Parseo y Generación de Código – 2do semestre 2018

Licenciatura en Informática con Orientación en Desarrollo de Software

Universidad Nacional de Quilmes

# Trabajo práctico 2

## Generador de código Flecha → Mamarracho

Fecha de entrega: 15 de noviembre

# Contents



# 1 Introducción

Este TP consiste en implementar un generador de código para el lenguaje de programación funcional Flecha, extendiendo el parser ya desarrollado para el TP 1. El lenguaje objeto será la máquina virtual Mamarracho, diseñada *ad hoc* para este TP. En la página de la materia pueden encontrar un intérprete de Mamarracho escrito en C++.

# 2 Descripción de los lenguajes

## 2.1 Lenguaje fuente (Flecha)

La sintaxis concreta de Flecha es la que ya fue especificada e implementada en el TP 1. En este TP el lenguaje Flecha contará con dos funciones primitivas `unsafePrintChar` y `unsafePrintInt` que sirven para mostrar valores en pantalla; esto no requiere extender la gramática, pero sí hacer algunas consideraciones. A continuación recordamos la forma que tienen los árboles de sintaxis abstracta de Flecha.

Un programa es una lista de definiciones:

`Program` ::= [ `Definition`, . . . , `Definition` ] Lista de $n$ definiciones, con $n \geq 0$.

Cada definición asocia un identificador a una expresión:

`Definition` ::= [`"Def"`, `ID`, `Expr`] Definición.

Las expresiones son las siguientes:

| | | | |
|---|---|---|---|
| `Expr` | ::= | `["ExprVar", ID]` | Variable. |
| | \| | `["ExprConstructor", ID]` | Constructor. |
| | \| | `["ExprNumber", NUM]` | Constante numérica. |
| | \| | `["ExprChar", NUM]` | Constante de caracter. |
| | \| | `["ExprCase", Expr, [CaseBranch, ..., CaseBranch]]` | Case de $n$ ramas, con $n \geq 0$. |
| | \| | `["ExprLet", ID, Expr, Expr]` | Declaración local. |
| | \| | `["ExprLambda", ID, Expr]` | Función anónima. |
| | \| | `["ExprApply", Expr, Expr]` | Aplicación. |

| | | | |
|---|---|---|---|
| `CaseBranch` | ::= | `["CaseBranch", ID, [ID, ..., ID], Expr]` | Rama del case de $n$ parámetros, con $n \geq 0$. |

## 2.2 Lenguaje objeto (Mamarracho)

El generador de código debe producir código para la máquina virtual Mamarracho. Al momento de la ejecución, el estado de Mamarracho consta de:

1. El **código fuente**, que es una secuencia de instrucciones, incluyendo definiciones de etiquetas.
2. El **puntero a la instrucción actual**.
3. Un **entorno global** que le da valor a los registros globales.
4. Un **entorno local** que le da valor a los registros locales.
5. Una **memoria** indexada por enteros de 64 bits.
6. Una **pila de llamadas** en la que se guardan las locaciones de retorno y el entorno local cada vez que se invoca a una función con las instrucciones `call`/`icall`.

En cada posición de memoria hay una **celda** de memoria. Una celda de memoria tiene espacio para guardar $n$ **slots** numerados desde 0 hasta $n - 1$. Cada *slot* almacena un **valor**. Usamos `Val` para representar el tipo de los valores. Los valores pueden ser:

1. **constantes numéricas**: enteros con signo de 64 bits,
2. **punteros** a celdas de memoria,
3. **locaciones**, que representan una posición dentro del código fuente del programa.

