Parseo y Generación de Código – 2do semestre 2016
Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Trabajo práctico 2
Generador de código para Cucaracha[1]

Fecha de entrega: 30 de noviembre

Índice

1. Introducción

Este trabajo consiste en programar el *back-end* para un compilador del lenguaje Cucaracha. Concretamente, dado el AST de un programa como el construido en el TP1, acompañado de la información de los tipos de todas las funciones, parámetros y variables locales, se debe generar código en *assembler* para la arquitectura x86-64.

Se recomienda utilizar el ensamblador `nasm`[2]. También usaremos algunas funciones de la biblioteca estándar de C, por lo que se recomienda *linkear* la salida con un compilador de C como `gcc`.

1.1. Ejemplo de programa en assembler

Vamos a trabajar siempre bajo una arquitectura de 64 bits. A continuación sigue un ejemplo de programa en assembler para comprobar que el entorno de desarrollo esté correctamente establecido. El siguiente ejemplo funciona en Linux. Debería funcionar en otros sistemas operativos POSIX con mínimas modificaciones.

```
section .text
global main
extern exit, putchar

main:

    mov rdi, 72
    call putchar

    mov rdi, 79
    call putchar

    mov rdi, 76
    call putchar

    mov rdi, 65
    call putchar

    mov rdi, 10    ; nueva línea (enter)
    call putchar

    mov rdi, 0
    call exit
```

- Guardar el archivo como `prueba.asm`.
- Ensamblarlo con `nasm -felf64 prueba.asm` para obtener un archivo objeto `prueba.o`.
- Linkear el archivo objeto con `gcc -o prueba prueba.o` para obtener un ejecutable `prueba`.
- Ejecutarlo con `./prueba`. Debería imprimir "HOLA" en pantalla.

[1] La cucaracha de Cucaracha es un dibujo de Michael Thompson licenciado bajo Creative Commons Attribution 3.0 United States License – `https://creativecommons.org/licenses/by/3.0/us/`

[2] `http://www.nasm.us/`

2. Guía de compilación

Describiremos cómo implementar el compilador de manera incremental, empezando por la implementación de la funcionalidad básica. Iremos agregando complejidad a medida que la necesitemos.

En términos generales, el compilador se estructura como un recorrido recursivo sobre el AST que genera código en assembler. La generación de código se puede hacer de distintas maneras:

- **Manera "funcional":** generando recursivamente alguna representación del programa en assembler y escribiendo la salida al final del proceso.
  - El código se puede representar directamente con un string:

    ```
    "mov rdi, 0\n" ++
    "add rax, rcx\n"
    ```

  - De manera más abstracta, se puede representar como un árbol o una lista de instrucciones:

    ```
    [InstruccionMov RegistroRdi (ValorInmediato 0),
     InstruccionAdd RegistroRax RegistroRcx]
    ```

- **Manera "objetosa":** emitiendo código assembler a medida que se visita el árbol.
  - El código se puede escribir directamente a un archivo de salida:

    ```
    salida.write('mov rdi, 0\n');
    salida.write('add rax, rcx\n');
    ```

  - De manera más abstracta, se puede contar con un objeto encargado indirectamente de construir la salida y generar el código assembler:

    ```
    salida.emitir_mov(RDI, 0);
    salida.emitir_add(RAX, RCX);
    ```

Se puede aplicar cualquiera las técnicas de arriba. Para un compilador de pequeña escala como el de este TP es razonable limitarse a una representación sencilla basada en strings.

Para los que programen en Haskell: tengan en cuenta que el recorrido sobre el AST puede requerir mantener estado para generar nombres de variables temporales, de modo que la función recursiva debería estar definida en una mónada.

2.1. Compilación de funciones y rutina principal

Cada función `fun f(...)` en el código fuente va a corresponderse con una rutina en assembler, que llamaremos `cuca_f`. La rutina principal `main` invoca a la rutina `cuca_main`, y luego invoca a la función `exit(0)` de la biblioteca estándar de C. Cada rutina tiene un `ret` al final para retornar.

Nota: llamamos `cuca_f` a la rutina para evitar conflictos con palabras clave y nombres de rutinas ya existentes; por ejemplo podríamos tener una función en Cucaracha que se llame `global` o `exit`, pero no podemos tener una rutina en assembler con esos nombres.

