IMPLEMENTACIÓN DE UN COMPILADOR EMPLEANDO EL LENGUAJE DE PROGRAMACIÓN HASKEL

JAIME CEPEDA VILLAMAYOR
ANDRÉS MONTORO MONTARROSO

INDICE

1. INTRODUCCIÓN

En el caso de estudio que se nos ha asignado, se ha diseñado e implementado un intérprete de un lenguaje imperativo utilizando el lenguaje de programación funcional Haskell. Para conseguir este fin, se han seguido los principios de diseño de los procesadores de lenguaje, que son:

- Análisis léxico: Que permite extraer los distintos tokens del lenguaje
- Análisis sintáctico: Que permite extraer la estructura del lenguaje dados los tokens, generando un Árbol Sintáctico Abstracto.
- Análisis semántico: Que interpreta el Árbol Sintáctico Abstracto conforme a la visión que se quiere dar.

Dadas estas partes, el procesador de lenguaje se ha desarrollado en 3 partes adaptándose a las fases del desarrollo de un procesador de lenguaje. En los siguientes apartados se detallará el proceso de desarrollo de las distintas fases.

2. ESPECIFICACIÓN DEL LENGUAJE

El lenguaje que se ha querido implementar, es una mezcla de varios lenguajes, que soporta las siguientes características:

1. Realiza operaciones aritméticas de suma, resta, multiplicación y división.
2. Tiene soporte de asignación de variables.
3. Soporta bucles if e if/else simples.
4. Soporta bucles while.
5. Imprime valores enteros por pantalla

Este lenguaje se ha detallado con una especificación **BNF**, que se detallará a continuación.

```
prog
=
exp
prog
|
exp;
```

```
(*
%left '+' '-'
%left '*' '/'
%left '<' '>' '<=' '>='
%left '&&' '||'
*)
```

```
exp = assignation      |
      if_expression    |
      print_expression |
      while_expression |
      operation        ;


assignation = var ident assignToken intOp newLineToken;


intOp = ident                   |
        intToken                |
        intToken plus intOp     |
        intToken minus intOp    |
        intToken multiply intOp |
        intToken divide intOp   ;


if_expression = if_body endIf newLineToken |
                if_body else_expr          ;


if_body = ifToken condition newLineToken if_bloq ;


else_expr = elseToken newLineToken if_bloq endIfToken newLineToken ;


if_bloq = bloq if_bloq  |
          bloq          ;


bloq = assignation      |
       operation        |
       print_expression ;


condition = intOp conditionToken intOp  ;


while_expression = whileToken condition newLineToken bloq_while endWhileToken
newLineToken ;


bloq_while = bloq bloq_while |
             bloq            ;
```

```
operation = intOp newLineToken ;

print_expression = printToken intOp newLineToken ;

intToken = "[0-9]+" ;

ifToken = "if" ;

endIfToken = "endif" ;

whileToken = "while" ;

endWhileToken = "endwhile" ;

conditionToken = ['||' '&&' '<' '>' '<=' '>='] ;

printToken = "print" ;

newLineToken = '\n' ;

plus = '+' ;

minus = '-' ;

multiply = '*' ;

divide = '/' ;

ident = [a-z A-Z \ \_ \'];
```

3. CONSTRUCCIÓN DEL INTERPRETE

En esta parte, se mostrará el proceso de desarrollo del intérprete. Se ha usado un desarrollo basado en tests, lo que nos ha permitido tener las distintas partes del código probado, evitando posibles errores no esperados a la hora de ejecutar el programa.

3.1. ANÁLISIS LÉXICO

La función del análisis léxico es reconocer los símbolos o tokens que componen el texto fuente. Cada token es un elemento del lenguaje fuente que tiene un significado propio.

Para implementar el análisis léxico hemos usado **Alex**, que es una herramienta para generar analizadores léxicos en Haskell que funciona mediante la descripción de tokens en forma de expresiones regulares.