Más precisamente, el tipo de los valores `Val` está dado por:

| | | | |
|---|---|---|---|
| `Val` | ::= | `VInt(i64)` | donde `i64` representa un entero con signo de 64 bits |
| | \| | `VPtr(u64)` | donde `u64` representa un entero sin signo de 64 bits |
| | \| | `VLoc(u64)` | |

La máquina virtual cuenta con un número ilimitado de **registros globales** (`@r1`, `@r2`, `@r3`, ...) y **registros locales** (`$r1`, `$r2`, `$r3`, ...). El nombre de un registro es un identificador arbitrario[1] que comienza con arroba (`@`) para los registros globales y con el signo pesos (`$`) para los registros locales. Usamos `Reg` para representar el tipo de los registros. Cada registro puede almacenar un valor. Si $r$ es un registro en el que hay almacenado un puntero $p$ a una celda de memoria con $n$ slots, escribimos $r[i]$ para denotar el valor almacenado en el $i$-ésimo slot de la celda apuntada por $p$, para cada $0 \leq i < n$. La semántica informal se describe al costado de cada instrucción:

[1] Con la sintaxis léxica `[_a-zA-Z][_a-zA-Z0-9]*`.

| | | | |
|---|---|---|---|
| Instruction | ::= | mov_reg($r_1$ : Reg, $r_2$ : i64) | $r_1 := r_2$ |
| | \| | mov_int($r$ : Reg, $n$ : i64) | $r :=$ VInt($n$) |
| | \| | mov_label($r$ : Reg, $l$ : Label) | $r :=$ VLoc($p$), donde $p$ es la locación de la etiqueta $l$ en el código fuente |
| | \| | alloc($r$ : Reg, $n$ : u64) | $r :=$ VPtr($p$), donde $p$ es un puntero a una celda de memoria nueva con $n$ slots |
| | \| | load($r_1$ : Reg, $r_2$ : Reg, $i$ : u64) | $r_1 := r_2[i]$ |
| | \| | store($r_1$ : Reg, $i$ : u64, $r_2$ : Reg) | $r_1[i] := r_2$ |
| | \| | print($r$ : Reg) | Imprime en la salida el valor almacenado en $r$. |
| | \| | print_char($r$ : Reg) | Imprime en la salida el caracter almacenado en $r$. |
| | \| | jump($l$ : Label) | Salta a $l$. |
| | \| | jump_eq($r_1$ : Reg, $r_2$ : Reg, $l$ : Label) | Si $r_1 == r_2$, salta a $l$. |
| | \| | jump_lt($r_1$ : Reg, $r_2$ : Reg, $l$ : Label) | Si $r_1 < r_2$, salta a $l$. |
| | \| | add($r_1$ : Reg, $r_2$ : Reg, $r_3$ : Reg) | $r_1 := r_2 + r_3$ |
| | \| | sub($r_1$ : Reg, $r_2$ : Reg, $r_3$ : Reg) | $r_1 := r_2 - r_3$ |
| | \| | mul($r_1$ : Reg, $r_2$ : Reg, $r_3$ : Reg) | $r_1 := r_2 * r_3$ |
| | \| | div($r_1$ : Reg, $r_2$ : Reg, $r_3$ : Reg) | $r_1 := r_2$ `div` $r_3$ |
| | \| | mod($r_1$ : Reg, $r_2$ : Reg, $r_3$ : Reg) | $r_1 := r_2$ `mod` $r_3$ |
| | \| | call($l$ : Label) | Guarda la dirección de retorno y el entorno local actual en la pila. Crea un nuevo entorno local y salta a la locación de la etiqueta $l$. |
| | \| | icall($r$ : Reg) | Similar a `call`, pero salta a la locación almacenada en $r$. |
| | \| | return() | Restaura la dirección de retorno y el entorno local de la pila, retornando a la posición del último `call`/`icall`. |

Notar que para poder pasar parámetros y devolver resultados de funciones se deben utilizar los registros globales, ya que cuando se ejecuta una instrucción `call`/`icall` se crea un nuevo entorno local.

