<< | a b - c corrimiento de bits a la izquierda
 >> | a b - c corrimiento de bits a la derecha
 >>> | a b - c corrimiento de bits a la derecha sin arrastre de bit

	5 2 << apila 20 5 1 >> apila 2 -2 1 >> apila -1 -1 1 >>> apila 9223372036854775807 (en binario 01111...11 64bits)
--	--

OTRAS OPERACIONES

NEG | a -- -a negar valor
 ABS | a -- |a| valor absoluto
 SQRT | a -- b raiz cuadrada
 */ | a b c -- d d=a*b/c sin pérdida de bits

	3 NEG apila -3 -3 ABS apila 3 25 SQRT apila 5 314 2 100 */ apila 6
--	---

OPERACIONES LÓGICAS

```
AND    | a b - c
OR     | a b - c
XOR    | a b - c
NOT    | a - b
```

```
$ff $55 AND | Apila $55
$2 $1 OR | apila 3
$3 2 XOR | apila 1
0 NOT | apila -1
```

OPERACIONES DE PUNTO FIJO

Para operar con punto fijo el lenguaje reconoce los números con el punto decimal y los traduce a esa representación en el formato 48.16.

Con estos números la suma y la resta son iguales a los enteros, pero la multiplicación y la división necesita de palabras especiales definidas en `r3/lib/math.r3`, junto con otras palabras útiles como las funciones trigonométricas.

`^r3/lib/math.r3`

```
::*. | f f -- f    multiplica dos números de punto fijo
::*. | f f -- f    divide dos números de punto fijo

::int. | f -- a    convierte a entero

|--- funciones trigonométricas
| el ángulo está en vueltas (180 grados=0.5)

::cos | f -- cos(f)
::sin | f -- sin(f)
::tan | f -- tan(f)
::sqrt. | f -- sqrt(f)
::ln. | x -- r
::exp. | x -- r
::root. | base root -- r
```

OPERACIONES DE PUNTO FIJO (AVANZADO)

La definición de un algoritmo incluye qué posibles valores tendrán los datos que se usan: es posible definir algoritmos utilizando solamente números enteros o con números reales, es decir, con decimales.

Los números de punto flotante utilizan una representación en binario que posee algunos problemas, esto no se explica aquí.

Hay que tener presente que los números representados en la memoria difieren del concepto matemático, en la computadora los números se almacenan en determinada cantidad de bits, no son infinitos y tampoco son una recta, se puede ver mas como un anillo, el valor más alto está unido al valor menor, así, en 8 bits, sumarle 1 a 127 nos dará el valor -128.

Los valores de punto fijo son enteros que, tratados con ciertos mecanismos, sirven para representar partes de una unidad. Lo que se hace es definir un lugar donde el valor de bit será uno, para la izquierda serán 1, 2, 4, 8, etc y para derecha serán $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$, etc. Por ejemplo en números de punto fijo con 8 bits como parte entera y 8 bits como parte decimal, llamados 8.8, tenemos los siguientes valores de los bits.

SIG	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
-----	----	----	----	---	---	---	---	-----	-----	-----	------	------	------	-------	-------

Aquí el lenguaje reconoce los números en punto fijo cuando encuentra el punto decimal en un número y lo que hace es convertir el número en el código fuente en su representación binaria como números de punto fijo con formato 48.16 (ya que la celda de memoria es de 64 bits).

2.5 en binario en formato 8.8 es 0000001010000000
bits prendido en valor 2 y valor $\frac{1}{2}$.

La suma y la resta serán iguales en punto fijo y en enteros, pero la multiplicación y la división necesitan ajustar la posición del punto decimal que que, en el caso de la multiplicación se duplicarán la cantidad de lugares decimales y en la división será necesario duplicar estos lugares en el número a dividir, estas dos operaciones, necesitan una rango de bits mayor al que se está operando para que no se pierdan bits, para esta necesidad, r3 define 2 operaciones en el diccionario base que realizan estas operaciones.

```
*>> | a b c -- d d=(a*b)>>c sin pérdida de bits
<</ | a b c -- d d=(a<<c)/b sin pérdida de bits
```

Estas operaciones permiten la multiplicación y la división con cualquier distribución de bits, se definen palabras para manejar números en punto fijo en la biblioteca

Es posible convertir la representación de un número en punto flotante, desde un entero y desde y hacia un punto fijo, esto es útil cuando se quiere leer un archivo o memoria donde los valores están guardados en este formato.

	<pre> --- integer to floating point ::i2fp i -- fp --- fixed point to floating point ::f2fp f.p -- fp ::fp2f fp -- fixed point</pre>
--	--

Condiciones

Los paréntesis son usados para indicar bloques de código, tener en cuenta que son palabras también y deben estar separados por espacios.

(bloque de código)

Junto con los bloques de código, tenemos un grupo de palabras que hacen las comparaciones y así poder construir los condicionales y las repeticiones.

Tenemos condicionales que solamente comprueban el tope de pila, sin modificarlo.

```
0? | a -- a ; a no se modifica, es verdadero cuando a=0
1? | a -- a ; a distinto de 0 ?
-? | a -- a ; a es negativo ?
+? | a -- a ; a es positivo ?
```

También podemos comparar el tope de la pila, llamado TOS (top of stack), con el segundo, llamado NOS (next of stack), consumiendo el TOS, o sea, la comparación quitara el tope de la pila.

```
=? | a b -- a ; a=b?
<? | a b -- a ; a<b ? menor
<=? | a b -- a ; a<=b ? menor o igual
>? | a b -- a ; a>b ? mayor
>=? | a b -- a ; a>=b ? mayor o igual
<>? | a b -- a ; a distinto de b ?
```

La condición se construye con una palabra que indica cual es la condición y a continuación un bloque de código que se va a ejecutar solamente si esta condición se cumple.

<p>CÓDIGO FUENTE</p> <p>3 4 <? (NEG) DUP</p> <p>↑</p> <p>nro</p> <p>4</p> <p>3</p>	<p>CÓDIGO FUENTE</p> <p>3 4 <? (NEG) DUP</p> <p>↑</p> <p><?</p> <p>VERDADERO !!! entra al bloque</p> <p>3</p>	<p>CÓDIGO FUENTE</p> <p>3 4 <? (NEG) DUP</p> <p>↑</p> <p>neg</p> <p>-3</p>	<p>CÓDIGO FUENTE</p> <p>3 4 <? (NEG) DUP</p> <p>↑</p> <p>dup</p> <p>-3</p> <p>-3</p>
<p>CÓDIGO FUENTE</p> <p>3 4 >? (NEG) DUP</p> <p>↑</p> <p>nro</p> <p>4</p> <p>3</p>	<p>CÓDIGO FUENTE</p> <p>3 4 >? (NEG) DUP</p> <p>↑</p> <p>>?</p> <p>FALSO !!! pasa el bloque</p> <p>3</p>		<p>CÓDIGO FUENTE</p> <p>3 4 >? (NEG) DUP</p> <p>↑</p> <p>dup</p> <p>3</p> <p>3</p>

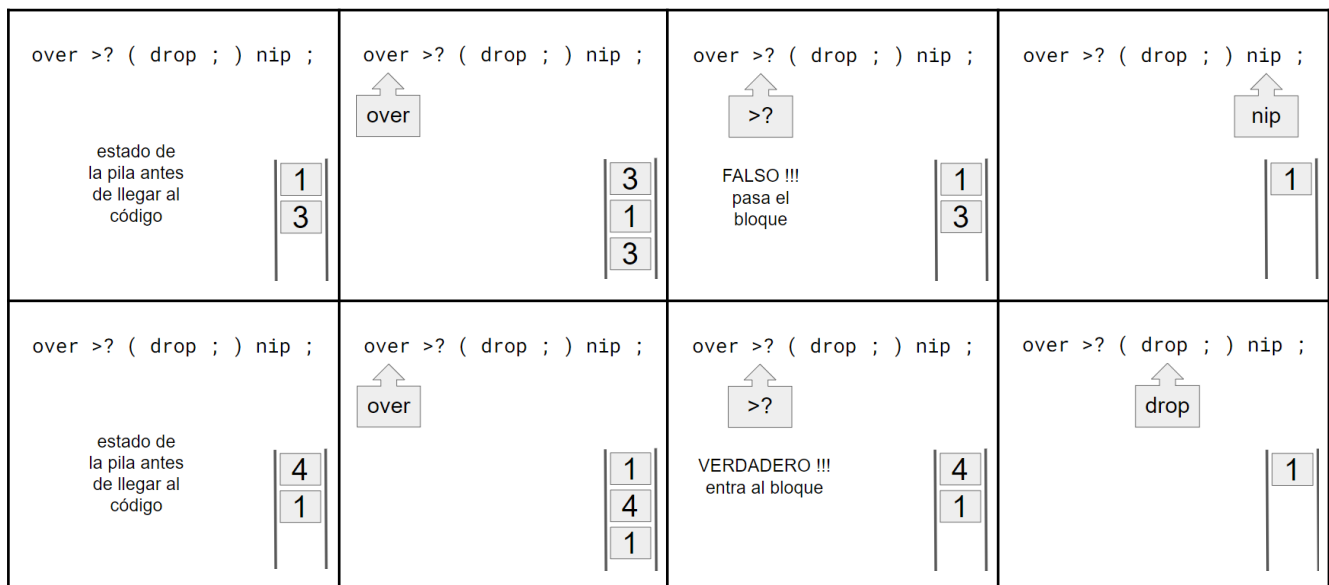
El segundo de la pila no se consume y por esto se pueden encadenar preguntas para un determinado valor.

