Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores (CI-3725)

Proyecto 3 (15 %): MeLón
Análisis Semántico

Esta es la segunda (y final) etapa de nuestro interpretador de MeLón.

Antes de comenzar haremos tres cambios puntuales a la sintaxis del lenguaje. La Figura 1 presenta la nueva gramática. Observar que en la Figura 1 que lo único que cambió con respecto a la entrega anterior fue:

- Intercambio entre la precedencia del menos unario y el constructor de lista (::).
- Nueva expresión regular para los enteros.
- Lista vacía es una constante (lo que significa que ahora se puede usar lista vacía como patrón).

Semántica

Después de haber construído el árbol sintáctico y haber verificado que todos los casos de los **fun** tienen igual cantidad de parámetros, debe proceder a interpretar el programa. Esto significa que todos los errores que no sean sintácticos se producirán a tiempo de ejecución.

Para describir la semántica de un lenguaje funcional se necesita describir como evaluar cada tipo de expresión (cada tipo de árbol sintáctico). Para ello se presenta la función `eval` en la Figura 2.

Un concepto que suele generar confusión en los lenguajes funcionales es que las funciones son consideradas ciudadanos de primera clase. Esto es, las funciones pueden ser pasados como parámetros, retornados por una función, evaluados parcialmente, etc. En MeLón las funciones serán ciudadanos de primera clase. Para poder implementar esto se define la clausura que no es más que una estructura de datos para representar la evaluación parcial de funciones. La clausura se denota `CLS` en la descripción de la semántica de lenguaje que se presenta en la Figura 2[1].

Observe en la semántica del lenguaje el valor `fake`, el valor `fake` es distinto a todos los demás valores del lenguaje y se usa para la definición de funciones recursivas. `eval` nunca evaluará a `fake`, por lo que programas como `let x = x in x tel` no serán validos y dará el error de "Error de recursión".

Otra característica de MeLón es la de *matching* de patrones. Una constante hace *match* solo consigo misma (`True` con `True`, `False` con `False`, `[]` con `[]`, `12` con `12`, etc), una variable hace *match* con todo. Dos listas hacen match si sus cabezas hacen *match* y sus colas hacen *match*.

Al tener inferencia de tipo, los lenguajes derivados de ML no tienen errores de tipo en tiempo de ejecución. Debido a que MeLón no tiene inferencia de tipos, todos los errores de tipo van a ser detectados a tiempo de ejecución. Los tipos de los operadores se presentan en la Figura 3.

Los errores son los siguientes:

1. Error de recursión: recursión definida incorrectamente (descrita en el texto).
2. Error de *lookup*: no se encontró variable cuando se hizo `lookup` en el ambiente, por ejemplo: `let t = 1 in x tel`.

[1] Se denota `CLS` en mayúscula para aclarar que es una estructura de datos que se esta creando y no una función que se está definiendo como `eval` o `apply`

```
Constantes C:
    enteros: "[1-9][0-9]*"
    booleanos: "true" | "false"
    listavacia: "[]"
Variables V:
    [a-zA-Z][a-zA-Z0-9_]*
Expresiones E:
    C | V |
    E E |                                                                  (Asocia a izquierda)
    "-" E |                                                                (Asocia a derecha)
    E "::" E |                                                             (Asocia a derecha)
    E "*" E | E "/" E                                                      (Asocia a izquierda)
    E "+" E | E "-" E |                                                    (Asocia a izquierda)
    E "<" E | E "<=" E | E ">" E | E ">=" E |  E "=" E | E "<>" E |       (No asocia)
    "!" E |                                                                (Asocia a derecha)
    E "/\" E |                                                             (Asocia a izquierda)
    E "\/" E |                                                             (Asocia a izquierda)
    "fun" P11 ... P1m "->" E [... "|" Pn1 ... Pnm "->" E] "nuf" |
    "let" P "=" E "in" E "tel" |
    "if" E "then" E "else" E "fi" |
    "(" E ")"
Patrones P:
    C | V |
    P "::" P |
    "(" P ")"
```

Figura 1: Gramática de MeLón

```
eval env [] = []
eval env True = True
eval env False = False
eval env Num = Num

eval env x = lookup x env
eval env (let p = e1 in e2 tel) = let env1 = extend env p fake in
                                      let v1 = eval env1 e2 in
                                            replace env1 v1
                                      tel
                                  tel
eval env (fun p1->e1|...|pn->en) = (CLS env (p1, e1):: ... ::(pn,en)::[])
eval e1 e2 = apply (eval env e1) (eval env e2)

apply (CLS env (p,e)::cases ) v = if (match v p) then
                                       eval (extend env p v) e
                                  else
                                       apply (CLS env cases) v
                                  end
match True True = True
match False False = True
match [] [] = True
match _ x = True
match v1::v2 p1::p2 = (match v1 p1) and (match v2 p2)
match _ _ = False

lookup x (x,v)::env = v
lookup x (p1::p2, v1::v2)::env = lookup x p1 v1 ? lookup x p2 v2 ? lookup x env
lookup x v = v
```

Figura 2: Semántica de MeLón

3. Error de tipos: cualquier aplicación de una función con valores contravenga las reglas de tipo presentadas en la Figura 3, por ejemplo `1 + True`.

