Compiladores, 2024-1
Práctica 2: Introducción al proceso de Compilación

Manuel Soto Romero
manu@ciencias.unam.mx

Javier Enríquez Mendoza
javiem@ciencias.unam.mx

Pedro Ulises Cervantes González
confundeme@ciencias.unam.mx

Braulio Aaron Santiago Carrillo
braulioa124@ciencias.unam.mx

01 de Agosto del 2023
Fecha de entrega: 15 de Agosto del 2023

1 Introducción

Considera el siguiente lenguaje de expresiones aritméticas y booleanas con notación post-fija **EAB**:

$$S ::= AExp \mid BExp$$
$$AExp\ a ::= v \mid n \mid a_0\ a_1\ + \mid a_0\ a_1 -$$
$$BExp\ b ::= t \mid f \mid b_0\ b_1 \&\& \mid b_0\ b_1 || \mid a_0\ a_1 =$$

El objetivo de esta practica es acercamos al proceso de compilación traduciendo este lenguaje a lenguaje ensamblador. La practica consiste en definir las siguientes funciones:

2 Análisis Léxico

El análisis léxico consiste en reunir secuencias de caracteres en unidades significativas llamadas tokens. Considera la siguiente definición de Tokens:

```
data Token = Var String | Number Int | Boolean Bool | Sum | Subs | And | Or | Equal deriving
    Show
```

2 pts Define la función **lexer** que recibe una cadena del lenguaje **EAB** y devuelve una lista de sus tokens.

```
lexer :: String -> [Token]
{ - Ejemplo -}
> lexer "22␣3␣+␣var␣==␣t␣&&"
> [Number 22,Number 3,Sum,Var "var",Equal,Boolean True,And]

> lexer "22␣+␣var␣var␣f␣t␣==␣&&"
> [Number 22,Sum,Var "var",Var "var",Boolean False,Boolean True,Equal,And]
```

3 Análisis Sintáctico

El análisis sintáctico consiste en determinar la estructura del programa, es decir, determina los elementos estructurales del programa así como sus relaciones. Considera la siguiente definición de ASA y Stack:

```
data ASA = VarASA String | NumberASA Int | BooleanASA Bool | Op Token ASA ASA deriving Show

type Stack = [ASA]
```

1.9 pts Define la función **scannerAux** que recibe una lista de tokens y un Stack, y devuelve su árbol de sintaxis abstracta correspondiente.

Hint: La solución es análoga al algoritmo para evaluar expresiones en notación post-fija. ¿Qué debemos guardar en el stack en lugar del valor parcial?

```
scannerAux :: [Token] -> Stack -> ASA
{ - Ejemplo -}
> scannerAux [Number 22,Number 3,Sum,Var "var",Equal,Boolean True,And] []
> Op And (BooleanASA True) (Op Equal (VarASA "var") (Op Sum (NumberASA 3)
    (NumberASA 22)))

> scannerAux [Number 22,Sum,Var "var",Var "var",Boolean False,Boolean
    True,Equal,And] []
> Expresion mal formada.
```

0.1 pts Define la función **scanner** que recibe una lista de tokens y devuelve su árbol de sintaxis abstracta correspondiente.

```
scanner :: [Token] -> ASA
{ - Ejemplo -}
> scanner [Number 22,Number 3,Sum,Var "var",Equal,Boolean True,And]
> Op And (BooleanASA True) (Op Equal (VarASA "var") (Op Sum (NumberASA 3)
    (NumberASA 22)))

> scanner [Number 22,Sum,Var "var",Var "var",Boolean False,Boolean True,Equal,And]
> Expresion mal formada.
```

4 Análisis Semántico

El análisis semántico consiste en verificar que el significado del programa sea claro y consistente con la especificación del lenguaje. Considera los siguientes tipos y sus juicios:

$$\frac{}{v : Number} \qquad \frac{}{n : Number} \qquad \frac{}{t : Boolean} \qquad \frac{}{f : Boolean}$$

$$\frac{a_0 : Number \qquad a_1 : Number}{a_0\ a_1\ + : Number} \qquad \frac{a_0 : Number \qquad a_1 : Number}{a_0\ a_1\ - : Number}$$

$$\frac{b_0 : Boolean \qquad b_1 : Boolean}{b_0\ b_1\ \&\& : Boolean} \qquad \frac{b_0 : Boolean \qquad b_1 : Boolean}{b_0\ b_1\ || : Boolean} \qquad \frac{a_0 : Number \qquad a_1 : Number}{a_0\ a_1\ == : Boolean}$$

```
data Type = Num | Bool deriving Show
```

1.9 pts Define la función **TypeCheckerAux** que recibe un **ASA** y devuelve el tipo de la expresión únicamente si el tipado del programa es consistente. En otro caso arroja un error indicando el problema con el programa.

```
typeCheckerAux :: ASA -> Type
{ - Ejemplo -}
> typeCheckerAux (Op And (BooleanASA True) (Op Equal (VarASA "var") (Op Sum
    (NumberASA 3) (NumberASA 22))))
> Bool

> typeCheckerAux (Op And (NumberASA 43) (Op Equal (VarASA "var") (Op Sum (NumberASA
    3) (NumberASA 22))))
>  El tipo de los argumentos NumberASA 43 y Op Equal (VarASA "var") (Op Sum
    (NumberASA 3) (NumberASA 22)) no son los esperados para el operador And
```