1	3
2	4 >? ("Mayor que 4" .print)
3	4 <? ("Menor que 4" .print)
4	drop

La línea 2 pregunta si en el tope de la pila hay un número mayor que 4, y si es así imprime el texto.

Se puede terminar la ejecución de una palabra dentro de un bloque de código, esto es, la ejecución de la palabra termina cuando encuentra ; , podemos definir la palabra min para obtener el mínimo de 2 números.

```
:min | a b -- c
      over >? ( drop ; ) nip ;
```



Una construcción que no está disponible en r3, y es la llamada IF-ELSE, esto tiene una explicación y es que la ausencia fomenta la factorización, esto es, obliga a separar la lógica o encontrar una forma de resolver el algoritmo de otra manera.

Las secuencia:

A ?? (B) else (C) D.

Según se verdadera o falsa la condición producirá:

A B D o A C D.

Para replicar en r3, es necesario definir otra palabra donde trasladar la condición.

```
:condicion ?? ( B ; ) C ;
```

A condicion D

El resultado es el mismo, la nueva palabra creada tendrá la función de elegir entre una opción o la otra.

Otra de las construcciones no disponible es la llamada SWITCH-CASE, que consiste en una secuencia de comparaciones que ejecutan diferentes partes del código, aquí tenemos dos alternativas para construirlo, si las comparaciones son con enteros en secuencia o si la comparación es con valores muy distantes o intervalos.

En el caso que las comparaciones pueden traducirse a un conjunto de enteros en secuencia, es posible hacer una tabla de saltos donde ese número se va a traducir en la dirección de una palabra que se ejecuta.

```
:a0 "accion 0" ;
:a1 "accion 1" ;
:a2 "accion 2" ;
:a3 "accion 3" ;
```

```

:a4 "accion 4" ;

#lista 'a0 'a1 'a2 'a3 'a4

:accion | n -- string
  3 << | 8 * tamaño de la celda
  'lista + @ ex ;

```

En el caso de que las comparaciones no se pueden traducir en una secuencia de enteros, lo mejor es hacer una palabra que compare y termine con todas las opciones posibles.

```

:casos | valor -- valor string
  5 <? ( "menor a 5" ; )
  6 =? ( "es 6" ; )
  7 =? ( "es 7" ; )
  111 <? ( "entre 8 y 110" ; )
  "mayor o igual a 111" ;

```

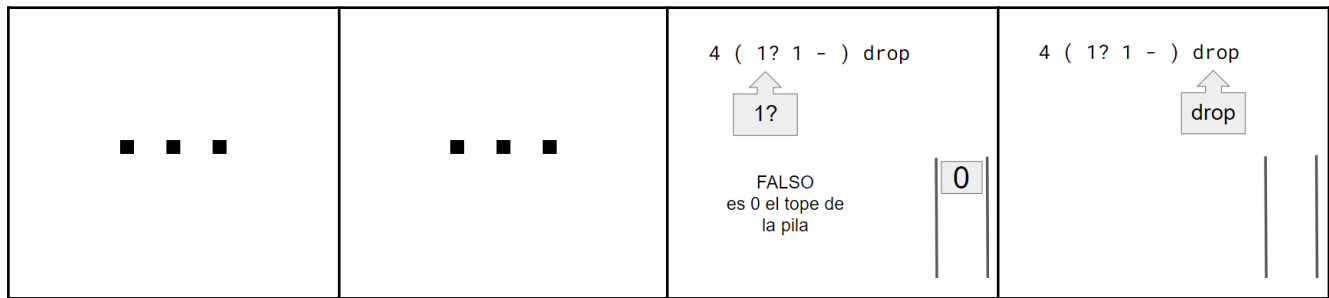
Esta definición tiene un final por cada rama de las comparaciones, aquí hay que tener cuidado que todos los finales de palabras produzcan el mismo comportamiento de la pila, salvo que se desee lo contrario.

Repetición

Cuando el condicional está dentro del bloque se construye una repetición.

Mientras se cumpla esta condición el bloque se repetirá, cuando esta condición sea falsa, la ejecución saltará a la siguiente palabra del final del bloque.

<div>4 (1? 1 -) drop</div> <div>↑ nro</div> <div>4</div>	<div>4 (1? 1 -) drop</div> <div>↑ 1?</div> <div>VERDADERO No es 0 el tope de la pila</div> <div>4</div>	<div>4 (1? 1 -) drop</div> <div>↑ nro</div> <div>1 4</div>	<div>4 (1? 1 -) drop</div> <div>↑ -</div> <div>3</div>
<div>4 (1? 1 -) drop</div> <div>↑ 1?</div> <div>VERDADERO No es 0 el tope de la pila</div> <div>3</div>	<div>4 (1? 1 -) drop</div> <div>↑ nro</div> <div>1 3</div>	<div>4 (1? 1 -) drop</div> <div>↑ -</div> <div>2</div>	<div>4 (1? 1 -) drop</div> <div>↑ 1?</div> <div>VERDADERO No es 0 el tope de la pila</div> <div>2</div>



podemos utilizar otros condicionales para contar de forma ascendente.

1	1 (10 <?
2	dup "%d " .print
3	1 +) drop

Este código imprime los números del 1 a 9 ya que cuando el tope de la pila se convierte en 10 en la línea 2 salta a la línea 3 después de)

Imprimir una tabla de 10 x 10, con dos bucles anidados:

1	#tabla * 800 10 x 10 x 8 (8 bytes por número)
2	
3	'tabla
4	0 (10 <? 1 +
5	0 (10 <? 1 +
6	rot @+ "%d " .print
7	rot rot
8) drop
9	.cr
10) drop

Los condicionales que no consumen la pila se van a ejecutar más rápido, simplemente por no tener el número que compara, debido a esto el valor 0 se convierte en la marca de finalización preferida para este comportamiento, y, la cuenta regresiva la preferencia.

```
10 ( 1? 1 - ) drop
```

Si necesitamos o no el 0 dentro del bucle, pondremos nuestro código antes o después de la resta.

Asimismo cuando estamos recorriendo memoria, en cualquier tamaño de bytes, es muy práctico tener una marca de finalización, y generalmente esta marca es 0. Esto hace que no haga falta registrar la cantidad de unidades de memoria en otro lugar y, cuando se recorre no hace falta tener este doble registro, la cantidad y la dirección donde se están poniendo o sacando valores. El más común es en el manejo de texto, donde la finalización es el byte 0

```
"hola" ( c@+ 1?
  usar_cada_caracter ) 2drop
```

Una de las cosas a tener cuidado es que cuando salimos del bucle, además de la dirección, tenemos el 0 apilado.

Usar una cantidad necesaria

```
"hola" 4 ( 1? 1- swap c@+  
  usar_cada_caracter swap ) 2drop
```

Es válido realizar varias comparaciones para salir del bucle, así para terminar un bucle si el valor es 0 o 13, podemos construir el siguiente código

```
( c@+ 1?  
  13 <>?  
  drop ) | aqui hay un 0 o un 13
```

Aquí necesitamos que cada salida del bucle tenga el mismo comportamiento en la pila.

Recursión

La recursión se construye llamando a la palabra que se está definiendo.

Por ejemplo, en la sucesión de fibonacci, esta se construye comenzando con un 1 en la posición 0 y 1, el siguiente término es la suma de los 2 anteriores y así sucesivamente.

1	:fibonacci n -- f
2	2 <? (1 nip ;)
3	1 - drop
4	1 - fibonacci swap fibonacci + ;

En el momento que se empieza a definir una palabra ya es posible invocar, entonces la recursión ocurre naturalmente.

Como toda definición recursiva se debe tener especial cuidado en la condición de corte y el estado de la pila cuando se termina la palabra.

Cuando tenemos una palabra antes de un ; (fin de palabra), el lenguaje internamente no va a retornar de esta palabra para terminar, sino que va terminar en la palabra llamada, esto se lo llama tail call o cortar la llamada, esto produce en la recursión que, si la ultima palabra se llama a sí misma, va a transformarse en un bucle.

```
:loopback | n -- 0  
  0? ( ; )  
  1 -  
  loopback ;
```

Variables y memoria

Las variables definen memoria para usar como almacenamiento de datos, las variables tienen un nombre, una dirección en memoria y un valor que se encuentra en esa memoria. La dirección real donde se encuentra cada variable va a obtenerse cuando se ejecute, no es necesario conocer el valor de esta dirección, sino usar su nombre para representarlo.