4. Error de aplicación: aplicar una no función por ejemplo: `1::2 3`.

5. Error de *matching*: ninguno de los patrones hace match con el parámetro. Por ejemplo: `(fun 1 ->1 | 2 ->2 nuf) 3`.

Ejemplos de evaluación

En esta sección hay varios ejemplos de evaluación de programas en MeLón. La idea es aclarar dudas sobre las reglas de evaluación de varios tipos de expresión. Nótese que en los ejemplos no necesariamente aparecen todos los pasos; lo importante es que aparecen todos los no triviales.

```
 (+): Entero x Entero -> Entero          (*): Entero x Entero -> Entero
 (-): Entero x Entero -> Entero          (/): Entero x Entero -> Entero
 (-): Entero -> Entero
(/\): Booleano x Booleano -> Booleano   (\/): Booleano x Booleano -> Booleano
 (!): Booleano -> Booleano
 (>): Entero x Entero -> Booleano        (<): Entero x Entero -> Booleano
(>=): Entero x Entero -> Booleano       (<=): Entero x Entero -> Booleano
 (=): Entero x Entero -> Booleano       (<>): Entero x Entero -> Booleano
 (=): Booleano x Booleano -> Booleano   (<>): Booleano x Booleano -> Booleano
(::): U x V -> (U x V)
```

Figura 3: Tipos de MeLón

let y fun

Se presenta un programa para obtener la multiplicación de dos por cinco utilizando una función de multiplicación por dos. El proceso de evaluación se puede observar en el listado 1.

```
let a = 2 in
let f = fun x -> x * a nuf in
f 5 tel tel
```

Test exitoso de lista vacía

En este programa vemos una función que prueba si una lista es vacía. Esta función es aplicada a una lista vacía. Se puede observar su evaluación en el listado 2.

```
fun (x::xs) -> false | [] -> true nuf []
```

Test fallido de lista vacía

Aquí vemos la función del ejemplo anterior ahora aplicada a una lista no vacía. El proceso de evaluación está en el listado 3.

```
fun (x::xs) -> false | [] -> true nuf [1,2,3,4]
```

Entrega

La entrega debe hacerse electrónicamente enviando un archivo tar comprimido (un tar.gz) con todos los archivos necesarios para ejecutar su programa dentro de un directorio. El directorio debe identificar al proyecto y a los autores. Por ejemplo si se está entregando el proyecto 1 y los autores tienen apellido Pérez y López entonces tanto el nombre del tar.gz como el nombre del directorio que genera debe ser proyecto1-perez-lopez. No incluya binarios en el tar.gz, si su proyecto necesita ser compilado incluya un makefile.

Su programa se debe invocar de la siguiente manera:

```
./interpretar_melon
```

Listado 1: Evaluación de 5*2

```
eval [] (let a = 2 in
         let f = fun x -> x * a nuf in
         f 5 tel tel)

=

eval [(a,2)] (let f = fun x -> x * a nuf in f 5 tel)

=

eval [(f,fun x -> x * a nuf),(a,2)] (f 5)

=

apply
(eval [(f,fun x -> x * a nuf),(a,2)] f)
(eval [(f,fun x -> x * a nuf),(a,2)] 5)

=

apply (eval [(a,2)] (fun x -> x * a nuf)) 5

=

apply (CLS [(a,2)] [(x,x*a)]) 5

=

match 5 x ?
eval (extend [(a,2)] x 5) x*a :
apply (CLS [(a,2)] []) 5

=

eval (extend [(a,2)] x 5) x*a :

=

eval [(x,5),(a,2)] x * a

=

10
```

Listado 2: Evaluación de test de lista vacía aplicado a lista vacía

```
eval [] (fun (x::xs) -> false | [] -> true nuf [])

=

apply (eval [] fun (x::xs) -> false | [] -> true nuf)
(eval [] [])

=

apply (CLS [] [((x:xs), false), ([], true)]) []

=

apply (CLS [] (((x::xs), false) :: [([], true)])) []

=

match [] (x::xs) ?
eval (extend [] (x::xs) []) false :
apply (CLS [] (([], true) :: [])) []

=

apply (CLS [] (([], true) :: [])) []

=

eval (extend [] [] []) true

=

eval [] true

=

true
```

Listado 3: Evaluación de test de lista vacía aplicado a lista no vacía

```
eval [] (fun (x::xs) -> false | [] -> true nuf [1,2,3,4])

=

apply (eval [] fun (x::xs) -> false | [] -> true nuf)
(eval [] [1,2,3,4])

=

apply (CLS [] [((x:xs), false), ([], true)]) [1,2,3,4]

=

apply (CLS [] (((x::xs), false) :: [([], true)]))
[1,2,3,4]

=

match (1::[2,3,4]) (x::xs) ?
eval (extend [] (x::xs) [1,2,3,4]) false :
apply (CLS [] (([], true) :: [])) [1,2,3,4]

=

eval (extend [] (x::xs) [1,2,3,4]) false

=

eval [(x, 1), (xs, [2,3,4])] false

=

false
```

y debe leer de la entrada standard y escribir salida standard.

Por favor incluya un archivo README donde indique cualquier cosa que no sea trivialmente clara a la persona que está corrigiendo. Indique claramente tanto en el README como en el correo electrónico información básica sobre su proyecto:

- Autores y carnets (con guión)
- Nombre y número del proyecto

Puede hacer su proyecto en cualquiera de los siguientes lenguajes de programación: Python (PLY), Java (Claire), Caml (camlyacc/camllex) o Haskell (Happy). Su programa debe funcionar sin modificación en cualquiera de las máquinas de la sala del LDC.

La entrega será electrónica enviando un email con el proyecto en formato tar.gz anexo a su instructor de práctica (y solo a su instructor de práctica). Puede entregar en cualquier momento antes del: 9/12/2009 a las 11:59:59 PM. Si no recibe un email de confirmación de su instructor en las 24 horas después de su entrega es su responsabilidad contactarlo inmediatamente. No será posible prorrogar esta entrega.