A continuación se muestra nuestro analizador léxico (las pruebas de este se encuentran en el Anexo A):

```
{
        module Tokens where
        }


        %wrapper "basic"


        $digit = 0-9
        $alpha = [a-zA-Z]
        $whiteSpace = [\ ]


        tokens :-
          $whiteSpace+                  ;
          \t+                           ;
          [\n \;]+                      { \s -> TokenNewLine}
          var                           { \s -> TokenVar }
          if                            { \s -> TokenIf }
          else                          { \s -> TokenElse }
          endif                         { \s -> TokenEndIf}
          while                         { \s -> TokenWhile}
          endwhile                      { \s -> TokenEndWhile}
          print                         { \s -> TokenPrint}
```

```
  \<                          { \s -> TokenLess }
  \>                          { \s -> TokenGreater }
  \<=                         { \s -> TokenLessEqual }
  \>=                         { \s -> TokenGreaterEqual }
  \=                          { \s -> TokenAssign }
  \+                          { \s -> TokenPlus }
  \-                          { \s -> TokenMinus }
  \*                          { \s -> TokenMultiply }
  \/                          { \s -> TokenDivide }
  $digit+                     { \s -> TokenInt (read s) }
  $alpha [$alpha $digit \_ \']* { \s -> TokenSym s }
{
-- The token type:
data Token = TokenVar
           | TokenInt Int
           | TokenSym String
           | TokenAssign
           | TokenPlus
           | TokenMinus
           | TokenMultiply
           | TokenDivide
           | TokenNewLine
           | SpaceToken String
           | TokenOr
           | TokenAnd
           | TokenLess
           | TokenGreater
           | TokenLessEqual
           | TokenGreaterEqual
           | TokenIf
           | TokenElse
           | TokenEndIf
           | TokenWhile
           | TokenEndWhile
           | TokenPrint
          deriving (Eq,Show)
scanTokens = alexScanTokens
}
```

3.2. ANÁLISIS SINTÁCTICO

El análisis sintáctico del código fuente es la segunda fase del compilador. Su principal tarea es analizar la estructura sintáctica del programa y sus componentes, y devolver mensajes de error si los hubiera.

En este caso de estudio, se ha desarrollado el analizador sintáctico adaptado a la especificación BFN mencionada anteriormente, generando un Árbol Sinntáctico Abstracto que nos permita interpretar el programa correctamente.

Para ello hemos empleado la herramienta **Happy** que consiste en un sistema generador de parser la cual toma un archivo que contiene una especificación BNF y produce código Haskell que contiene un analizador para la gramática.
A continuación se muestra el código de nuestro analizador sintáctico (las pruebas de este se encuentran en el Anexo B):

```
{
        module Grammar where
        import Tokens
        }


        %name parseTokenss
        %tokentype { Token }
        %error { parseError }


        %token
            nl          { TokenNewLine }
            int         { TokenInt $$ }
            var         { TokenVar}
            sym         { TokenSym $$}
            else        { TokenElse }
            endif       { TokenEndIf }
            while       { TokenWhile }
            endwhile    { TokenEndWhile }
            print       { TokenPrint }
            if          { TokenIf }
            '='         { TokenAssign }
            '+'         { TokenPlus }
            '-'         { TokenMinus }
            '*'         { TokenMultiply }
            '/'         { TokenDivide }
            '<'         { TokenLess }
            '>'         { TokenGreater }
            '<='        { TokenLessEqual }
            '>='        { TokenGreaterEqual}




        %left '+' '-'
        %left '*' '/'
```

```
%left '<' '>' '<=' '>='

%%

prog : exp prog              { $1 : $2 }
     | exp                   { [$1] }

exp : if_expression                                    { IfExp $1 }
    | print int_op nl                                  { Print $2 }
    | while while_cond while_body endwhile nl          { WhileExp $2 $3}
    | var sym '=' int_op nl                            { Assign $2 $4 }
    | int_op nl                                        { Tok $1 }

if_expression : if if_cond if_body endif nl                    { If $2 $3 }
              | if if_cond if_body else nl else_body endif nl   { IfElse $2 $3 $6 }

if_cond : cond nl { $1 }

while_cond : cond nl { $1 }

while_body : prog   { $1 }

if_body : prog      {$1}

else_body : prog    {$1}

int_op : sym                      { Sym $1 }
       | int                      { Int $1 }
       | int_op '+' int_op        { Plus $1 $3 }
       | int_op '-' int_op        { Minus $1 $3 }
       | int_op '*' int_op        { Multiply $1 $3 }
       | int_op '/' int_op        { Divide $1 $3 }

cond : int_op '<'  int_op            { Less $1 $3 }
     | int_op '<=' int_op            { LessEqual $1 $3 }
     | int_op '>'  int_op            { Greater $1 $3 }
     | int_op '>=' int_op            { GreaterEqual $1 $3 }
```

```
{


parseError :: [Token] -> a
parseError _ = error "Parse error"


data Exp = Assign String IntOp
         | Tok IntOp
         | IfExp IfBody
         | Print IntOp
         | WhileExp Conditional [Exp]
         deriving (Eq, Show)


data IfBody = If Conditional [Exp]
            | IfElse Conditional [Exp] [Exp]
            deriving (Eq, Show)


data Conditional = Less IntOp IntOp
          | LessEqual IntOp IntOp
          | Greater IntOp IntOp
          | GreaterEqual IntOp IntOp
          deriving (Eq, Show)


data IntOp = Int Int
           | Sym String
           | Plus IntOp IntOp
           | Minus IntOp IntOp
           | Multiply IntOp IntOp
           | Divide IntOp IntOp
           deriving (Eq, Show)


}
```