1	#vidas 3
2	#posicionX #posicionY

3	#mapa * \$400 1KB
4	#lista 3 1 4
5	#energia 1000
6	
7	vidas 3
8	1 + 4
9	'vidas 4 \$1000
10	!

Como ejemplo, pondremos la dirección donde se encuentran comienzan las variables en \$1000, usando números en hexadecimal, este código se refleja en la memoria de la siguiente manera

	Codigo	Nombre	dir: \$hex (dec)	valor en mem
1	#vidas 3	vidas	\$1000 (4096)	3
2	#posicionX #posicionY	posicionx	\$1008 (4104)	0
		posiciony	\$1010 (4112)	0
3	#mapa * 100	mapa	\$1018 (4120)	0 0 0 ... 0
4	#lista 3 1 4	lista	\$1418 (5144)	3 1 4
5	#energia 1000	energia	\$1430 (5168)	1000

Para cambiar el valor en una variable utilizamos la palabra:

! | valor direccion --

llamada aquí STORE (Almacenar) o escribir en memoria. Esta palabra guarda en la dirección el valor indicado

Usando el nombre de una variable apilamos el valor que contiene esa dirección de memoria, sin embargo, cuando operamos sobre esa dirección es necesario obtener este valor para una dirección en pila, por esto necesitamos una palabra que lo obtenga:

@ | direccion -- valor

llamada aquí FETCH (Buscar/obtener) o leer de memoria. Esta palabra obtiene de la dirección el valor que se encuentra en este lugar.

Hay que tener en cuenta que cada número en memoria se almacena en un espacio de 8 bytes (64 bits) y por lo tanto si hay una secuencia de números, estos estarán a esta distancia.

Tenemos algunas palabras para realizar tareas comunes como incrementar el valor de una variable, por ejemplo:

+! | val direccion -

Guardar un valor en una dirección e incrementar la direccion

```
!+ | valor direccion -- direccion+8
```

Leer un valor de una dirección e incrementar la direccion

```
@+ | direccion -- direccion+8 valor
```

1	:listashow
2	'lista
3	@+ "%d " .print imprime 3
4	@+ "%d " .print imprime 1
5	drop ;
6	
7	'lista 8 + @ apila 1
8	
9	1 'posicionX +! suma 1 a posicionx
10	
11	listashow imprime 3 1
12	5 6 5 6
13	'lista 5 6 \$1418
14	!+ 5 \$1420
15	!
16	listashow imprime 6 5

Es una práctica común es usar una parte de memoria con un tamaño determinado, esto se puede lograr utilizan * luego del nombre de la variable. Toda la memoria especificada sera puesta en 0. La dirección de la variable mapa apunta a un lugar de memoria que tiene reservado 1 K byte de memoria.

1	
2	#mapa> 'mapa
3	
4	:+elemento elemento --
5	mapa> !+ 'mapa ! ;
6	
7	:recorre
8	'mapa (mapa> <?
9	@+ "%d " .println
10) drop ;
11	
12	3 +elemento
13	4 +elemento
14	5 +elemento
15	recorre imprime 3 4 5

Estas palabras se podrían definir si no estuviera en el diccionario base por ejemplo.

```
+! | val dir -
```

	CÓDIGO	ESTADO DE LA PILA
--	--------	-------------------

1	:+!	val Dir
2	dup	val Dir Dir
3	@	val Dir valEnD
4	rot	Dir valEnD val
5	+	Dir valEnD+Val
6	swap	valEnD+Va Dir
7	!	
8	;	

Es posible acceder a diferentes tamaños de valores con diferentes palabras para el manejo de memoria, estos pueden ser

byte/carácter	c@ c! c!+ c@+ c+!	1 byte	8 bits
word	w@ w! w!+ w@+ w+!	2 bytes	16 bits
dword	d@ d! d!+ d@+ d+!	4 bytes	32 bits
qword	@ ! !+ @+ +!	8 bytes	64 bits

Texto y memoria

Las comillas dobles permiten manejar texto dentro del programa. Se identifica este texto como prefijo y se marca el fin del texto con otra comilla doble.

El texto en el código fuente se denomina string o cadena de caracteres.

Se puede pensar el texto como una secuencia de bytes en memoria, así que en realidad, el manejo de texto es igual al manejo de memoria, la única diferencia es que la unidad será el byte en vez del qword (8 bytes).

En caso que se quiera incluir en este texto una comilla doble, se puede indicar una doble comilla doble para que incluya una comilla doble.

Algunos ejemplos de texto válido:

	<pre>"ejemplo de texto" " ejemplo de texto " "Decir ""HOLA"" a todos"</pre>
--	---

Cuando r3 encuentra un texto lo que hace es guardarlo en memoria, cada carácter va a ocupar un byte y corresponde al código ASCII, este código es una tabla de conversión de cada letra del alfabeto a un número, en este caso un byte, por ejemplo la letra A es el número 65, la B el número 66, etc. Al final de la memoria donde se guardo el texto se agrega un byte 0 (cero), indicando el final del mismo, hecho esto se apila la DIRECCIÓN del inicio del texto.

El lugar en la memoria donde se guarda este texto está separado de la memoria de las variables, salvo que este texto está en la definición de una variable, en cuyo caso, esta memoria si estará en el lugar de las variables.

	<pre>#vidas 3 #texto "CUIDADO con el perro" #perros 4</pre>
--	---

Vamos a estar trabajando con direcciones de memoria y tenemos que saber esto con precisión. Cuando se quiere recorrer el texto caracter por caracter, la siguiente definición imprime el código ASCII de cada carácter del texto.

1	:printascii t --
2	(c@+ 1? "%d " .print) 2drop ;
3	
4	"AB" printascii imprime por pantalla 65 66

Para imprimir los numeros que estan la pila, una de las definiciones más útiles es la palabra `sprint`, que se encuentra definida en la bibliotecas `r3/lib/mem.r3`, usada por `.print` y `.println`, esta palabra recorre el texto que se pasa como parámetro (en realidad a través de su dirección), buscando el carácter `%` y aquí reemplaza esta indicación con el texto convertido del valor que extrae de la pila.

`%d` : imprime el número en base decimal
`%b` : imprime el número en base binario
`%h` : imprime el número en base hexadecimal
`%s` : imprime el texto de la dirección
`%%` : imprime un signo %

1	253 254 255 "%d %b %h" .print
2	
3	imprime por pantalla:
4	255 11111110 fc

Se debe tener cuidado ya que el texto desapila los valores solicitados, a veces podemos olvidar un `over` o un `dup` para mantener estos números en la pila.

Algunas definiciones útiles para manejar texto están definidas en las bibliotecas `r3/lib/str.r3` y `r3/lib/parse.r3`.

Por ejemplo para contar los caracteres que contiene el texto:

1	::count s1 -- s1 cnt
2	0 over (c@+ 1?
3	drop swap 1 + swap) 2drop ;

Se recorre hasta encontrar un byte 0, sumando un contador en la pila. Cuando termina, quita el último 0 y la dirección, quedando la cantidad en la pila.

Registros A y B

Tenemos algo así como dos variables privilegiadas llamadas registros A y B. Estos registros son muy útiles para recorrer la memoria, para leer o escribir valores. Además se mantienen entre llamadas de palabras.

1	>A a -- ; carga el registro A
2	A> -- a ; apila el registro A, su valor.
	A+ a -- ; suma al registro A el valor en pila

3	A@ -- a ; trae de memoria
4	A! a -- ; guarda en memoria
5	A@+ -- a ; trae de memoria e incrementa A
6	A!+ a -- ; guarda en memoria e incrementa A
7	

Cuando se usan hay que tener cuidado que quien llame a estas palabras si los usa también, van a ser modificados, entonces es necesarios guardarlo de algún modo para que esto no ocurra.

1	#lista1 * \$ffff 64kb
2	#lista2 * \$ffff 64kb
3	#minimo
4	
5	:buscar
6	a@+ minimo <? ('minimo ! ;) drop ;
7	
8	'lista1 >a
9	a@ 'minimo !
10	1000 (1? 1- buscar) drop
11	
12	'lista2 >a
13	a@ 'minimo !
14	1000 (1? 1- buscar) drop
15	

la palabra buscar usará la dirección que tiene el registro A, en la línea 8 será la lista1 y en la línea 12 será la lista2, se podría pasar como parámetro en la pila pero habría que mover el contador de elementos. En la variable mínimo se tendrá el valor mínimo de cada lista al terminar el bucle. El ejemplo de los bucles anidados con el uso de registros evita el movimiento de pila necesario para traer la dirección recuperar el valor y avanzar.