Por ejemplo:

Entrada	Salida

	`section .text`
	`global main`
	`extern exit`

`fun hola() {` `}`	`cuca_hola:` `  ret`
`fun chau() {` `}`	`cuca_chau:` `  ret`
`fun main() {` `}`	`cuca_main:` `  ret`

	`main:`
	`  call cuca_main`
	`  mov rdi, 0`
	`  call exit`

2.2. Compilación de constantes numéricas y primitivas putChar/putNum

Como primer paso para implementar todas las funcionalidades del lenguaje, veremos cómo compilar instrucciones del estilo "`putChar(65)`" y "`putNum(65)`", donde se invoca a alguna de las dos funciones primitivas con un parámetro numérico.

La funcionalidad para compilar una expresión (`ExprT`) tal como `65 + 2 * x` debería recibir dos parámetros:

1. La lista de registros del procesador que se encuentran disponibles para almacenar valores temporales.
2. El registro del procesador en el que se debe almacenar el resultado final del cómputo.

Para compilar una expresión que representa una constante numérica como `65` (`ExprConstNum`), lo único que se debe hacer es emitir una instrucción:

```
mov registro, 65
```

donde `registro` es el registro del procesador en el que se desea almacenar el resultado del cómputo de la expresión.

Al momento de compilar una instrucción que corresponde a una invocación a una función (`StmtCall`), si se trata de una función primitiva como `putChar` o `putNum`, el compilador no debe tratarlas igual que las invocaciones al resto de las funciones, sino que debe emitir código dedicado (específico).

- La función `putChar` de Cucaracha que imprime el caracter ASCII asociado a un número se compila a una invocación a la función `putchar` de la biblioteca estándar de C. El primer parámetro se recibe en el registro `rdi`. Por ejemplo:

Entrada	Salida

`fun main() {`
`  putChar(72)`
`  putChar(79)`
`  putChar(76)`
`  putChar(65)`
`  putChar(10)`
`}`
	`section .text`
	`global main`
	`extern exit, putchar`
	`cuca_main:`
	`  mov rdi, 72`
	`  call putchar`
	`  mov rdi, 79`
	`  call putchar`
	`  mov rdi, 76`
	`  call putchar`
	`  mov rdi, 65`
	`  call putchar`
	`  mov rdi, 10`
	`  call putchar`
	`  ret`
	`main:`
	`  call cuca_main`
	`  mov rdi, 0`
	`  call exit`

Observar que se debe agregar `putchar` a la lista de rutinas declaradas como `extern`.

- La función `putNum` de Cucaracha que imprime un número en notación decimal se compila a una invocación a la función `printf` de la biblioteca estándar de C. Por ejemplo, `putNum(65)` se compila a `printf("%lli", 65)`. Se debe dar valor a los siguientes tres registros para invocar a la función `printf`:
  - `rdi`: la constante de texto `"%lli"` que se ubica en la sección de datos.
  - `rsi`: el número a imprimir.
  - `rax`: la constante 0, indicando que no se proveen más parámetros que esos.

Entrada	Salida

`fun main() {`
`  putNum(12345)`
`}`
	`section .data`
	`lli_format_string db "%lli"`
	`section .text`
	`global main`
	`extern exit, printf`
	`cuca_main:`
	`  mov rsi, 12345`
	`  mov rdi, lli_format_string`
	`  mov rax, 0`
	`  call printf`
	`  ret`
	`main:`
	`  call cuca_main`
	`  mov rdi, 0`
	`  call exit`

Observar que se define una constante `lli_format_string` en la sección de datos donde se guarda la cadena `"%lli"`. Esta constante se pasa en el registro `rdi` a `printf`. Además, se agrega `printf` a la lista de rutinas declaradas como `extern`.

2.3. Pasaje de parámetros y variables locales

El pasaje de parámetros en Cucaracha hara uso de la siguiente convención:

- Cada parámetro y cada variable local tendrá asignado un espacio de 64 bits (8 bytes) en la pila del sistema.
- El registro `rsp` se utilizará (como es usual) como puntero al tope de la pila. El registro `rsp` apunta al elemento en el tope que está *presente* en la pila (es decir, no al primer espacio *libre*).