3.3. ANÁLISIS SEMÁNTICO

Por último implementamos en lenguaje **Haskell** las funcionalidades de un analizador semántico que consisten en dar significado a las construcciones del lenguaje fuente y completar los aspectos del lenguaje que es difícil de representar por una gramática libre de contexto.

Haskell nos ha dado ventajas clave para el desarrollo del intérprete, que se detallarán a continuación:

- La recursividad y las llamadas a funciones son las estructuras básicas del programa, lo que nos permite anidar todo el análisis del programa en distintas funciones.
- El uso del Pattern Matching nos ha permitido distinguir las distintas estructuras que tiene el programa, y ejecutar el programa de una forma secuencial, y sin errores.
- Los tipos de datos que tiene Haskell son inmutables, lo que impide que tengamos comportamientos no deseados a la hora de interpretar el programa.
- Haskell es un lenguaje funcional puro, con un tipado estático, por lo que siempre vas a saber que parámetros tiene cada función, y lo que va a retornar. Además, el compilador de Haskell te impide hacer más cosas de las que indicas en la declaración de las funciones, por lo que "obliga" a hacer un código limpio.
- Al ser un lenguaje funcional puro, el tratamiento de la entrada/salida es más complicado. Para solventar ese error y poder imprimir valores por pantalla y recoger información de archivos, se ha usado la Mónada IO para la impresión de valores por pantalla.
- El tratamiento de las variables (actualizar valores) se ha hecho utilizando map, para sustituir el valor en la variable indicada.

Con todo lo indicado anteriormente, se ha construido un intérprete de un lenguaje completo, desde la extracción de los tokens, hasta la interpretación de las estructuras que conforman el programa.

A continuación se muestra el código del analizador semántico (las pruebas de este se encuentran en el Anexo C):

```
module
Lib
        where


        import Grammar


        type MapValue = (String, Int)


        emptyDataStore :: [MapValue]
        emptyDataStore = []


        addToDataStore :: [MapValue] -> MapValue -> [MapValue]
        addToDataStore l val = val:l
```

```haskell
getVal :: String -> [MapValue] -> Int
getVal a store = case lookup a store of
    Just val -> val
    Nothing  -> 0


calcIntOp :: IntOp -> [MapValue] -> Int
calcIntOp (Int a) _ = a
calcIntOp (Sym a) store = getVal a store
calcIntOp (Plus  val1 val2) store = calcIntOp val1 store + calcIntOp val2 store
calcIntOp (Minus val1 val2) store = calcIntOp val1 store - calcIntOp val2 store
calcIntOp (Multiply val1 val2) store = calcIntOp val1 store * calcIntOp val2 store
calcIntOp (Divide val1 val2) store = div (calcIntOp val1 store) (calcIntOp val2
store)


changeMapValue :: MapValue -> MapValue -> MapValue
changeMapValue val1@(a, _) val2@(c, _)
    | a == c = val1
    | otherwise = val2


updateDataStore :: [MapValue] -> MapValue -> [MapValue]
updateDataStore store mapVal = map (changeMapValue mapVal) store


saveVal :: MapValue-> [MapValue] -> IO [MapValue]
saveVal (var, val) store = return ( case lookup var store of
    Just _ -> updateDataStore store (var, val)
    Nothing -> addToDataStore store (var, val))


evalCond :: Conditional -> [MapValue] -> Bool
evalCond (Less val1 val2) store = calcIntOp val1 store < calcIntOp val2 store
evalCond (LessEqual val1 val2) store = calcIntOp val1 store <= calcIntOp val2 store
evalCond (Greater val1 val2) store = calcIntOp val1 store > calcIntOp val2 store
evalCond (GreaterEqual val1 val2) store = calcIntOp val1 store >= calcIntOp val2
store



evalIfExpression :: IfBody -> [MapValue] -> IO [MapValue]
evalIfExpression (If cond part1) store = do
    if evalCond cond store
        then eval part1 store
        else return store
```

```
evalIfExpression (IfElse cond part1 part2) store = do
    if evalCond cond store
        then eval part1 store
        else eval part2 store


evalWhileExpression :: Conditional -> [Exp] -> [MapValue] -> IO [MapValue]
evalWhileExpression cond val store = do
    if evalCond cond store
        then do
            newStore <- eval val store
            evalWhileExpression cond val newStore
        else return store


printValue :: IntOp -> [MapValue] -> IO [MapValue]
printValue val store = do
    print (calcIntOp val store)
    return store


evalItem :: Exp -> [MapValue] -> IO [MapValue]
evalItem (Assign val intOp) store = do
    let res = calcIntOp intOp store
    saveVal (val, res) store
evalItem (Tok val) store = do
    let _ = calcIntOp val store
    return store
evalItem (IfExp val) store = do
    evalIfExpression val store
evalItem (Print val) store = do
    printValue val store
evalItem (WhileExp cond val) store = do
    evalWhileExpression cond val store


eval :: [Exp] -> [MapValue] -> IO [MapValue]
eval [x] store = evalItem x store
eval (x:xs) store = do
        newStore <- evalItem x store
        eval xs newStore
```