1	#tabla * 800 10 x 10 x 8 (8 bytes por número)
2	
3	'tabla >a
4	0 (10 <? 1 +
5	0 (10 <? 1 +
6	a@+ "%d " .print
7) drop
8	.cr
9) drop

Pila de retorno

Existe una segunda pila que se encarga de manejar la llamada entre palabras, guardando la dirección de retorno que será usada cada vez que se ejecutar el fin de palabra ; Disponemos de las siguientes palabras para manipular la pila de retorno.

```
>R | a -- rstack: -- a
R> | -- a rstack: a --
R@ | -- a rstack: a -- a
```

Cuando se llama a una palabra (una definición de código, no de dato), el lenguaje apila en retorno el lugar donde debe retornar una vez que termina la ejecución de la palabra llamada.

Es conveniente no usar esta pila ya que un desnivel entre llamadas va a producir que el código se rompa.

Sin embargo se puede usar como un lugar alternativo para guardar o recuperar valores teniendo siempre cuidado que cada palabra que apile valores en este lugar tendrá que desapilar de forma precisa.

También es posible alterar esta pila para cambiar el flujo de ejecución, otra vez, hay que pensar con especial cuidado que va a pasar.

Conexión con el Sistema Operativo

La computadora solo trabaja con números y cualquier comunicación con el usuario o con el resto del mundo se realiza mediante palabras que resuelve el sistema operativo y no dependen del lenguaje, aunque sí de su conexión con el mismo.

El acceso a bibliotecas externas es fundamental porque nos permite acceder a capacidades del hardware no disponibles directamente, ya sea por criterio del fabricante que no desea que se vea su funcionamiento interno o por complejidad del recurso que no se desea volver a programar.

Generalmente se necesita la documentación de la biblioteca para saber qué funciones se necesita importar y el uso y funcionamiento de estas palabras será responsabilidad de la biblioteca, una vez que tenemos acceso a la biblioteca podemos construir nuestro programa a partir de ahí.

En el caso de Windows la conexión principal con el SO es la biblioteca KERNEL32.DLL. La biblioteca r3/win/kernel32.r3 hace la conexión para poder escribir por pantalla, cargar/grabar archivos, obtener información de procesos, etc.

Por ejemplo cuando inicia el programa, tenemos acceso a lo que se llama TERMINAL, podemos escribir texto, para hacer esto debemos incluir en nuestro programa la biblioteca:

1	<code>^r3/win/console.r3</code>	palabras para consola
2		
3	<code>: "hola mundo" .println ;</code>	imprime por consola

La palabra clave es `.println` cuya función es tomar la dirección del texto e imprimirlo por terminal y bajar a siguiente línea. Esta palabra está construida a base de llamadas al SO y algún procesamiento necesario para que sea más fácil el uso.

Conexión con el Sistema Operativo (Avanzado)

La conexión con el SO se construye a través de llamadas a funciones a bibliotecas dinámicas, en el caso de windows estas son las llamadas .DLL (dynamic link libraries).

La distribución actual, además de conectarse con el SO, hace uso de las capacidades gráficas del hardware mediante las bibliotecas SDL versión 2, ya que es multiplataforma.

biblioteca dinámica en windows	biblioteca en r3 en windows
SDL2.dll SDL2_image.dll	<code>^r3/win/sdl2.r3</code> <code>^r3/win/sdl2image.r3</code>

SDL2_mixer.dll	^r3/win/sdl2mixer.r3
SDL2_net.dll	^r3/win/sdl2net.r3
SDL2_ttf.dll	^r3/win/sdl2ttf.r3

r3 está preparado para usarse en Linux, MAC o RPI, aunque no están todavía disponibles.

La forma de conectarse con una biblioteca es mediante 2 palabras, para cargar la biblioteca y para obtener la dirección de la función a ejecutar y un conjunto de palabras que hacen la llamada.

```
LOADLIB      | "name" - liba
GETPROC      | liba "name" - aa
```

Las palabras para llamar a estas funciones van a tomar los parámetros de la pila, según la cantidad de parámetros y dejan la respuesta del SO.

```
SYS0  | aa -- r
SYS1  | a aa -- r
SYS2  | a b aa -- r
..
SYS10 | a b c d e f g h i j aa -- r
```

Programando un Juego en r3

Para hacer programas con gráficos necesitamos acceso a esta capacidad de nuestra computadora, para esto, vamos a utilizar la biblioteca SDL: <https://www.libsdl.org/>

Las palabras que manejan esta biblioteca las encontramos en SDL2GFX.r3

Este listado es la base de cualquier programa gráfico.

1	^r3/win/sdl2gfx.r3	incluye las palabras gráficas
2		
3	:juego	
4	0 SDLcls	limpia la ventan con color 0 (negro)
5	SDLredraw	refresca la ventana
6	SDLkey	apila la tecla
7	>esc<=? (exit)	si soltó escape... sale
8	drop ;	
9		
10	:main	
11	"r3sdl" 640 480 SDLinit	inicia ventana de 640 x 480
12	'juego SDLshow	repite palabra juego hasta exit
13	SDLquit ;	quita la ventana
14		
15	: main ;	

Este código va a crear una ventana de gráficos.

El programa empieza en la línea 15, llama a la palabra main.

En la línea 11 hace una ventana gráfica de 640 por 480 píxeles.

La línea 12 usa **SDLshow** para hacer un loop con esta palabra, se necesita la dirección de la palabra y no la palabra, hasta que, dentro de esta palabra, se llama a **exit**.

La línea 4 limpia la ventana con el color 0, negro, entre esta palabra y la siguiente podemos dibujar los graficos que queramos hasta la línea 5 donde redibuja la ventana.

La línea 6 obtengo la tecla del usuario.

en la línea 7 comparo si se soltó la tecla ESC y ejecutó **exit** si ocurre esto.

Dibujos del juego

Con esta base empezaremos por crear una hoja de sprites, esto es, un archivo con todos los gráficos que usaremos en el juego. Utilizaremos sprites de 16 x 16 píxeles.



Cada dibujo va a numerarse de arriba a abajo y de izquierda a derecha, comenzando de 0, tenemos 9 sprites.

Comenzamos definiendo una variable para guardar los sprites, y las coordenadas para dibujar
En la línea 19 utilizamos para cargar el archivo gráfico:

```
::ssload | w h file -- sprite
```

El ancho (w) y el alto (h) de cada sprite en la hoja y el lugar y nombre del archivo (file) es lo que va a obtener de la pila la palabra **SSLOAD** y el valor que queda es la referencia a este gráfico. Lo guardamos en una variable y lo vamos a utilizar cuando dibujemos en pantalla.

En la línea 8, para dibujar en pantalla usamos

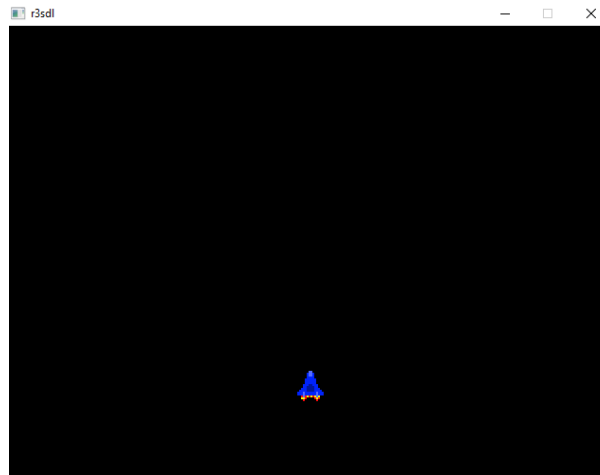
```
::sspritez | x y zoom nrosprite sprite --
```

Aquí X e Y son la posición de píxeles que será el centro de nuestro sprite, el nivel de zoom, aquí un número con punto decimal llamado número de punto fijo..donde podremos achicar o agrandar el dibujo, en nuestro caso lo duplicaremos, luego el número de sprite y por último la hoja donde se encuentra, que es donde lo cargamos.

El código se encuentra en la distribución del lenguaje:

```
1      ^r3/win/sdl2gfx.r3
2
3      #sprites
4      #x 320 #y 380
5
6      :juego
7          0 SDLcls
8          x y 2.0 0 sprites sspritez
9          SDLredraw
10
11         SDLkey
12         >esc< =? ( exit )
13         <le> =? ( -1 'x +! )
14         <ri> =? ( 1 'x +! )
15         drop ;
16
17     :main
18         "r3sdl" 640 480 SDLinit
19         16 16 "media/img/manual.png" ssload 'sprites !
20         'juego SDLshow
21         SDLquit ;
22
23     : main ;
```

r3/curso/manual01.r3



Mejorando el movimiento

En el código, las flechas izquierda y derecha restan y suman a la variable X, donde guardamos la posición de la nave.

Si probamos el programa notamos que el efecto de movimiento no es el deseado, necesitamos que mientras esté presionada la tecla, la nave se mueva y para esto necesitamos tener variables de velocidad y sumar constantemente esta velocidad a la posición.

Para que la velocidad tenga la posibilidad de ser una parte del entero vamos a utilizar números de punto fijo, para esto cuando lo indicamos debemos agregar obligatoriamente el punto decimal, y necesitamos convertirlo a entero cuando indicamos el pixel en pantalla por lo que utilizaremos la palabra

```
::int. | punto.fijo -- entero
```

Utilizamos **<le>** (tecla LEft o flecha a la izquierda) para detectar cuando se oprime la tecla y **>le<** cuando se suelta la tecla, ya que necesitamos cambiar la velocidad en estos dos estados.