- El registro `rbp` se utilizará como puntero a la posición de la pila al inicio de la invocación a cada función.
- Tener en cuenta que en la arquitectura x86-64 la pila crece *hacia las direcciones de memoria más bajas*.
- Tener en cuenta que las instrucciones `call` y `ret` hacen push/pop de valores en la pila.
- La función invocadora mete los parámetros en la pila antes de hacer el `call`.
- La función invocada mete los parámetros en la pila antes de hacer el `call`.
- Por simplicidad ignoraremos todas las cuestiones de alineamiento.

Un esquema de la pila si se invoca a la función `f` que recibe dos parámetros `x` e `y` y tiene tres variables locales `a`, `b` y `c`:

direcciones de memoria bajas	0x00..0
⋮	
variable local `c`	← `rbp - 24` = `rsp` : tope de la pila
variable local `b`	← `rbp - 16`
variable local `a`	← `rbp - 8`
viejo valor del registro `rbp`	← `rbp`
dirección de retorno	← `rbp + 8`
parámetro `x`	← `rbp + 16`
parámetro `y`	← `rbp + 24`
⋮	
direcciones de memoria altas	0xff..f

`[rbp + 8]`	dirección de retorno
`[rbp + 16]`	valor del parámetro `x`
`[rbp + 24]`	valor del parámetro `y`
`[rbp - 8]`	valor de la variable local `a`
`[rbp - 16]`	valor de la variable local `b`
`[rbp - 24]`	valor de la variable local `c`

Siguiendo con esta convención, todas las rutinas deben ser de la siguiente forma:

```
cuca_f:
  push rbp            ; guardar el viejo valor de rbp en la pila
  mov rbp, rsp        ; actualizar el valor de rbp al tope actual de la pila
  sub rsp, 8 * N      ; reservar espacio en la pila para N variables locales
  .
  .
  mov rsp, rbp        ; recuperar el viejo tope de la pila
  pop rbp             ; recuperar el viejo valor de rbp
  ret
```

Además, el código para invocar a una función `f` con N argumentos $\texttt{e}_0$, $\texttt{e}_2$, ..., $\texttt{e}_{N-1}$ que no devuelve ningún resultado será de la forma:

```
  sub rsp, 8 * N      ; reservar espacio en la pila para N argumentos
  ...; evaluar el argumento 0 y guardarlo en [rsp]
  ...; evaluar el argumento 1 y guardarlo en [rsp + 8]
  .
  .
  ...; evaluar el argumento N-1 y guardarlo en [rsp + 8 * (N - 1)]
  call cuca_f
  add rsp, 8 * N      ; "devolver" el espacio de los N argumentos
```

Por ejemplo:

Entrada

```
fun f(a : Int, b : Int) {
    putChar(b)
    putChar(a)
}

fun main() {
  f(65, 66)
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_f:
    push rbp
    mov rbp, rsp
    mov rdi, [rbp + 24]  ; segundo parámetro (b)
    call putchar
    mov rdi, [rbp + 16]  ; primer parámetro (a)
    call putchar
    pop rbp
    ret
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 16          ; reservar espacio (2 argumentos)
    mov rdi, 65
    mov [rsp + 0], rdi   ; primer argumento (65)
    mov rdi, 66
    mov [rsp + 8], rdi   ; segundo argumento (66)
    call cuca_f
    add rsp, 16          ; liberar espacio (2 argumentos)
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

2.4. Accesos a parámetros y variables locales

Para compilar una expresión `x` y guardarla en un registro, basta con hacer un `mov`:

- Si `x` es el i-ésimo parámetro, se lo encuentra en la posición de memoria `[rbp + 8 * (i + 1)]` (por ejemplo, el segundo parámetro está en `[rbp + 24]`).
- Si `x` es la i-ésima variable local, se la encuentra en la posición de memoria `[rbp - 8 * i]` (por ejemplo, la segunda variable local está en `[rbp - 16]`).

Por ejemplo:

