Lenguajes de Programación, 2021-1
Práctica 2: Sintaxis.

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

Javier Enríquez Mendoza
javierem_94@ciencias.unam.mx

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

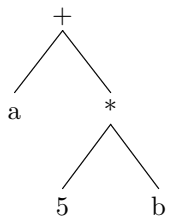
Fecha de entrega: 17 de Octubre de 2020

De manera muy simple un programa escrito en algún lenguaje de programación puede ser interpretado como una secuencia de caracteres. Para estudiar el significado del programa a alto nivel es necesario comprender como es que este fue formado, de manera que tenemos que tomar en cuenta 2 características esenciales de estos: La sintaxis y la semántica.

Sintaxis.

La sintaxis de un lenguaje esta formada por el conjunto de reglas que define las combinaciones de símbolos que forman expresiones correctamente estructuradas.

Los lenguajes de programación tienen una sintaxis tanto concreta como abstracta. La sintaxis *concreta* significa cadenas. Por ejemplo, `a + 5 * b` es una expresión aritmética dada como una cadena. La sintaxis *concreta* de un lenguaje generalmente se define mediante una gramática libre de contexto.
Por otro lado la expresión `a + 5 * b` también se puede ver como el siguiente árbol:



Ahora estamos en el nivel de sintaxis abstracta y estos árboles son llamados *árboles de sintaxis abstracta*. Para recuperar las ventajas de la notación de cadena lineal, escribimos nuestros árboles de sintaxis abstracta como cadenas con paréntesis para indicar el anidamiento (y con identificadores en lugar de los símbolos `+` y `*`), por ejemplo, la expresión anterior la representaríamos así:

```
Suma a (Mul 5 b)
```

Hemos transformado los árboles de sintaxis abstracta a términos "ordinarios" con los que nos será más fácil trabajar. Estos términos se encuentran sobre algún tipo de datos que define la sintaxis abstracta del lenguaje. Para entender como funcionan estos conceptos vamos a trabajar con un pequeño lenguaje de expresiones aritméticas y booleanas:

```
type Id = String

data EAB = V Id | N Int | B Bool
         | Succ EAB | Pred EAB
         | Add EAB EAB | Mul EAB EAB
         | Not EAB | And EAB EAB | Or EAB EAB
         | Lt EAB EAB | Gt EAB EAB | Eq EAB EAB
         | If EAB EAB EAB
         | Let Id EAB EAB
         deriving (Eq)
```

Es importante entender que hasta ahora solo hemos definido la sintaxis, no la semántica. Aunque hemos definido un constructor `Add`, este es simplemente un nombre sugerente y no implica que se comporte como una suma. Por ejemplo, la expresión `Add (N 0) (N 0)` es diferente de `N 0`, aunque puede parecer que son semánticamente equivalentes, sintácticamente no lo son.

- Define una instancia de la clase `Show` para el tipo de dato `EAB`.

```
instance Show EAB where
    show (V x) = show x
    ...

{- Ejemplo -}
> show $ Mul(Add((I 1), (I 2)) (N 10))
    (1 + 2) * 10
```

- Define la función `freeVars` que dada una expresión `E` de tipo `EAB` devuelve el conjunto de variables libres que aparecen en `E`.
  Nota: Recordar que una variable esta libre si no esta ligada por alguna expresión.

```
freeVars :: EAB  -> [Id]

{- Ejemplo -}
> freeVars (Add (V "x") (V "y"))
    [V "x", V "y"]

> freeVars (Let "x" (N 666) (Add (V "y") (V "x")))
    [V "y"]
```

- La sustitución es el proceso de reemplazar una variable por una expresión en otra expresión. Define la función `substitution` tal que `substitution e x a` es el resultado de reemplazar cada ocurrencia de `x` en `e` por `a`. Si x esta ligada entonces no realizar ningúna modificación.

```
{- Vamos a definir un tipo Substitution -}
type Substitution = (Id, EAB)

substitution :: EAB -> Substitution -> EAB

{- Ejemplo -}
> substitution (Add (V "x") (N 1)) ("x", N 10)
    Add (N 10) (N 1) -- En realidad la salida debería ser 10 + 1 debido al primer
        ejercicio

> substitution (Let "x" (N 666) (Add (N 0) (V "x"))) ("x", N 0)
    Let "x" (N 666) (Add (N 0) (V "x"))
```

- Dos expresiones son α-equivalentes si una se puede convertir en el otro simplemente haciendo sustituciones sobre las variables ligadas. Define la función `alphaEq` que recibe dos expresiones y determina si son α-equivalentes.

```
{- Vamos a definir un tipo Substitution -}
alphaEq :: EAB -> EAB -> Bool

{- Ejemplo -}
> alphaEq (Let "x" (I 1) (V "x")) (Let "y" (I 1) (V "y"))
    True

> alphaEq (Let "x" (I 1) (Add (V "x") (V "x")))
          (Let "y" (I 1) (Add (V "y") (V "z")))
    False
```

Optimizaciones.

El análisis de programas, también conocido como análisis estático, describe todo un campo de técnicas para el análisis estático (en tiempo de compilación) de programas. Los objetivos más comunes que se quieren lograr mediante un análisis estático son:

- **Optimización:** El propósito es mejorar el comportamiento del programa, generalmente reduciendo su tiempo de ejecución o los requisitos de espacio.
- **Detección de errores:** El propósito es detectar errores de programación comunes que conducen a excepciones en tiempo de ejecución u otros comportamientos no deseados.

La optimización del programa consiste en realizar una transformación del programa y consta de dos fases: el análisis (para determinar si se cumplen ciertas propiedades requeridas) y la transformación.

El `plegado constante` (o *constant folding)* es una de las optimizaciones más comunes realizadas por un compilador. El plegado constante significa calcular el valor de las expresiones constantes en el tiempo de compilación y sustituir su valor por el cálculo. por ejemplo, si tuviéramos el programa

```
let x = 42 - 5 in
    let y = x * 2 + 5 * 4 in y + 0
        y + 0
```

después de realizar un `plegado constante` obtendríamos el programa

```
let x = 37 in
    let y = x * 2 + 20 in y
```

- Define la función `aFolding` que reciba una expresión del lenguaje `EAB` y regrese la expresión resultante de aplicarle *plegado constante* a la expresión.
  Nota: Tener en cuanto el caso `e + 0`

```
aFolding :: EAB -> EAB

{- Ejemplo -}
> aFolding Mul(Add((I 1), (I 2)) (V "x"))
    Mul (N 3) (V "x")
```

- La noción anterior se puede aplicar sobre expresiones booleanas y es muy útil pues pueden ayudar a reducir la cantidad de código o incluso eliminar código que nunca será ejecutado. Define la función `bFolding` que reciba una expresión del lenguage `EAB` y le aplique el plegado constante además de eliminar ramas "muertas" de las expresiones `if`, es decir, si tenemos `if True a else b` esta expresión puede reducirse a simplemente `a`.
  Nota: Tener en cuanto los casos como `x or True`, `x and False`

```
bFolding :: EAB -> EAB

{- Ejemplo -}
> bFolding (If (Lt (N 1) (N 2) ) (V "x") (V "y"))
    -- Aplicamos constant folding y obtenemos:
    -- If (B True) V("x") (V "y")
    -- Eliminamos la rama muerta y obtenemos:
    V "x"

> bFolding (And (Not (V "y")) (Or (B True) (V "x")))
  And (Not (V "y")) (B True)
```