Creamos una nueva palabra para separar el código que maneja el jugador, y acá dibujamos el sprite y sumamos las posiciones con las velocidades.

Probando el código notaremos la diferencia en el funcionamiento.

1	^r3/win/sdl2gfx.r3
2	
3	#sprites
4	#x 320.0 #y 380.0
5	#xv 0 #yv 0
6	
7	:jugador
8	x int. y int. 2.0 0 sprites sspritez
9	xv 'x +! yv 'y +! ;
10	
11	:juego
12	0 SDLcls
13	jugador
14	SDLredraw
15	
16	SDLkey

17	>esc<=? (exit)
18	<le>=? (-1.0 'xv !) >le<=? (0 'xv !)
19	<ri>=? (1.0 'xv !) >ri<=? (0 'xv !)
20	drop ;
21	
22	:main
23	"r3sdl" 640 480 SDLinit
24	16 16 "media/img/manual.png" ssload 'sprites !
25	'juego SDLshow
26	SDLquit ;
27	
28	: main ;

r3/curso/manual02.r3

Llegaron los extraterrestres

Agregamos otro elemento al programa, el alien, necesitamos la posición en **XA** y **YA**, y moverlo, no con las teclas, sino que sumando constantemente a **YA**, que va a ser que baje en la pantalla. La línea 11 reinicia el alien, **XA** será un número al azar entre 0 y el ancho de pantalla, en punto fijo. La línea 16 va a reiniciar el alien cuando salga por abajo, o sea, la coordenada **YA** sea mayor al alto de la ventana. La línea 24 primero dibuja el jugador y luego el alien.

..	
7	
8	#xa 0.0 #ya 100.0
9	
10	:+alien
11	-16.0 'ya ! 640.0 randmax 'xa ! ;
12	
13	:alien
14	xa int. ya int. 2.0 2 sprites sspritez
15	2.0 'ya +!
16	ya 480.0 >? (+alien) drop ;
..	
24	jugador alien
..	

r3/curso/manual13.r3

Disparos

Agregamos otras dos posiciones para guardar las coordenadas del disparo. Necesitamos indicar cuando está presente el disparo y para esto, en este caso, elegimos utilizar la variable XD, cuando sea negativa será que no hay disparo. Solamente podemos tener un disparo en pantalla. La palabra disparo va a dibujar la bala y modificar su posición. la línea 26 consulta sobre la variable xd y si esta es negativa, termina la palabra, esto es, no hace mas nada aqui. La línea 27 se encarga de dibujar el disparo. La línea 28 resta 4 a la posición en Y, quiere decir que irá subiendo en pantalla.

La línea 29 pregunta si la posición en YD es negativo, o sea, sale de pantalla, si ocurre esto pone en negativo el X para que no entre a esta palabra más, por la línea 26.

La palabra +disparo primero controla que no haya una bala en la pantalla, y si es posible, copiar las coordenadas de nuestra nave en las coordenadas de la bala para que funcione este mecanismo.

La línea 44 pregunta si se presiona la tecla ESCAPE y ejecuta +disparo cuando esto ocurre.

..	
22	#xd #yd
23	
24	:disparo
25	\$ffffff SDLColor
26	xd -? (drop ;) si es negativo sale
27	int. 1 - yd int. 3 8 SDLFRect dibuja un rectangulo relleno
28	-4.0 'yd +!
29	yd -? (-1.0 'xd !) drop ;
30	
31	:+disparo
32	xd +? (drop ;) drop
33	x 'xd ! y 'yd ! ;
..	
37	disparo jugador alien
..	

r3/curso/manual04.r3

Colisión y contar puntos

Definimos una variable para almacenar los puntos en la línea 35. Vamos a agregar la detección de la colisión entre la bala y el alien.

Para esto, en la línea 41, vamos a calcular la distancia que hay en x entre estas dos variables XA y XD, restando y obteniendo el valor absoluto y calculando que si hay más 16 pixeles, no va a producirse el choque.

Vamos a hacer lo mismo para la coordenada Y, en la línea 42, pero aquí calcularemos que la distancia sea de 8 pixeles, esto es, porque el dibujo del alien es mas ancho que alto, variando estos numeros sera mas dificil o mas facil pegarle.

Esto se realiza en la palabra CHOCO?, donde si se cumplen las condiciones, entonces sumaremos 1 al puntaje, borraremos el disparo y agregaremos un nuevo alien, borrando el anterior.

La línea 48 agrega al bucle principal esta comprobación del choque de la bala y el alien.

..	
35	#puntos 0
36	
37	:+punto
38	1 'puntos +! -1 'xd ! +alien ;
39	
40	:choco?
41	xa xd - abs 16.0 >? (drop ;) drop
42	ya yd - abs 8.0 >? (drop ;) drop

43	+punto ;
..	
48	choco?
..	

r3/curso/manual5.r3

Perder también hay que programarlo

Para que sea totalmente funcional falta agregar una variable de vidas, en la línea 37 y agregar cuando el alien choca a nuestra nave.

La forma de hacerlo es igual. Definiendo la palabra `perdio?` en la línea 51 y 52 calculamos que tanto en la coordenadas X como en Y la distancia entre las coordenadas del jugador y el alien sean menores a un umbral definido. Cuando esto ocurre se resta la cantidad de vidas y se comprueba que las vidas sean 0 para salir del programa llamado a la palabra `exit`.

las líneas 55 y 56 retornan la nave al inicio y agrega un nuevo alien, quitando el anterior para que continúe el juego.

..	
50	:perdio?
51	xa x - abs 16.0 >? (drop ;) drop
52	ya y - abs 16.0 >? (drop ;) drop
53	-1 'vidas +!
54	vidas 0? (exit) drop
55	320.0 'x !
56	380.0 'y !
57	+alien
58	;
..	
63	choco? perdio?
..	

r3/curso/manual06.r3

Mostrar texto

Para utilizar texto en una ventana gráfica, tenemos varias posibilidades. Veamos como unas de las más simples. Incluyendo a la biblioteca `bfont`, tenes dos fuentes de ancho fijo, en la línea 79 elegimos la fuente.

Usamos para ubicar dónde se va a imprimir el texto

`bat | x y --`

para imprimir un texto

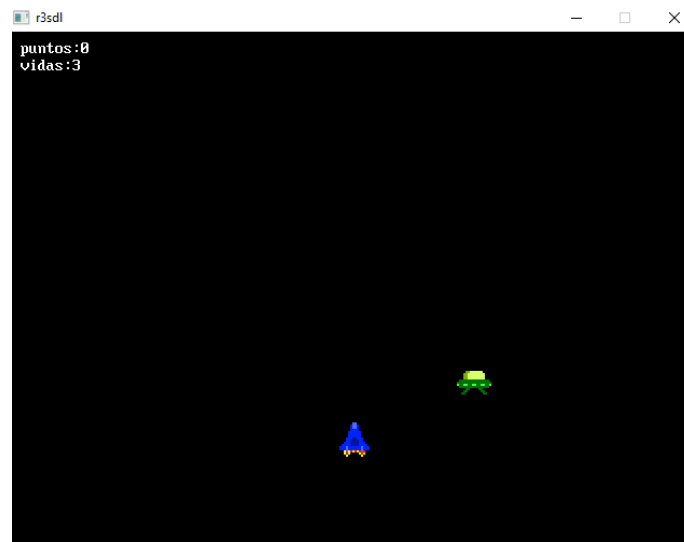
`bprint | "" --`

tener en cuenta que `bprint` no produce el texto con los modificadores de %, por eso acá usamos `sprint` para convertir a un texto con valores de la pila.

Las líneas 66 y 67 preparan el texto para mostrar vidas y puntos y lo imprimen en el lugar indicado por `bat`

..	
3	^r3/util/bfont.r3
..	
66	10 8 bat puntos "puntos:%d" sprint bprint
67	10 24 bat vidas "vidas:%d" sprint bprint
..	
79	bfont1
..	

r3/curso/manual7.r3



Estrellas de fondo

Vamos a agregar un fondo de estrellas que se mueven al juego utilizando un buffer de memoria. Necesitamos guardar posiciones X e Y para cada estrella y vamos a dibujar un pixel en estos lugares.

Para guardar estas posiciones utilizamos una parte de memoria, y elegimos guardar los números en 16 bits, 2 bytes, por eso utilizamos w!+ y w@+.

La palabra +estrella agrega las coordenadas X e Y en memoria en orden inverso dibuja.estrellas recorre en buffer y extrae estas coordenadas y llama a dibujar cada punto

Para mover las estrellas, en este caso vamos a incrementar en 1 la coordenada Y, cuando supera el límite de la pantalla (variable SH) vuelve a 0 para que aparezca por el otro lado de la ventana.