```
mov rdi, [rbp + 24]  ; leer el segundo parámetro
mov rsi, [rbp - 16]  ; leer la segunda variable local
```

El orden de los parámetros y variables locales se puede elegir arbitrariamente, pero obviamente la elección tiene que ser consistente a lo largo del programa.

2.5. Asignación a parámetros y variables locales

Para compilar una asignación `x := expresión`, compilar primero el valor de la expresión en algún registro, por ejemplo en `rdi`, y luego hacer un `mov` a la dirección en la que se encuentra el parámetro o la variable local.

Por ejemplo:

```
...; compilar la expresión y guardar el resultado en rdi
mov [rbp + 24], rdi  ; reasignar el segundo parámetro a dicho valor
```

Por ejemplo:

Entrada

```
fun f(a : Int) {
    a := 65
    b := 66
    putChar(a)
    putChar(b)
}

fun main() {
  f(0)
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_f:
    push rbp
    mov rbp, rsp
    sub rsp, 8           ; reservar espacio para b
    mov rdi, 65
    mov [rbp + 16], rdi  ; asignar a := 65
    mov rdi, 66
    mov [rbp - 8], rdi   ; asignar b := 66
    mov rdi, [rbp + 16]  ; acceder al parámetro a
    call putchar
    mov rdi, [rbp - 8]   ; acceder a la variable local b
    call putchar
    mov rsp, rbp         ; devolver el espacio de b
    pop rbp
    ret
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 8      ; reservar espacio para el parámetro a
    mov rdi, 0
    mov [rsp], rdi  ; pasaje del parámetro a = 0
    call cuca_f
    mov rsp, rbp    ; devolver el espacio del parámetro a
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

2.6. Compilación de constantes booleanas

Las constantes booleanas se representan con enteros de 64 bits. El falso se representa con la constante binaria $\underbrace{0\ldots0}_{64\text{ veces}}$ y el verdadero con la constante binaria $\underbrace{1\ldots1}_{64\text{ veces}}$. Notar que en complemento a 2 la constante `True` se puede escribir directamente como -1. Por ejemplo:

```
mov rdi, 0    ; guardar en rdi la representación de la constante False
mov rsi, -1   ; guardar en rsi la representación de la constante True
```

2.7. Compilación de operadores aritméticos, lógicos y relacionales

Para compilar una expresión e, supondremos que disponemos de un conjunto de registros que se encuentran libres, y que queremos guardar el resultado de evaluar e en uno de dichos registros.

Supongamos primero que los registros disponibles son suficientes para evaluar la expresión e. En ese caso podemos hacerlo con un algoritmo como el siguiente:

Entrada: Una expresión e y una lista de registros disponibles $r_1, r_2, \ldots, r_n$.
Salida: Código para evaluar la expresión e usando los registros $r_1, r_2, \ldots, r_n$ y dejando el resultado final en r_1.

if e es una expresión atómica
    por ejemplo una variable o una constante
  Generar código para "$r_1 := e$".
elseif e es una operación unaria **op**(e')
  Generar código para evaluar e' usando los registros $r_1, r_2, \ldots, r_n$.
  Generar código para "r_1 := **op**(r_1)".

```
elseif e es una operación binaria op(e1,e2)
    Generar código para evaluar e1 usando los registros r1,r2,...,rn.
    Generar código para evaluar e2 usando los registros r2,...,rn.
    Generar código para "r1 := op(r1, r2)".
end
```

El conjunto de los registros de los que vamos a disponer en la arquitectura x86-64 es:

```
rdi   rsi   rax   rbx   rcx   rdx   r8   r9   r10   r11   r12   r13   r14   r15
```

Si lo necesitan, pueden reservar algunos de los registros para almacenar valores temporales. Un ejemplo de cómo compilar una expresión sencilla que se puede calcular usando únicamente los registros del procesador:

Entrada

```
fun main() {
  a := 75 - (4 + 6)
  putChar(a)
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 8          ; reservar espacio para a
    mov rdi, 75         ; guardar 75 en el 1er registro
    mov rsi, 4          ; guardar 4 en el 2do registro
    mov rax, 6          ; guardar 6 en el 3er registro
    add rsi, rax        ; sumar 4 y 6
    sub rdi, rsi        ; restar 75 y 10
    mov [rbp - 8], rdi  ; guardar el resultado en a
    mov rdi, [rbp - 8]
    call putchar
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

En caso de que los registros no alcancen para compilar una expresión, se deben reservar más variables locales en la pila. Por ejemplo, si dispusiéramos únicamente de los registros `rdi` y `rsi`, el código anterior se compilaría de la manera siguiente (**comparar con el código anterior**):

Entrada