Un programa es una lista de instrucciones y declaraciones de etiquetas. Una declaración de etiqueta es de la forma "`etiqueta:`". Por ejemplo, el siguiente programa en Mamarracho imprime diez veces el número 9 en la salida:

```
  mov_int($actual, 0)
  mov_int($limite, 10)
  mov_int($uno, 1)
  mov_int($valor, 9)
iniciar_loop:
  jump_lt($actual, $limite, continuar_loop)
  jump(fin_loop)
continuar_loop:
  print($valor)
  add($actual, $actual, $uno)
  jump(iniciar_loop)
fin_loop:
```

# 3 Estructura del compilador

## 3.1 Representación de los datos

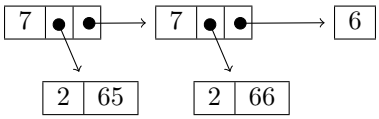
Todos los valores del lenguaje Flecha (enteros, caracteres, funciones y estructuras de datos armadas con constructores) se representan con celdas de memoria del lenguaje Mamarracho. El primer slot de la celda es siempre un entero que sirve como un **tag** para indicar qué clase de valor se encuentra almacenado en esa celda. Cada constructor que aparezca en el código fuente tendrá asociado un tag único.

Reservaremos un tag para representar valores de tipo `Int`, un tag para representar valores de tipo `Char`, y un tag para representar funciones (clausuras). La primera etapa del compilador consiste en un recorrido del AST para recolectar el conjunto de todos los constructores que aparezcan en el código fuente del programa y armar una tabla de tags. Por ejemplo, si en el código fuente del programa aparecen los constructores `True`, `False`, `Nil` y `Cons`, el

compilador podría generar la siguiente tabla de tags:

| Constructor | tag |
|---|---|
| número entero (`Int`) | 1 |
| caracter (`Char`) | 2 |
| clausura (`Closure`) | 3 |
| `True` | 4 |
| `False` | 5 |
| `Nil` | 6 |
| `Cons` | 7 |

Así, por ejemplo, el valor `Cons 'A' (Cons 'B' Nil)` se podría representar con el siguiente conjunto de celdas:




Asumiremos que cada constructor del lenguaje Flecha se utiliza siempre con la misma cantidad de argumentos, tanto en las aplicaciones como en las ramas de un case. Por ejemplo, en el siguiente programa el constructor `Nil` se utiliza con aridad 0, y el constructor `Cons` se utiliza con aridad 2:

```
def longitud l = case l
                 | Nil       -> 0
                 | Cons x xs -> 1 + longitud l
def lista = Cons 1 (Cons 2 (Cons 3 Nil))
```

Observar que esta restricción prohíbe utilizar constructores aplicados parcialmente; por ejemplo, en dicho programa sería incorrecto escribir una expresión como `(map (Cons 1) lista)` porque en esa expresión el constructor `Cons` se utiliza con aridad 1. Es opcional (pero recomendable) que el compilador verifique que todas las ocurrencias de un constructor tengan siempre la misma aridad.

## 3.2 Guía de compilación

Recordemos que un programa en Flecha consta de una lista de definiciones: $[\texttt{"Def"}, x_1, e_1], ..., [\texttt{"Def"}, x_n, e_n]$ de constantes y funciones globales. En el momento de la compilación, a cada uno de los nombres **globales** $x_i$ del programa le atribuiremos un registro global `@G_`$x_i$ en Mamarracho. Por ejemplo, si el programa define una constante global "`def misMejoresAmigos = ...`" su valor quedará almacenado en el registro `@G_misMejoresAmigos`.

Para compilar un programa de la forma $[\texttt{"Def"}, x_1, e_1], ..., [\texttt{"Def"}, x_n, e_n]$ el compilador genera código para calcular el valor de la expresión $e_i$ y almacenar ese valor en el registro `$G_`$x_i$, para todo $i = 1..n$. El "corazón" del compilador es una función recursiva que recibe un entorno, una expresión arbitraria $e$ del lenguaje Flecha, y un registro arbitrario $r$, y genera una secuencia de instrucciones en lenguaje Mamarracho que calculan el valor de la expresión $e$ y almacenan su resultado en el registro $r$. El tipo de dicha función será algo como:

```
compilarExpresion :: Env -> Expr -> Reg -> [Instruccion]
```

Donde `Env` representa un **entorno**. Un entorno es un diccionario que indica dónde se encuentra almacenado el valor de cada variable del lenguaje Flecha. Más precisamente, `Env` es un diccionario que asocia nombres de variables a `Binding`s. Hay dos posibles formas de *binding*:

```
Binding  ::=  BRegister(Reg)
           |  BEnclosed(Int)
```