Por último tenemos una palabra para agregar las estrellas que necesitamos, SW es el ancho de la ventana, SH es el alto, **RANDMAX** produce un número aleatorio entre 0 y el valor de la pila.

Como indicamos un tamaño de 1024 bytes y cada estrella ocupa 4, 2 para x y 2 para Y, podremos guardar 256 coordenadas, una parte de la programar consiste en definir límites y cantidades

Necesitamos agregar el dibujo y movimiento de las estrellas al bucle principal, en la línea 84 y rellenar las coordenadas al inicio del programa

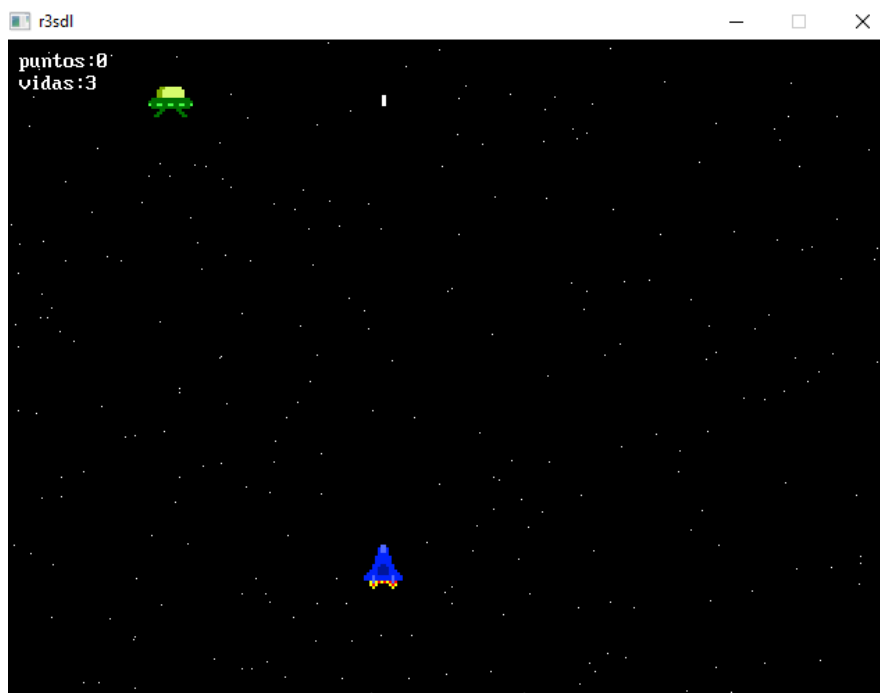
..	
5	#buffer * 1024 lugar para guardar las coordenadas
6	#buffer> 'buffer
7	


```

8      :+estrella | x y -
9          buffer> w!+ w!+ 'buffer> ! ;
10
11      :.estrellas
12      $ffffff sdlColor
13          'buffer ( buffer> <?
14              w@+ swap
15              over 1 + sh >? ( 0 nip )
16              over 2 - w!
17              w@+ rot
18              SDLPoint ) drop ;
19
20      :llenaestrellas
21          256 ( 1? 1 -
22              sw randmax sh randmax +estrella
23              ) drop ;
24
25      ..
84      .estrellas
26      ..
102     llenaestrellas
27      ..

```

r3/curso/manual08.r3



Animar

Los sprites de la hoja consisten en una imagen con las diferentes dibujos que representan animaciones, generalmente se trata de secuencias que representan caminar, correr, saltar, esperar para personajes o estado para objetos como, vaso lleno o vaso vacío.

La forma de utilizar estos gráficos es, primero, organizar estas animaciones en una grilla regular, es decir, todos los casilleros del mismo tamaño. Este tamaño se indica cuando se carga la hoja. Tenemos palabras que van a dibujar este sprite, indicando el número de gráfico, este número comienza en 0 y va de izquierda a derecha y de arriba a abajo.

Para animar lo que necesitamos es alternar este número en el tiempo, utilizamos una palabras construidas para este fin

Muchos sprites

A veces necesitamos dibujar muchas instancias de un determinado sprite.

Para esto usaremos una biblioteca que permite manejar una porción de memoria como lista de multi celdas con una capacidad de 16 números.

```
^r3/util/arr16.r3
```

Este módulo está construido para realizar exactamente lo que necesitamos.

Primero, reservar la cantidad de memoria necesaria para contener un máximo indicado por el programador, y necesitamos una variable para indicar a qué lista hacemos referencia.

Se necesita una variable con dos valores para guardar este mecanismo

```
#listalien 0 0
```

para inicializar vamos a definir cuál es la capacidad máxima de esta lista

```
200 'listalien p.ini
```

Cada elemento van a ser 16 celdas de memoria, en la primera vamos a poner una dirección de palabra especial que va a dibujar el sprite en pantalla y calcular la próxima posición.

cada vez que se llame a esta palabra, va a recibir de la pila la dirección de memoria donde está el inicio de las 15 celdas restantes y debe quitarla de la pila al finalizar la palabra.

Si queremos que este sprite desaparezca, vamos a dejar en la pila un 0.

En estas celdas podemos guardar las variables que queramos, como la posición, la velocidad de desplazamiento, la animación del sprite, etc.

Estos valores se acceden o modifican por la posición relativa a la dirección que tenemos en la pila, Aunque si queremos podemos definir un palabra que indique a qué nos estamos refiriendo.

Es conveniente definir las posiciones consecutivas de las celdas como datos que vamos a necesitar para, por ejemplo, dibujar nuestro sprite, así, como estamos utilizando:

```
sspritez | x y z nro sprite -
```

Podemos usar ese orden para guardar los valores, usando el registro A podemos recorrer y usar estos valores, líneas 26 a 32.

Para dibujar constantemente en pantalla debemos agregar en el bucle de dibujo, antes de SDLredraw, línea 69.

Podríamos agregar estos elementos con f1 para probar, entonces cuando testeamos las teclas presionadas agregamos, línea 80.

Esto debería agregar dibujos a la pantalla

..	
15	#listalien 0 0 lista de aliens
..	

```

25 |----- Aliens
26 :alien | adr -
27     >a
28     a@+ int. a@+ int. | x y , convertidos a enteros
29     a@+ | zoom
30     a@+ | nro de sprite
31     a@+ | hoja de sprite
32     sspritez ;
33
34 :+alien
35     'alien 'listalien p!+ >a
36     640.0 randmax a!+ | x
37     480.0 randmax a!+ | y
38     2.0 randmax 1.0 + a!+
39     12 randmax a!+
40     sprites a!+
41     ;
..
69     'listalien p.draw
..
80     <f1> =? ( +alien )
..
88     200 'listalien p.ini
..

```

r3/curs0/manual9.r3



Utilizando este recurso vamos a utilizar una lista de sprites para los disparos y otra para los aliens, aunque es posible utilizar una lista para ambos, a la vez, es conveniente separarlos porque cuando comprobemos si una bala chocó con un alien, esta separación nos simplifica el código ya para cada bala vamos a recorrer la lista de aliens. Si no está separado debería comprobar en cada recorrido el tipo de cada sprite.

```

14 #listalien 0 0 | lista de aliens
15 #listshoot 0 0 | lista de disparos
..
42 |----- Disparo
43 #hit
44 :choque | x y i n p -- x y p
45     dup 8 + >a
46     pick4 a@+ - pick4 a@+ -
47     distfast 20.0 >? ( drop ; ) drop
48     dup 'listalien p.del
49     1 'puntos +!
50     0 'hit !
51     ;
52
53 :bala | v --
54     objsprite
55
56     1 'hit !
57     dup @ | x
58     over 8 + @ -20.0 <? ( 3drop 0 ; ) | y ; fuera de pantalla
59     'choque 'listalien p.mapv | 'vector list --
60     2drop
61     hit 0? ( nip ; ) drop
62     drop
63     ;
64
65 :+disparo
66     'bala 'listshoot p!+ >a
67     x a!+ y 16.0 - a!+ | x y
68     1.0 a!+ | zoom
69     8 2 10 vci>anim | vel cnt ini
70     a!+ sprites a!+ | anim sheet
71     0 a!+ -3.0 a!+ | vx vy
72     ;
73
74 |----- Alien
75 :alien | v --
76     objsprite
77     dup @ x -
78     over 8 + @
79     500.0 >? ( 3drop 0 ; ) | llego abajo?
80     y - distfast
81     30.0 <? ( pierdevida )
82     drop
83     drop
84     ;
85
86 :+alien
87     'alien 'listalien p!+ >a
88     500.0 randmax 70.0 + a!+
89     -16.0 a!+
90     2.0 a!+ | zoom
91     7 4 2 vci>anim | vel cnt ini
92     a!+ sprites a!+ | anim sheet
93     2.0 randmax 1.0 -

```

```

94      a!+ 2.0 a!+ | vx vy
95      ;
96
97 :horda
98      50 randmax 1? ( drop ; ) drop
99      +alien
100     ;
..
127     'listalien p.draw
128     'listshoot p.draw
..
148     200 'listalien p.ini
149     200 'listshoot p.ini
..