```
fun main() {
  a := 75 - (4 + 6)
  putChar(a)
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 16          ; reservar espacio para a y
                         ; para una variable temporal
    mov rdi, 75          ; guardar 75 en el 1er registro
    mov rsi, 4           ; guardar 4 en el 2do registro
    mov r15, 6
    mov [rbp - 16], r15  ; guardar 6 en la primera
                         ; variable temporal
    add rsi, [rbp - 16]  ; sumar 4 y 6
    sub rdi, rsi         ; restar 75 y 10
    mov [rbp - 8], rdi   ; guardar el resultado en a
    mov rdi, [rbp - 8]
    call putchar
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

Tener en cuenta que cada función debe reservar espacio en la pila para guardar todas sus variables locales y, además, todas las variables temporales que se requieran para compilar las expresiones que aparecen en su cuerpo. Para ello se recomienda emitir el "prólogo" de cada función después de haber visitado su cuerpo, para saber cuántas variables temporales se necesitan en total.

2.7.1. Código para compilar los operadores

Para compilar las operaciones binarias y unarias se recomienda usar las siguientes instrucciones:

Operación	**Código**	**Efecto**	**Comentarios**
suma	`add` x`,` y	$x := x + y$	
resta	`sub` x`,` y	$x := x - y$	
producto	`imul` x	`rax` $:=$ `rax` $* x$	Deja el resultado en `rax` y pisa el valor del registro `rdx`.
"y" lógico	`and` x`,` y	$x := x \& y$	Operación bit a bit.
"o" lógico	`or` x`,` y	$x := x \mid y$	Operación bit a bit.
"no" lógico	`not` x	$x := \sim x$	Operación bit a bit.

Por simplicidad puede ser conveniente reservar los registros `rax` y `rdx`, es decir, no considerarlos disponibles para almacenar los resultados intermedios en los cálculos de las expresiones, y usarlos únicamente como registros temporales para calcular el producto (y potencialmente como registros temporales para algunos otros fines).

Para compilar los operadores relacionales `<=`, `>=`, `<`, `>`, `==`, `!=`, se recomienda emitir la instrucción "`cmp` x`,` y" y a continuación una instrución de salto condicional.

Operación	**Código**	**Efecto**
Menor o igual	`jle` *etiqueta*	Salta a *etiqueta* si $x \leq y$.
Mayor o igual	`jge` *etiqueta*	Salta a *etiqueta* si $x \geq y$.
Menor estricto	`jl` *etiqueta*	Salta a *etiqueta* si $x < y$.
Mayor estricto	`jg` *etiqueta*	Salta a *etiqueta* si $x > y$.
Igual	`je` *etiqueta*	Salta a *etiqueta* si $x = y$.
Distinto	`jne` *etiqueta*	Salta a *etiqueta* si $x \neq y$.

Se deben crear etiquetas nuevas para cada operación relacional, y se sugiere combinar las operaciones de salto condicional anteriores con la operación de salto incondicional `jmp` *etiqueta*.

Por ejemplo:

Entrada	Salida

`fun main() {`
`  putChar(42 == 42)`
`}`
	`section .text`
	`global main`
	`extern exit, putchar`
	`cuca_main:`
	`  push rbp`
	`  mov rbp, rsp`
	`  mov rdi, 42`
	`  mov rsi, 42`
	`  cmp rdi, rsi  ; comparar los dos valores`
	`  je .label_1  ; si son iguales saltar a .label_1`
	`  mov rdi, 0  ; si no eran iguales`
	`  ;  guardar "False" en rdi`
	`  jmp .label_2  ;  y saltar a .label_2`
	`.label_1:`
	`  mov rdi, -1  ; si eran iguales`
	`  ;  guardar "True" en rdi`
	`.label_2:`
	`  call putchar`
	`  mov rsp, rbp`
	`  pop rbp`
	`  ret`
	`main:`
	`  call cuca_main`
	`  mov rdi, 0`
	`  call exit`

2.8. Compilación de invocaciones a funciones que devuelven valores y del `return`

Las funciones retornarán valores en un registro distinguido. Por ejemplo, pueden devolver el valor en el registro `rax`. De esta manera podemos compilar la instrucción `return` simplemente como un `mov` al registro `rax`. La invocación a una función que devuelve un valor se compila de igual manera que la invocación a un procedimiento. Cuando la subrutina retorna el control a la rutina que la invocó, puede asumir que el resultado de retorno se encuentra almacenado en `rax`.

