Parseo y Generación de Código – 2do semestre 2023

Licenciatura en Informática con Orientación en Desarrollo de Software

Universidad Nacional de Quilmes

# Trabajo práctico 2

## Intérprete para el lenguaje Flecha

Fecha de entrega: 8 de noviembre

## Contents



# 1 Introducción

Este TP consiste en implementar un intérprete para el lenguaje de programación funcional Flecha, extendiendo el parser ya desarrollado para el TP 1.

# 2 Descripción del lenguaje 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 $n \geq 0$.

## 2.1 Guía de interpretación

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 interpretación, se cuenta con un entorno global `G`, de tal modo que a cada uno de los nombres **globales** $x_i$ del programa se le asocia un valor global $\texttt{G}[x_i]$.

Para interpretar un programa de la forma $[\texttt{"Def"}, x_1, e_1], ..., [\texttt{"Def"}, x_n, e_n]$ se debe calcular el valor de la expresión $e_i$ y almacenar ese valor en $\texttt{G}[x_i]$ para cada $i \in 1..n$. El "corazón" del intérprete es una función recursiva que recibe un entorno, una expresión arbitraria $e$ del lenguaje Flecha y devuelve el valor de la expresión $e$. El tipo de dicha función será algo como:

```
evaluar :: EntornoGlobal -> EntornoLocal -> Expresion -> Valor
```

Donde `EntornoLocal` es el tipo de los **entornos locales**. Un entorno local es un diccionario que indica el valor de cada variable local del lenguaje Flecha (excluyendo las variables globales, que están almacenadas en el entorno global).

Los valores del lenguaje se pueden representar con el siguiente tipo de datos inductivo en Haskell[1]:

```
data Valor = VChar Char
           | VInt Int
           | VStruct String [Valor]
           | VClausura String Expr EntornoLocal
```

donde:

1. `VChar c` representa el caracter `c`,
2. `VInt n` representa el número entero `n`,
3. `VStruct c [v1, ..., vn]` representa la estructura encabezada por el constructor `c` aplicado a los valores `v1`, ..., `vn`.
4. `VClausura x e env` representa la clausura en la que `x` es el nombre del parámetro formal, `e` es el cuerpo de la función y `env` es el entorno local en el que la clausura fue construida.

Por ejemplo, el resultado de evaluar la lista `Cons 1 (Cons 2 (Cons 3 Nil))` es el siguiente valor:

```
VStruct "Cons" [VInt 1, VStruct "Cons" [VInt 2, VStruct "Cons" [VInt 3, VStruct "Nil" []]]]
```

Además, el programa puede tener como efectos secundarios la escritura de caracteres en la salida. Si el intérprete se implementa en un lenguaje puramente funcional, el tipo de retorno debe permitir la presencia de efectos (por ejemplo, en Haskell, el tipo de retorno no será `Valor` sino `IO Valor`). Cada vez que se introduce una definición de variable local el entorno local se extiende asociando esa variable a su correspondiente valor.

### 2.1.1 Caracteres

Nuestro primer objetivo será interpretar el siguiente programa:

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

Al interpretar un caracter, se devuelve su valor. Para interpretar la expresión `(unsafePrintChar e)`, es decir cualquier expresión que tenga esta forma:

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

se debe interpretar recursivamente la expresión $e$ y escribir el caracter en la salida estándar (`stdout`).

El efecto de este programa debe ser imprimir el caracter `'A'` en la salida.

[1] Esta representación se puede y se debe adaptar de la manera que sea conveniente al lenguaje de implementación del intérprete.

### 2.1.2 Números enteros

Se interpretan de manera similar a los caracteres. El lenguaje incluye una primitiva `(unsafePrintInt e)` que debe escribir en la salida el número en notación decimal.

### 2.1.3 Constructor aislado

Un constructor aislado (de aridad 0) se debe interpretar como un valor que representa dicho constructor.

### 2.1.4 Variables

Para evaluar una variable $[\texttt{"ExprVar"}, x]$, se debe buscar en el entorno de qué manera está ligada. Primero se hace una búsqueda en el entorno local. Si no se la encuentra allí, se hace una búsqueda en el entorno global. Si no está en ninguno de los dos entornos, el programa resulta en un error.

### 2.1.5 Declaraciones locales (let)

Para evaluar una declaración local $[\texttt{"ExprLet"}, x, e_1, e_2]$, se debe interpetar $e_1$ obteniendo un valor $v_1$ y a continuación interpretar $e_2$ en el entorno local extendido para que $x$ tome el valor $v_1$. Observar que el lenguaje tiene semántica *call-by-value* porque $e_1$ se evalúa siempre antes que $e_2$.

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

El resultado de evaluar una función anónima debe ser una clausura, que incluye: el nombre del parámetro, el cuerpo de la función y el entorno local en el que la clausura fue creada.

### 2.1.7 Aplicación de función y de constructor

Para evaluar una aplicación `["ExprApply", `$e_1$`, `$e_2$`]`, se debe evaluar primero $e_1$ hasta obtener un valor $v_1$. Si el valor $v_1$ es una clausura, se debe evaluar $e_2$ y a continuación proceder a evaluar el cuerpo de la función en el correspondiente entorno local. Si el valor $v_1$ es un constructor posiblemente aplicado a argumentos (por ejemplo, $e_1$ = `(Cons 1)` y $e_2$ = `Nil`), se debe evaluar $e_2$ y construir un nuevo valor en el que el constructor pasa a estar aplicado a un argumento más.

### 2.1.8 *Pattern matching* (`case`)

Para evaluar un `case` de la forma $[\texttt{"ExprCase"}, e, ramas]$. se debe, en primer lugar evaluar la expresión $e$ obteniendo un valor $v$. A continuación, se deben recorrer las ramas una por una en orden hasta encontrar la primera que "encaje" con el valor $v$, y proceder a evaluar el cuerpo de la rama con el entorno local apropiadamente extendido.

### 2.1.9 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 deben evaluar todos sus argumentos y 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`.

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