```

r3/curso/manual10.r3

Explosiones y Efectos especiales

Los efectos especiales tienen como objetivo decorar lo que está pasando en la pantalla, la mayoría de las veces no influyen en el funcionamiento del juego, por lo tanto, vamos a construirlo con una lista de sprites que solamente se crean cuando ocurre algo y se destruyen automáticamente.

Vamos a agregar una lista sprites para las explosiones de los aliens, podríamos tener varios tipos de efectos especiales, y aquí si, agruparlos en una sola lista porque no van a tener interacción con el juego, son solo decorativos. Podríamos poner nubes, estrellas, efectos de chispas, quizás necesitaríamos varias listas si utilizamos distintos planos para estos sprites, es decir, que se dibujen antes o después del jugador y de los enemigos.

En la línea 24 preguntamos si estamos en el último cuadro de la animación de la explosion para quitar el sprite.

La línea 26 agrega la explosion cuando se produce la colisión del disparo y el alien utilizando las coordenadas del mismo.

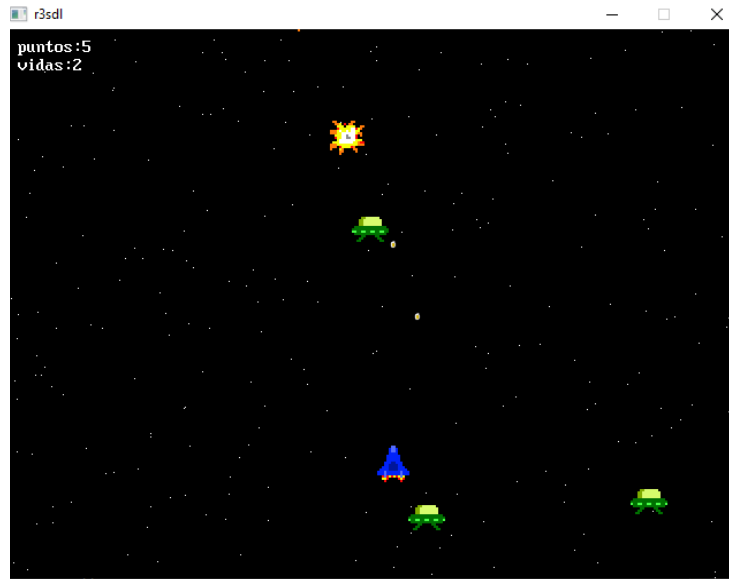
```

..
16 #listfx 0 0 | fx
..
28 |----- fx
29 :explosion
30     objsprite
31     24 + @ nanim 10 =? ( drop 0 ; )
32     drop
33     ;
34
35 :+explo      | y x --
36     'explosion 'listfx p!+ >a
37     swap a!+ a!+      | x y
38     2.0 a!+           | zoom
39     7 5 6 vci>anim     | vel cnt ini
40     a!+  sprites a!+ | anim sheet
41     0 a!+ 0 a!+      | vx vy
42     ;
..

```

68	<code>pick4 pick4 +explo</code>
..	
178	<code>200 'listfx p.ini</code>
..	

`r3/curso/manual11.r3`



Sonidos y Música

El sonido y la música agregan otro nivel de realismo al juego, para esto vamos a utilizar la biblioteca de música y sonidos de SDL2

`^r3/win/sdl2mixer.r3`

Al inicio del programa vamos a cargar los archivos de música o sonidos que se van a disparar durante la ejecución.

Para esto vamos a definir variables para guardar las referencias a estos elementos, podemos cargar un sonido o un tema musical completo

Las palabras para cargar los archivos de sonido son:

```
::Mix_loadMus | filename -- referencia
```

Carga en memoria un archivo de sonido que será usado como música, generalmente de larga duración.

```
::Mix_LoadWAV | filename -- referencia
```

Carga un archivo de sonido que será usado como efecto, generalmente de corta duración.

Los principales formatos de sonidos son .wav, .mp3 o .ogg, tanto para música como para sonido.

```
::Mix_PlayMusic | musica loop -
```

Reproduce un archivo de sonido, generalmente largo, música es la referencia que se cargó con Mix_LoadMus y loop será la cantidad de veces que se va a reproducir, con -1 se repetirá infinitamente, hasta que se apague, terminando la reproducción o comenzando otra música.

```
::SNDplay | sonido -
```

Reproduce un sonido, lo que necesita de la pila es la referencia que se cargó con Mix_LoadWAV, esta biblioteca mantiene varios canales de sonido, quiere decir que mientras suene la música pueden aparecer varios efectos de sonido, como explosiones o disparos y sonarán todos a la vez. La biblioteca posee más posibilidades pero con solamente estas palabras es posible hacer lo básico.

..	
8	#snd_shoot sonido de disparo
9	#snd_explode sonido de explosion
..	
45	snd_explode SNDplay en +explo
..	
94	snd_shoot SNDplay en +disparo
..	
172	"media/snd/shoot.mp3" Mix_LoadWAV 'snd_shoot !
173	"media/snd/explode.mp3" Mix_LoadWAV 'snd_explode !
..	

r3/curso/manual12.r3

Bibliotecas

Palabras más comunes de las principales bibliotecas.

Comunicación con el sistema operativo

^lib/core.r3

::msec | -- msec

Apila los milisegundos del sistema que transcurrieron desde que se inició el programa.

::time | -- hms

Apila la hora actual, hora, minutos y segundos, en un solo valor, con la siguiente distribución de bits.

```
::time | -- ; imprime el tiempo en formato hh:mm:ss
time
dup 16 >> $ff and "%d:" .print | hora
dup 8 >> $ff and      "%d:" .print | minuto
$ff and              "%d" .print | segundo
;
```

::date | -- ymd

Apila la fecha actual, año, mes y día, también está empaquetado en un solo número.

```
::date | - ; imprime fecha en formato yyyy-mm-dd
date
dup 16 >> $ffff and      "%d-" .print | anio
dup 8 >> $ff and          "%d-" .print | mes
$ff and                  "%d" .print | dia
;
```

Imprimir en consola

El lenguaje comienza en una ventana llamada consola donde lo que se puede escribir solamente texto, el lugar donde se escribe se lo llama cursor y este se desplaza a medida que ponen caracteres. Cuando llega al final de la consola, esta se desplaza hacia arriba para generar lugar, esto se lo llama SCROLL.

^win/console.r3

::.cls | --

Limpiar la consola, borrar todos los caracteres

::.write | "texto" --
Escribir un texto en la consola

::.print | .. "texto con %" --
Escribir un texto en la consola, si hay una secuencia de % extraer de la pila los valores necesarios y luego bajar de línea

::.home | --
Llevar el cursor al inicio de la consola

::.at | x y --
Ubicar el CURSOR a la fila y columna indicada

::.fc | color --
Define un color de frente para el texto

::.bc | color --
Define un color de fondo para el texto

::.input | --
Espera y guarda una línea de texto ingresada por teclado

##pad
La variable contiene el texto ingresado.

::.inkey | -- key
Retorna la tecla presionada, cero en caso que no haya ninguna

Números aleatorios

^lib/rand.r3
que incluye las palabras para generar números aleatorios.

::.rerand | s1 s2 -
Dado dos números, inicia el generador de números aleatorios.

::.rand | -- rand
Apila un número aleatorio de 64 bits

::.randmax | max -- valor
Apila un número elegido al AZAR entre 0 y MAX-1, cada vez que es llamado apila un número distinto.

Usualmente voy a inicializar con algún valor variable en el tiempo:

	time msec rerand
--	------------------

	10.0 randmax apila un número entre 0 y 10.0 5.0 randmax 5.0 - apila un número entre -5.0 y 0 5.0 randmax 5.0 + apila un número entre 5.0 y 10.0
--	---

Gráficos con biblioteca SDL2

La biblioteca SDL2 permite acceder a las capacidades gráficas. <https://www.libsdl.org/>
Esta librería permite iniciar una ventana gráfica y dibujar sobre ella, hay que tener en cuenta que, además de la ventana gráfica, es necesario tener los mecanismos para responder al TECLADO o al MOUSE y sus botones.

Ventana de gráficos

^win/sdl2.r3

::SDLinit | "titulo" w h --

Activar una ventana gráfica de w por h pixels, con título

::SDLfull | -

Poner la ventana en pantalla completa

::SDLquit

Salir de la ventan gráfica

::SDLcls | color --

Limpiar la pantalla con el color elegido

::SDLredraw | --

Refrescar la pantalla

::SDLshow | 'word --

Ejecutar WORD cada vez que se redibuja la pantalla

::exit

Salir de SHOW

##SDLkey

Código de la tecla presionada, cero si no hay tecla

##SDLchar

Código del carácter que representa

##SDLx

##SDLy

posición x e y del cursor en la ventana

##SDLb

Estado de los botones del cursor, cero cuando no está presionado ninguno

Cargar archivos gráficos

^win/sdl2image.r3

incluye win/sdl2.r3

Palabras para cargar archivos de imágenes, recordar que los PNG pueden tener transparencia y los JPG no.