Hay una dificultad extra al momento de compilar invocaciones a funciones. Por ejemplo, para compilar la expresión `1 + f()`, se almacena primero el `1` en el registro auxiliar `rdi` y luego se invoca a `f`. El inconveniente es que `f` podría necesitar usar el registro `rdi`. Lo que haremos es lo siguiente:

- Antes de invocar a una función, se almacenarán uno por uno todos los registros que estén siendo utilizados como auxiliares, usando la instrucción `push` *registro*.
- Al finalizar la invocación a una función, se recuperarán uno por uno todos los registros de la pila, usando la instrucción `pop` *registro*.

Entrada

```
fun uno() : Int {
  return 1
}

fun main() {
  putChar(65 + 2 * uno())
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_uno:
    push rbp
    mov rbp, rsp
    mov rax, 1   ; valor de retorno
    mov rsp, rbp
    pop rbp
    ret
cuca_main:
    push rbp
    mov rbp, rsp
    mov rdi, 65
    mov rsi, 2
    push rdi       ; guardar 1er registro temporal
    push rsi       ; guardar 2do registro temporal
    call cuca_uno  ; invocar a uno()
    pop rsi        ; recuperar 2do registro temporal
    pop rdi        ; recuperar 1er registro temporal
    mov rbx, rax   ; guardar en rbx el valor
                   ;   devuelto por la función uno()
    mov rax, rsi
    imul rbx
    mov rsi, rax
    add rdi, rsi
    call putchar
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

2.9. Compilación de estructuras de control – `if` y `while`

Las estructuras de control se compilan de la siguiente manera. Se muestra como compilar el `if` con `else`. Los nombres de las etiquetas deben generarse de tal manera que sean únicos (no debe haber dos etiquetas repetidas). El `if` sin else y el `while` quedan como ejercicios.

Entrada

```
if <condición> {
    <bloque1>
} else {
    <bloque2>
}
```

Salida

```
{código para evaluar <condición> en el registro rdi}
cmp rdi, 0      ; comparar el valor de la condición con 0
je .label_else  ; si es igual a 0 (es decir, False),
                ; saltar a .label_else
{código para ejecutar <bloque1>}
jmp .label_fin
.label_else:
{código para ejecutar <bloque2>}
.label_fin:
```

2.10. Compilación de vectores

Para compilar las operaciones de vectores se procederá de la manera que se describe a continuación.

2.10.1. Creación de un vector

Para compilar la creación de un vector $[e_1, \ldots, e_n]$ se reservará un espacio de $n + 1$ lugares en la pila. En el primer lugar se guardará el tamaño del vector (n) y en los n lugares restantes se guardará el resultado de evaluar cada una de las expresiones $e_1, \ldots, e_n$. El resultado final de evaluar la expresión $[e_1, \ldots, e_n]$ será la posición de memoria en la que empieza el vector. Por ejemplo:

Entrada

```
fun main() {
  x := [1, 2, 3]
}
```

Salida

```
section .text
global main
extern exit
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 8
    sub rsp, 32     ; reservar espacio para 4 lugares
                    ;   en la pila
    mov rdi, rsp    ; rdi es un puntero a la posición
                    ;   de memoria donde comienza el vector
    mov qword [rdi], 3        ; almacenar el tama\~no del vector
    mov rsi, 1
    mov qword [rdi + 8], rsi  ; almacenar el primer elemento
    mov rsi, 2
    mov qword [rdi + 16], rsi ; almacenar el segundo elemento
    mov rsi, 3
    mov qword [rdi + 24], rsi ; almacenar el tercer elemento
    mov [rbp - 8], rdi
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