Una asociación de la forma $x \mapsto \texttt{BRegister}(r)$ representa que la variable $x$ de Flecha se encuentra guardada en el registro $r$ de Mamarracho. El entorno se inicializa con todas las variables globales ligadas a registros, por ejemplo:

$$\{\texttt{main} \mapsto \texttt{BRegister}(\texttt{\$G\_main}), \texttt{foo} \mapsto \texttt{BRegister}(\texttt{\$G\_foo})\}$$

Cada vez que se introduce una definición de variable local el entorno se extiende asociando esa variable a un registro que no haya sido utilizado previamente. El *binding* `BEnclosed(`$i$`)` es necesario para las variables almacenadas en una clausura léxica, como veremos más adelante.

### 3.2.1 Caracteres

Nuestro primer objetivo será compilar el siguiente programa:

```
def main = unsafePrintChar 'A'
```

Para compilar un caracter, es decir una expresión ["ExprChar", $n$], se debe:

- Crear una celda de dos slots usando la instrucción `alloc`.
- Guardar el tag para el tipo `Char` en el primer slot usando la instrucción `store`.
- Guardar $n$ en el segundo slot usando la instrucción `store`.

Para compilar la expresión `(unsafePrintChar e)`, es decir cualquier expresión que tenga esta forma:

$$[\texttt{"ExprApply"}, [\texttt{"ExprVar"}, \texttt{"unsafePrintChar"}], e]$$

- Compilar recursivamente la expresión $e$ en el registro $r$
- Cargar el segundo slot usando la instrucción `load`.
- Mostrar el caracter en pantalla usando la instrucción `print_char`.

El programa compilado podría quedar así:

```
alloc($r0, 2)           % Reservar una celda de dos slots.
mov_int($t, 2)          % 2 = tag para el tipo Char.
store($r0, 0, $t)
mov_int($t, 65)         % 65 = letra 'A'
store($r0, 1, $t)
load($r1, $r0, 1)
print_char($r1)
mov_reg @G_main, $r0    % Guardar en @G_main el valor del caracter.
```

### 3.2.2 Números enteros

Se compilan de manera similar a los caracteres. El lenguaje incluye una primitiva `(unsafePrintInt e)` que se debe compilar usando la instrucción `print`.

### 3.2.3 Constructor aislado

Un constructor aislado (de aridad 0) se debe compilar como una celda de memoria con un único slot que contiene el tag del constructor. Por ejemplo, para compilar el constructor `True`:

```
alloc($r, 1)            % Reservar una celda de 1 slot.
mov_int($t, 4)          % 4 = Tag para el constructor True.
store($r, 0, $t)
```

### 3.2.4 Variables

Para compilar una variable ["ExprVar", $x$], se debe buscar en el entorno de qué manera está ligada.

Si la variable está ligada a `BRegister(`$r$`)`, significa que está almacenada en el registro $r$. En ese caso, se debe copiar el contenido de dicho registro al registro objetivo, usando la instrucción `mov_reg`. Por ejemplo, el resultado de compilar:

```
def foo = 42
def main = unsafePrintInt foo
```

podría ser el siguiente código:

```
% Código para "def foo = 42":
alloc($r0, 2)
mov_int($t, 1)
store($r0, 0, $t)
mov_int($t, 42)
store($r0, 1, $t)
mov_reg(@G_foo, $r0)

% Código para "def main = unsafePrintInt foo":
mov_reg($r1, @G_foo)
load($r2, $r1, 1)
print($r2)
mov_reg(@G_main, $r1)
```

Por otro lado, si una variable está ligada en el entorno a `BEnclosed(`$i$`)`, esto significa que la variable se encuentra almacenada en una clausura léxica. Más adelante se describe cómo compilar una variable en este caso.

### 3.2.5 Declaraciones locales (let)

Para compilar una declaración local ["ExprLet", $x, e_1, e_2$]:

- Reservar un registro fresco $\$tmp$.
- Compilar $e_1$ guardando su valor en $\$tmp$.
- Compilar $e_2$ en el entorno extendido con la asociación $x \mapsto \$tmp$.