::loading | "archivo" -- img

Cargar un archivo de imagen

::unloading | img --

Eliminar la imagen

Dibujar gráficos

^win/sdl2gfx.r3

incluye win/sdl2image.r3

incluye win/sdl2.r3

Palabras para dibujar diferentes formas, definidas por coordenadas de la ventana o cargadas desde un archivo con gráficos, generalmente un PNG

La biblioteca `^r3/win/sdl2gfx.r3` define palabras para dibuja como

::SDLColor | col --

Definir el color de dibujo

::SDLPoint | x y --

Dibujar un pixel

::SDLLine | x1 y1 x2 y2 --

Dibujar una línea de x1,y1 a x2,y2

::SDLFRect | x y w h -

Dibujar un rectángulo relleno

::SDLRect | x y w h --

Dibujar un rectángulo

::SDLFEllipse | rx ry x y --

Dibujar una elipse relleno

::SDLEllipse | rx ry x y --

Dibujar una elipse

::SDLTriangle | x y x y x y --

Dibujar un triángulo relleno

::SDLImagewh | img -- w h

Obtener el ancho y algo de un a imagen

::SDLImage | x y img --

Dibujar una imagen en la posición

::SDLImages | x y w h img --

Dibujar una imagen en posición con tamaño

::SDLImageb | box img --

Dibujar una imagen en la caja definida

::SDLImagebb | box box img --

Dibujar una parte de la imagen en la caja definida

::SDLspriteZ | x y zoom img --

Dibujar una imagen en posición con escala

::SDLSpriteR | x y ang img --

Dibujar una imagen en posición con rotación

::SDLspriteRZ | x y ang zoom img --

Dibujar una imagen en posición con rotación y escala

Hojas de Mosaicos o Tiles

::tsload | w h filename -- ts

Cargar una imagen como una hoja de tiles

::tscolor | rrggbb 'ts --

Definir el color de los tiles, se tiñe el color blanco

::tsdraw | n 'ts x y --

Dibujar un tile en pantalla

::tsdraws | n 'ts x y w h --

Dibujar un tile en pantalla con tamaño

Hojas de sprites

Los sprites que son partes de una imagen, se dibujan desde el centro del tamaño definido

::ssload | w h file -- ss

Cargar una hoja de sprites

::ssprite | x y n ss --

dibujar el sprite N en posición centrada

::sspriter | x y ang n ss --

Dibujar el sprite N en posición centrada con rotación

::sspritez | x y zoom n ss --

Dibujar el sprite N en posición centrada con escala

::sspriterz | x y ang zoom n ss --

Dibujar el sprite N en posición centrada con rotación y escala

Apéndices

Apendice 1 - Diccionario BASE

<pre> ; () [] -- vec EX vec -- 0? a -- a 1? a -- a +? a -- a -? a -- a <? a b -- a >? a b -- a =? a b -- a >=? a b -- a <=? a b -- a <>? a b -- a AND? a b -- c NAND? a b -- c BT? a b c -- a DUP a -- a a DROP a -- OVER a b -- a b a PICK2 a b c -- a b c a PICK3 a b c d -- a b c d a PICK4 a b c d e -- a b c d e a SWAP a b -- b a NIP a b -- b ROT a b c -- b c a 2DUP a b -- a b a b 2DROP a b -- 3DROP a b c -- 4DROP a b c d -- 2OVER a b c d -- a b c d a b 2SWAP a b c d -- c d a b >R a -- rstack: -- a R> -- a rstack: a -- R@ -- a rstack: a -- a AND a b -- c OR a b -- c XOR a b -- c NOT a -- b + a b -- c - a b -- c * a b -- c </pre>	<pre> Fin de palabra Comienzo de bloque para IF o WHILE Fin de bloque para IF o WHILE Comienzo de definición sin nombre Fin de definición sin nombre ejecutar una palabra por su direccion si TOS=Zero? conditional si TOS<>Zero? conditional si TOS>=0? si TOS<0? si a<b? quita TOS si a>b? quita TOS si a=b? quita TOS si a>=b? quita TOS si a<=b? quita TOS si a<>b? quita TOS si a AND b? quita TOS si a NAND b? quita TOS si a<=b<=c? quita TOS y NOS duplica TOS quita TOS duplica Segundo en la pila (NOS) duplica el 3er elemento duplica el 4er elemento duplica el 5er elemento intercambia TOS y NOS quita NOS Rota 3 elementos de la pila Duplica 2 valores Quita 2 elementos Quita 3 elementos Quita 4 elementos Dupila 2 elementos a partir del 3ro intercambia 4 elementos Apila en pila R Desapila de pila R Lee el tope de pila R c=a AND b c=a OR b c=a XOR b b=NOT a d=a+b d=a-b d=a*b </pre>
---	---

/	a b -- c	d=a/b
<<	a b -- c	d=a shift left b
>>	a b -- c	d=a shift right b
>>>	a b -- c	d=a shift right b sin signo
MOD	a b -- c	d=a mod b
/MOD	a b -- c d	c=a/b d=a mod b
*/	a b c -- d	d=a*b/c Sin pérdida de bits
*>>	a b c -- d	d=(a*b)>>c Sin pérdida de bits
<</	a b c -- d	d=(a<<c)/b Sin pérdida de bits
NEG	a -- b	b=-a
ABS	a -- b	b= a
SQRT	a -- b	b=square root(a)
CLZ	a -- b	b=count lead zeros of a
@	a -- [a]	fetch dword address
C@	a - byte[a]	fetch byte from address
W@	a - word[a]	fetch word address
D@	a - dword[a]	fetch dword address
@+	a -- b [a]	fetch qword and inc 8
C@+	a -- b byte[a]	fetch byte and inc 1
W@+	a -- b word[a]	fetch word and inc 2
D@+	a -- b dword[a]	fetch dword and inc 4
!	a b --	store A in address B
C!	a b --	store byte A in address B
W!	a b --	store word A in address B
D!	a b --	store dword A in address B
!+	a b -- c	store A in B and inc 8
C!+	a b -- c	store byte A in B and inc 1
W!+	a b -- c	store word A in B and inc 2
D!+	a b -- c	store dword A in B and inc 4
+	a b --	increment in mem B, A
C+	a b --	increment in mem B, byte A
W+	a b --	increment in mem B, word A
D+	a b --	increment in mem B, dword A
>A	a --	load register A
>B	a --	load register B
B>	-- a	push register B
A>	-- a	push register A
A+	a --	add to A
B+	a --	add to B
A@	-- a	fetch from A
B@	-- a	fetch from B
A!	a --	store in mem A
B!	a --	store in mem B
A@+	-- a	fetch A and inc 8
B@+	-- a	fetch B and inc 8
A!+	a --	store in mem A, inc 8
B!+	a --	store in mem A, inc 8
CA@	-- a	fetch from A
CB@	-- a	fetch from B
CA!	a --	store in mem A
CB!	a --	store in mem B
CA@+	-- a	fetch A and inc 1
CB@+	-- a	fetch B and inc 1

CA!+ a --	store in mem A, inc 1
CB!+ a --	store in mem A, inc 1
DA@ -- a	fetch from A
DB@ -- a	fetch from B
DA! a --	store in mem A
DB! a --	store in mem B
DA@+ -- a	fetch A and inc 4
DB@+ -- a	fetch B and inc 4
DA!+ a --	store in mem A, inc 4
DB!+ a -	store in mem A, inc 4
MOVE d s c --	copy S to D, C qword
MOVE> d s c --	copy from S to D, C qword in rev.
FILL d v c --	fill D, C qword with V
CMOVE d s c --	copy from S to D, C bytes
CMOVE> d s c --	copy S to D, C bytes in rev.
CFILL d v c --	fill from D, C bytes with V
DMOVE d s c --	copy S to D, C dwords
DMOVE> d s c --	copy from S to D, C dwords in rev.
DFILL d v c --	fill D, C dwords with V
MEM -- a	Comienzo de memoria libre
LOADLIB "name" -- liba	Cargar biblioteca dinámica
GETPROC liba "name" -- aa	Obtener dirección de función
SYS0 aa -- r	Lllamar a funcion sin parámetro
SYS1 a aa -- r	Lllamar a funcion con 1 parámetro
SYS2 a b aa -- r	Lllamar a funcion con 2 parámetros
SYS3 a b c aa -- r	Lllamar a funcion con 3 parámetros
SYS4 a b c d aa -- r	Lllamar a funcion con 4 parámetros
SYS5 a b c d e aa -- r	Lllamar a funcion con 5 parámetros
SYS6 a b c d e f aa -- r	Lllamar a funcion con 6 parámetros
SYS7 a b c d e f g aa -- r	Lllamar a funcion con 7 parámetros
SYS8 a b c d e f g h aa -- r	Lllamar a funcion con 8 parámetros
SYS9 a b c d e f g h i aa -- r	Lllamar a funcion con 9 parámetros
SYS10 a b c d e f g h i j aa -- r	Lllamar a funcion con 10 parámetros