2.10.2. Tamaño de un vector

Para calcular el tamaño de un vector basta con desreferenciar el puntero:

Entrada

```
fun main() {
  x := [1,2,3]
  putChar(#x)
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 8
    sub rsp, 32
    mov rdi, rsp
    mov qword [rdi], 3
    mov rsi, 1
    mov qword [rdi + 8], rsi
    mov rsi, 2
    mov qword [rdi + 16], rsi
    mov rsi, 3
    mov qword [rdi + 24], rsi
    mov [rbp - 8], rdi
    mov rax, [rbp - 8]          ; cargar en rax el valor de la
                                ;   variable local x
                                ;   que es un puntero a una posición
                                ;   de memoria
    mov rdi, [rax]              ; desreferenciar rax
    call putchar
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

2.10.3. Acceso a un vector

Para acceder a la n-ésima posición de un vector basta con desreferenciar el puntero; se debe sumar uno a la posición para tener en cuenta que la primera posición de memoria se utiliza para guardar el tamaño del vector, y se debe multiplicar el índice por 8 para tener en cuenta que el tamaño de cada entrada del vector es de 64 bits (es decir, 8 bytes). La multiplicación por 8 se puede implementar fácilmente usando la operación `sal rax, 3` que hace un *shift* 3 bits a la izquierda.

Entrada

```
fun main() {
  x := [10,20,30]
  putChar(x[2])
}
```

Salida

```
section .text
global main
extern exit, putchar
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 8
    sub rsp, 32
    mov rdi, rsp
    mov qword [rdi], 3
    mov rsi, 10
    mov qword [rdi + 8], rsi
    mov rsi, 20
    mov qword [rdi + 16], rsi
    mov rsi, 30
    mov qword [rdi + 24], rsi
    mov [rbp - 8], rdi
    mov rdi, 2          ; rdi = 2
    mov rax, rdi        ; rax = rdi
    inc rax             ; rax = rdi + 1
    sal rax, 3          ; rax = 8 * (rdi + 1)
    add rax, [rbp - 8]  ; rax = [rbp - 8] + 8 * (rdi + 1)
    mov rdi, [rax]      ; desreferenciar el puntero rax
    call putchar
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

2.10.4. Asignación a un vector

Para asignar a la n-ésima posición de un vector en la instrucción `x[n] := e` se debe almacenar el resultado de evaluar la expresión `e` en la posición de memoria correspondiente a n (que se debe calcular de manera análoga a la hecha para el acceso a un vector). Por ejemplo:

Entrada

```
fun main() {
  x := [10,20,30]
  x[2] := 33
}
```

Salida

```
section .text
global main
extern exit
cuca_main:
    push rbp
    mov rbp, rsp
    sub rsp, 8
    sub rsp, 32
    mov rdi, rsp
    mov qword [rdi], 3
    mov rsi, 10
    mov qword [rdi + 8], rsi
    mov rsi, 20
    mov qword [rdi + 16], rsi
    mov rsi, 30
    mov qword [rdi + 24], rsi
    mov [rbp - 8], rdi
    mov rdi, 2          ; rdi = 2
    mov rsi, 33
    mov rax, rdi        ; rax = rdi
    inc rax             ; rax = rdi + 1
    sal rax, 3          ; rax = 8 * (rdi + 1)
    add rax, [rbp - 8]  ; rax = [rbp - 8] + 8 * (rdi + 1)
    mov [rax], rsi      ; almacenar en la posición
                        ;  del puntero rax
    mov rsp, rbp
    pop rbp
    ret
main:
    call cuca_main
    mov rdi, 0
    call exit
```

3. Pautas de entrega

Para entregar el TP se debe enviar el código fuente por e-mail a la casilla `foones@gmail.com` hasta las 23:59:59 del día estipulado para la entrega, incluyendo `[TP lds-est-parse]` en el asunto y el nombre de los integrantes del grupo en el cuerpo del e-mail. No es necesario hacer un informe sobre el TP, pero se espera que el código sea razonablemente legible. Se debe incluir un README indicando las dependencias y el mecanismo de ejecución recomendado para que el programa provea la funcionalidad pedida.