Observar que el lenguaje tiene semántica *call-by-value* porque $e_1$ se evalúa siempre antes que $e_2$.

### 3.2.6 Funciones anónimas ($\lambda$)

Un programa Flecha compilado se organiza como un conjunto de rutinas auxiliares seguido de la rutina principal. Cada rutina está encabezada por una etiqueta ($\texttt{rtn}_1$, $\texttt{rtn}_2$, ..., $\texttt{rtn}_m$):

```
jump start
rtn_1:
  <rutina 1>
rtn_2:
  <rutina 2>
...
rtn_m:
  <rutina m>
start:
  <programa>
```

A cada función anónima que aparece en el código fuente le corresponde una rutina $\texttt{rtn}_i$. Una función anónima ["ExprLambda", $x, e$] se compila como una **clausura léxica**. Una clausura es una celda de memoria con $2 + n$ slots. En el primer slot se almacena el tag clausura. En el segundo slot se almacena la locación de la rutina $\texttt{rtn}_i$. En los $n$ slots restantes, se guardan los valores de todas las variables libres de aparecen en $e$ (excluyendo el parámetro $x$, los nombres de las constantes globales definidas en el programa como `misMejoresAmigos`, y las primitivas como `unsafePrintChar` o `ADD`). A dichas $n$ variables $a_1, a_2, \ldots, a_n$ las llamamos **variables clausuradas**.

Por ejemplo, el código para una función `(\ x -> e)` en la que aparecen tres variables libres `a`, `b` y `c` será algo como lo siguiente:

```
alloc($r, 5)       % Reservar una celda con 5 slots (5 = 2 + 3).
mov_int($t, 3)
store($r, 0, $t)   % Guardar el tag "clausura" en $r[0].
mov_label($t, rtn_i)
store($r, 1, $t)   % Guardar la rutina rtn_i en $r[1].
<guardar en $t el valor de la variable "a">
store($r, 2, $t)
```

```
<guardar en $t el valor de la variable "b">
store($r, 3, $t)
<guardar en $t el valor de la variable "c">
store($r, 4, $t)
```

Todas las rutinas reciben exactamente dos parámetros en los registros globales `@fun` y `@arg`, y devuelven exactamente un resultado en el registro `@res`. La rutina $\texttt{rtn}_i$ tiene la siguiente estructura:

```
rtn_i:
  mov_reg($fun, @fun)    % Mover el parámetro @fun a un registro local.
  mov_reg($arg, @arg)    % Mover el parámetro @arg a un registro local.
  % Ejecutar el cuerpo de la función y guardar
  % el resultado en un registro local $res.
  <código compilado para el cuerpo de la función>
  mov_reg(@res, $res)    % Mover el resultado al registro global @res.
  return()
```

**Acceso a parámetros y variables clausuradas.** Cuando se compila el cuerpo $e$ de una función `(\ x -> e)`, se debe extender el entorno del siguiente modo:

- El parámetro $x$ debe quedar ligado a `BRegister($arg)`. Es decir, el valor del parámetro $x$ se encuentra inmediatamente en el registro local `$arg`.
- Si las variables clausuradas son $a_1, a_2, \ldots, a_n$ el valor de la $i$-ésima variable clausurada se encuentra en el slot `$fun[i + 2]`. Por ejemplo, si se quiere recuperar el valor de $a_3$ y guardarlo en el registro `$r` se utilizará la siguiente instrucción:

  ```
  load($r, $fun, 5)
  ```

  Para ello se registrará en el entorno que (para cada $i = 1..n$) la variable $a_i$ se encuentra ligada a `BEnclosed(`$i$`)`.

