Lenguajes de Programación, 2021-1
Práctica 3: Semántica.

Favio E. Miranda Perea
favio@ciencias.unam.mx

Javier Enríquez Mendoza
javierem_94@ciencias.unam.mx

L. Fernando Loyola Cruz
loyola@ciencias.unam.mx

Fecha de entrega: 18 de Noviembre de 2020

Ya hemos definido las reglas necesarias para formar expresiones validas de nuestro lenguaje `EAB`.

```
type Id = String

data EAB =  V Id | N Int | B Bool
          | Succ EAB | Pred EAB
          | Plus EAB EAB | Mul EAB EAB | IsZero EAB
          | If EAB EAB EAB | Let Id EAB EAB
```

Ahora es turno de definir los mecanismos necesarios para dar un significado a las expresiones de nuestro lenguaje, es decir, la semántica del lenguaje.

Semántica Denotativa

Las semánticas operativas que veremos más adelante están definidas mediante reglas de inferencia que nos dicen cómo será la evaluación de una expresión. Pero esas reglas no nos dicen directamente cuál es el significado de una expresión. De esto se trata la *semántica denotacional*: mapear objetos sintácticos a su denotación o significado.

- Define la función `dEval` que recibe una expresión del lenguaje EAB y devuelve el valor asociado a ésta. La manera en como se debe hacer el paso de parámetros debe ser mediante sustitución.
  En nuestro lenguaje los posibles valores son los números naturales y los booleanos, esto lo podemos modelar en Haskell mediante el tipo de dato `Value` definido de la siguiente manera.

```
data Value = Nat Int | Boolean Bool

dEval :: EAB -> Value

{- Ejemplo -}
dEval $ Let "x" (Plus (I 1) (I 2)) (Plus (V "x") (Succ (I 0)))
    Nat 4
```

- Define la función `dEval2` que recibe una expresión del lenguaje EAB y devuelve el valor asociado a ésta. La manera es como se debe hacer el paso de parámetros debe ser haciendo el uso de un estado.

```
type State = Id -> Value

dEva2 :: State -> EAB -> Value

{- Ejemplo -}
s :: State
s _ = undefined

dEval2 s $ Let "x" (Plus (I 1) (I 2)) (Plus (V "x") (Succ (I 0)))
    Nat 4
```

Transpiladores.

La *transpilación* es un caso particular en donde el lenguaje objetivo es otro lenguaje de alto nivel. Un transpilador, entonces, es un programa que genera otro programa en otro lenguaje cuyo comportamiento es el mismo que el original.

Definamos los siguiente lenguajes de expresiones booleanas.

```
type Id = String

data EB1 = Tt1 | Ff1
            | VarB1 Id
            | Not EB1 | And EB1 EB1
            | Let Id EB1 EB1
            deriving (Show,Eq)

data EB2 = Tt2 | Ff2
            | VarB2 Id
            | If EB2 EB2 EB2
            | Where EB2 Id EB2
            deriving (Show,Eq)
```

- Defina la función `toEB2` que recibe una expresión del lenguaje `EB1` y regresa una expresión equivalente del lenguaje `EB2`. En los comentarios de la función explique la equivalencia de las expresiones.

```
toEB2 :: EB1 -> EB2
```

- Defina la función `toEB1` que recibe una expresión del lenguaje `EB2` y regresa una expresión equivalente del lenguaje `EB1`. En los comentarios de la función explique la equivalencia de las expresiones.

```
toEB1 :: EB2 -> EB1
```

- Defina un interprete para el lenguaje `EB1` dado por la función `eval1`. El paso de parámetros deberá ser mediante sustitución

```
eval1 :: EB1 -> EB1
```

- Defina un interprete para el lenguaje `EB1` dado por la función `eval2`. El paso de parámetros deberá ser mediante el uso de un estado.

```
eval2 :: State -> EB1 -> EB1
```

- Defina un interprete para el lenguaje `EB2` dado por la función `eval3`. El paso de parámetros deberá ser mediante sustitución

```
eval3 :: EB2 -> EB2
```

- Defina un interprete para el lenguaje `EB2` dado por la función `eval4`. El paso de parámetros deberá ser mediante el uso de un estado.

```
eval4 :: State -> EB2 -> EB2
```

- Defina la función `evalEB1` que evalue una expresión del lenguaje EB1 haciendo una transpilación a EB2 y luego evaluando la expresión resultante.

```
evalEB1 :: EB1 -> Bool
```

- Defina la función `evalEB2` que evalue una expresión del lenguaje EB1 haciendo una transpilación a EB2 y luego evaluando la expresión resultante.

```
evalEB2 :: EB2 -> Bool
```

Semántica Dinámica

Una manera de especificar el significado de una expresión es mediante una semántica operacional, la cual se puede definir por medio de un sistema de transiciones que especifica los posibles pasos de evaluación para todas las expresiones.

- Define la función `eval1` de manera que `eval1 e = e'` syss `e` $\longrightarrow$ `e'`.

```
eval1 :: EAB -> EAB

{- Ejemplo -}
> eval1 $ Plus (I 1) (Plus (I 10) (I 20))
    1 + 30 -- 1 + (10 + 20) ⟶ 1 + 30

> eval1 $ Let "x" (Plus (I 1) (I 2)) (Plus (V "x") (Succ (I 0)))
  let x = 3 in x + 0 -- let x = 1+2 in x+0 ⟶ let x = 3 in x+0
```

- Define la función `evals` de manera que `evals e = e'` syss `e` $\longrightarrow_*$ `e'` y `e'` es un estado bloqueado

```
evals :: EAB -> EAB

{- Ejemplo -}
> evals $ Let "x" (Plus (I 1) (I 2)) (Plus (V "x") (Succ (I 0)))
  3 -- let x = 1+2 in x+0 ⟶* 3

> evals $ Plus (Plus (I 10) (I 20)) (B True)
  30 + True -- 30 + True ⟶̸*
```

- Define la función `isValid` de manera que `evals e = e'` syss `e` $\longrightarrow_*$ `e'` y `e'` es un valor

```
isValid :: EAB -> Bool

{- Ejemplo -}
> isValid $ Let "x" (Plus (I 1) (I 2)) (Plus (V "x") (Succ (I 0)))
  True

> isValid $ Plus (Plus (I 10) (I 20)) (B True)
  False
```

Semántica Estática

Otra alternativa para determinar la semántica de un programa es mediante la semántica estática que consiste en analizar y verificar el tipo de las expresiones para garantizar que el programa esté bien formado en términos semánticos.
Dado que nuestro lenguaje cuenta con variables, necesitamos algún mecanismo para poder determinar el tipo de una expresión con variables. Para lograr esto se introduce el concepto de *contexto* que no es mas que una estructura en donde asociamos un tipo al nombre de una variable.

En Haskell podemos definir un contexto como una lista de tuplas en donde la primer proyección corresponde a la variable y la segunda proyección al tipo de esta.

```
type Ctx = [(Id, Type)]
```

en donde tipo es el `TDA` que define a los tipos del lenguaje `EAB`.

```
data Type = Nat | Boolean
```

- Define la función `vt` que recibe un contexto Γ, una expresión `e` del lenguaje EAB y un tipo `T` y decide si $\Gamma \vdash e : T$

```
vt :: Ctx -> EAB -> Type -> Bool

{- Ejemplos -}
> vt $ [] (Let "x" (Plus (I 1) (I 2)) (Plus (V "x") (Succ (I 0)))) Nat
    True

> vt $ [("x", Bool)] (Plus (V "x") (I 10)) Nat
    False
```