0.1 pts Define la función **TypeChecker** que recibe un **ASA** y devuelve dicho **ASA** si el tipado del programa es consistente.

```
typeChecker :: ASA -> ASA
{ - Ejemplo -}
> typeChecker (Op And (BooleanASA True) (Op Equal (VarASA "var") (Op Sum (NumberASA
    3) (NumberASA 22))))
> Op And (BooleanASA False) (Op Equal (VarASA "var") (Op Sum (NumberASA 3)
    (NumberASA 22)))

> typeChecker (Op And (NumberASA 43) (Op Equal (VarASA "var") (Op Sum (NumberASA 3)
    (NumberASA 22))))
> El tipo de los argumentos NumberASA 43 y Op Equal (VarASA "var") (Op Sum
    (NumberASA 3) (NumberASA 22)) no son los esperados para el operador And
```

5 Optimización de Código Fuente

1 pto El plegado constante es una optimización que elimina expresiones cuyo valor se puede calcular previo a ejecutar el código. Define la función **constantFolding** que recibe un **ASA** y devuelve el **ASA** resultante de aplicarle plegado contante.

```
constantFolding :: ASA -> ASA
{ - Ejemplo -}
> constantFolding (Op And (BooleanASA True) (Op Equal (VarASA "var") (Op Sum
    (NumberASA 3) (NumberASA 22))))
> Op Equal (VarASA "var") (NumberASA 25)

> constantFolding (Op And (Op Equal (VarASA "var") (VarASA "var")) (Op Equal
    (VarASA "var") (Op Sum (VarASA "var") (NumberASA 22))))
> Op And (Op Equal (VarASA "var") (VarASA "var")) (Op Equal (VarASA "var") (Op Sum
    (VarASA "var") (NumberASA 22)))
```

Considera las siguientes definiciones

```
data Value = N Int | B Bool | S String
instance Show Value where
  show (N n) = show n
  show (B b) = show b
  show (S s) = show s

data ThreeAddress = Assign String Value| Operation String String Token String
instance Show ThreeAddress where
  show (Assign t v) = show t ++ "␣=␣" ++ show v
  show (Operation t a op b) = show t ++ "␣=␣" ++ show a ++ tokenThreeAddress op ++ show b
```

0.2 pts Define la función **fresh** que recibe una lista de enteros y devuelve el menor natural posible que no este en la lista.

```
fresh :: [Int] -> Int
{ - Ejemplo -}
> fresh [1,2,3]
> 0

> fresh [4,2,3,0]
> 1
```

0.7 pts Define la función **threeAddressAux** que recibe un **ASA** y la lista de los enteros que han sido usado en variables temporales, y devuelve una tripleta con la traducción correspondiente en código de tres direcciones, la variable temporal que almacena el resultado actual y los enteros que se han usado para variables temporales.

```
threeAddressAux :: ASA -> [Int] -> ([ThreeAddress],String,[Int])
{ - Ejemplo -}
> threeAddressAux (Op Equal (VarASA "var") (NumberASA 25)) []
> (["t0" = "var","t1" = 25,"t2" = "t0" == "t1"],"t2",[2,1,0])

> threeAddressAux (Op Equal (NumberASA 50) (VarASA "var")) []
> (["t0" = 50,"t1" = "var","t2" = "t0" == "t1"],"t2",[2,1,0])
```

0.1 pts Define la función **threeAddressAux** que recibe un **ASA** y devuelve su traducción correspondiente en código de tres direcciones.

```
threeAddress :: ASA -> [ThreeAddress]
{ - Ejemplo -}
> threeAddress (Op Equal (VarASA "var") (NumberASA 25))
> ["t0" = "var","t1" = 25,"t2" = "t0" == "t1"]

> threeAddress (Op Equal (NumberASA 50) (VarASA "var"))
> ["t0" = 50,"t1" = "var","t2" = "t0" == "t1"]
```

6 Generación de Código

La generación de código recibe una representación intermedia del programa y genera código en lenguaje maquina. Considera las siguientes nemotecnias de lenguaje ensamblador.

```
Asignar el valor V en el registro R1
    MOV R1 V

Aplicar la operacion con los registros R2 y R3, y guardar el valor en el registro R1
    ADD R1 R2 R3
    SUBS R1 R2 R3
    AND R1 R2 R3
    OR R1 R2 R3
    EQ R1 R2 R3
```

2 pts Define la función **assembly** que recibe un programa en código de tres direcciones y devuelve su traducción correspondiente a lenguaje ensamblador.

```
assembly :: [ThreeAddress] -> String
{ - Ejemplo -}
> assembly ["t0" = "var","t1" = 25,"t2" = "t0" == "t1"]
> MOV "t0" "var"
  MOV "t1" 25
  EQ "t2" "t0" "t1"

> assembly ["t0" = 50,"t1" = "var","t2" = "t0" == "t1"]
> MOV "t0" 50
  MOV "t1" "var"
  EQ "t2" "t0" "t1"
```

7 Extra

0.5 pts Utilizando las funciones definidas anteriormente Define la función **compile** que recibe un programa en **AEB** y devuelve su traducción correspondiente a lenguaje ensamblador.

```
compile :: String -> String
{ - Ejemplo -}
> compile "22␣3␣+␣var␣==␣t␣&&"
> MOV "t0" "var"
  MOV "t1" 25
  EQ "t2" "t0" "t1"
```

8 Entrega

- La practica será entregada en equipos de máximo 5 integrantes.
- La entrega será por Google Classroom.
- Únicamente un miembro del equipo sube la solución de la practica. El resto debe indicar los integrantes de su equipo en un comentario privado.
- Únicamente anexar el archivo con extensión `.hs` con la solución. El nombre del archivo debe ser el nombre del integrante que subió el archivo empezando por apellidos.
- Deberás entregar el ejercicio a más tardar a las 23:59 del día indicado. Después de esta hora, el ayudante rechazará el ejercicio.