### 3.2.7 Aplicación (caso típico)

Empezamos describiendo cómo compilar una aplicación `["ExprApply",` $e_1$`,` $e_2$`]` en el caso típico. En tal caso, el resultado de evaluar $e_1$ es una clausura. El código será como el siguiente:

```
<evaluar e1 y guardar el valor en un registro temporal $r1>
<evaluar e2 y guardar el valor en un registro temporal $r2>
load($r3, @fun, 1)  % Cargar la locación de la clausura en $r3.
mov_reg(@fun, $r1)  % Pasar e1 como parámetro en el registro global @fun.
mov_reg(@arg, $r2)  % Pasar e2 como parámetro en el registro global @arg.
icall($r3)          % Invocar.
mov_reg($r, @res)   % Obtener el resultado del registro global @res.
```

### 3.2.8 Aplicación (constructor)

Si en una aplicación `["ExprApply",` $e_1$`,` $e_2$`]` la función $e_1$ es un constructor posiblemente aplicado a argumentos (por ejemplo, $e_1$ = `(Cons 1)` y $e_2$ = `Nil`), debemos proceder de otra manera. En general, tendremos un constructor `C` aplicado a $n$ expresiones: `C` $e_1$ $\ldots$ $e_n$ y se debe generar código para construir una celda de memoria de $1 + n$ slots, donde el primer slot corresponde al tag del constructor. Por ejemplo, para `Cons` $e_1$ $e_2$ generaremos:

```
<evaluar e1 y guardar el valor en un registro temporal $r1>
<evaluar e2 y guardar el valor en un registro temporal $r2>
alloc($r, 3)
mov_int($t, 7)     % 7 = tag para el constructor Cons.
store($r, 0, $t)
store($r, 1, $r1)
store($r, 2, $r2)
```

### 3.2.9 *Pattern matching* (case)

Para compilar un `case` de la forma ["ExprCase", *expr*, *ramas*]. se debe, en primer lugar compilar la expresión $e$ guardando el resultado en un registro temporal `$val`. Supongamos que el case cuenta con $n$ ramas, cuyos constructores son $\texttt{C}_1, \texttt{C}_2, \ldots, \texttt{C}_n$, y que los tags de dichos constructores son $\texttt{TAG}_1, \texttt{TAG}_2, \ldots, \texttt{TAG}_n$, respectivamente. El case compilado tiene la forma siguiente:

```
<evaluar expr y guardar el valor en un registro temporal $val>
load($tag, $val, 0)           % Obtener el tag del valor analizado.
mov_int($test, <TAG_1>)
jump_eq($tag, $test, RAMA_1)  % Si el tag es TAG_1 saltar a la rama 1.
mov_int($test, <TAG_2>)
jump_eq($tag, $test, RAMA_2)  % Si el tag es TAG_2 saltar a la rama 2.
...
mov_int($test, <TAG_n>)
jump_eq($tag, $test, RAMA_n)  % etc.

RAMA_1: <código para la rama 1> jump FIN_CASE
RAMA_2: <código para la rama 2> jump FIN_CASE
...
RAMA_n: <código para la rama n> jump FIN_CASE
FIN_CASE:
```

Las etiquetas $\texttt{RAMA}_1, \ldots, \texttt{RAMA}_n$ y `FIN_CASE` son etiquetas frescas.

Para compilar una rama de la forma ["CaseBranch", *constructor*, $[x_1, \ldots, x_n], e$], se deben recuperar los $n$ argumentos del constructor que se encuentran almacenados la celda de memoria referenciada por `$val`, usando la instrucción `load`. El $i$-ésimo argumento del constructor se guarda en un registro temporal $\texttt{\$r}_i$, y se compila la expresión $e$ en el entorno extendido con $\{x_1 \mapsto \texttt{BRegister}(\texttt{\$r}_1), \ldots, x_n \mapsto \texttt{BRegister}(\texttt{\$r}_n)\}$

### 3.2.10 Aplicación (primitiva)

Si en una aplicación `["ExprApply", `$e_1$`, `$e_2$`]` la función es una primitiva aplicada **exactamente** al número de argumentos que recibe (por ejemplo, $e_1$ = `(ADD 11)` y $e_2$ = `31`) se debe emitir código para calcular la primitiva en cuestión.

Se deja como ejercicio opcional completar la implementación de todas las primitivas que reconoce el parser del TP 1: `OR`, `AND`, `NOT`, `EQ`, `NE`, `GE`, `LE`, `GT`, `LT`, `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `UMINUS`.

# 4 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. Se recomienda probar el programa con el conjunto de tests provistos.