4. GUÍA DE EJECUCION DEL PROJECTO

1º—Instalar Stack.

2º—Clonar el siguiente repositorio:

```
git clone https://github.com/JCepedaVillamayor/funcional-compiler.git
```

3º—Ir a la carpeta del proyecto y construir el proyecto:

```
cd <project-path>
stack build
```

4º—Para ejecutar el proyecto lanzar el siguiente comando:

```
stack exec functional-compiler-exe
```

ANEXO A

```
module
LexerSpec
            where


            import Test.Hspec
            import Tokens


            lexerSpec :: IO()
            lexerSpec = hspec $ do
                describe "Lexer" $ do
                    it "Returns an integer token" $ do
                        let expectedTokens = [TokenInt 123]
                        scanTokens "123" `shouldBe` expectedTokens


                    it "Returns a var and a string" $ do
                        let expectedTokens = [TokenVar, TokenSym "helloWorld"]
                        scanTokens "var helloWorld" `shouldBe` expectedTokens


                    it "Returns only an new line token given enter" $ do
                        let expectedTokens = [TokenNewLine]
                        scanTokens "\n" `shouldBe` expectedTokens


                    it "Returns only a new line token given semicolon" $ do
                        let expectedTokens = [TokenNewLine]
                        scanTokens ";" `shouldBe` expectedTokens


                    it "Returns a plus and a minus token" $ do
                        let expectedTokens = [TokenPlus, TokenMinus]
                        scanTokens "+-" `shouldBe` expectedTokens

                    it "Returns a multiply and a divide token" $ do
                        let expectedTokens = [TokenDivide, TokenMultiply]
                        scanTokens "/*" `shouldBe` expectedTokens

                    it "Returns a Less and LessEqual token" $ do
                        let expectedTokens = [TokenLess, TokenLessEqual]
                        scanTokens "<<=" `shouldBe` expectedTokens
```

```
it "Returns a Greater and GreaterEqual token" $ do
    let expectedTokens = [TokenGreater, TokenGreaterEqual]
    scanTokens ">>=" `shouldBe` expectedTokens

it "Returns a print token" $ do
    let expectedTokens = [TokenPrint]
    scanTokens "print" `shouldBe` expectedTokens


it "Returns an if, else and endif token" $ do
    let expectedTokens = [TokenIf, TokenElse, TokenEndIf]
    scanTokens "if else endif" `shouldBe` expectedTokens


it "Returns a while and an endwhile token" $ do
    let expectedTokens = [TokenWhile, TokenEndWhile]
    scanTokens "while endwhile" `shouldBe` expectedTokens
```

ANEXO B

```
module
GrammarSpec
              where


              import Test.Hspec
              import Tokens
              import Grammar


              grammarSpec :: IO ()
              grammarSpec = hspec $ do
                describe "Grammar" $ do
                  it "Returns an int" $ do
                    let tokens = [TokenInt 123, TokenNewLine]
                    let expectedStructure = [Tok (Int 123)]
                    parseTokenss tokens `shouldBe` expectedStructure


                  it "Returns a var" $ do
                    let tokens = [TokenSym "abc", TokenNewLine]
                    let expectedStructure = [Tok (Sym "abc")]
                    parseTokenss tokens `shouldBe` expectedStructure

                  it "Returns a variable assignment" $ do
                    let tokens = [TokenVar, TokenSym "abc", TokenAssign, TokenInt 3,
              TokenNewLine]
                    let expectedStructure = [Assign "abc" (Int 3)]
                    parseTokenss tokens `shouldBe` expectedStructure


                  it "Returns a calculation (sum multiply)" $ do
                    let tokens = [TokenInt 3, TokenPlus, TokenInt 4, TokenMultiply, TokenInt
              5, TokenNewLine]
                    let expectedStructure = [Tok (Plus (Int 3) (Multiply (Int 4) (Int 5)))]
                    parseTokenss tokens `shouldBe` expectedStructure

                  it "Returns a calculation (minus divide)" $ do
                    let tokens = [TokenInt 3, TokenDivide, TokenInt 4, TokenMinus, TokenInt 5,
              TokenNewLine]
                    let expectedStructure = [Tok (Minus (Divide (Int 3) (Int 4)) (Int 5))]
                    parseTokenss tokens `shouldBe` expectedStructure


                  it "Returns an if condition" $ do
```

```
      let tokens = [TokenIf, TokenInt 3, TokenLess, TokenInt 4, TokenNewLine,
TokenInt 3, TokenNewLine, TokenEndIf, TokenNewLine]
      let expectedStructure = [IfExp (If (Less (Int 3) (Int 4)) [Tok (Int 3)])]
      parseTokenss tokens `shouldBe` expectedStructure

    it "Returns a print statement" $ do
      let tokens = [TokenPrint,TokenSym "a",TokenNewLine]
      let expectedStructure = [Print (Sym "a")]
      parseTokenss tokens `shouldBe` expectedStructure


    it "Returns an ifelse condition" $ do
      let tokens = [TokenIf, TokenInt 3, TokenLess,
                    TokenInt 4, TokenNewLine, TokenInt 4,
                    TokenNewLine, TokenElse, TokenNewLine,
                    TokenInt 4, TokenNewLine, TokenEndIf,
                    TokenNewLine]


      let expectedStructure = [IfExp (IfElse (Less (Int 3) (Int 4)) [Tok (Int
4)] [Tok (Int 4)])]
      parseTokenss tokens `shouldBe` expectedStructure
```

ANEXO C

```
module
LibSpec
          where


          import Test.Hspec
          import Lib
          import Grammar


          libSpec :: IO()
          libSpec = hspec $ do
              describe "Lib" $ do
                  it "Assign a value correctly" $ do
                      let expression = [(Assign "a" (Int 1))]
                      let expected = [("a", 1)]
                      obtained <- eval expression emptyDataStore
                      obtained `shouldBe` expected


                  it "Calc a value given a variable" $ do
                      let expression = [(Assign "a" (Int 1)), (Assign "a" (Plus (Sym "a") (Sym
          "a")))]
                      let expected = [("a", 2)]
                      obtained <- eval expression emptyDataStore
                      obtained `shouldBe` expected


                  it "Does anything when undeclared variable" $ do
                      let expression = [Tok (Sym "abc")]
                      let expected = []
                      obtained <- eval expression emptyDataStore
                      obtained `shouldBe` expected


                  it "Evaluate and if (condition true)" $ do
                      let expression = [IfExp (If (Less (Int 3) (Int 4)) [Assign "a" (Int
          3)])]
                      let expected = [("a", 3)]
                      obtained <- eval expression emptyDataStore
                      obtained `shouldBe` expected
```

```
        it "Evaluate and if (condition false)" $ do
            let expression = [IfExp (If (Greater (Int 3) (Int 4)) [Assign "a" (Int
3)])]
            let expected = []
            obtained <- eval expression emptyDataStore
            obtained `shouldBe` expected


        it "Evaluate and ifelse (condition true)" $ do
            let expression = [IfExp (IfElse (Less (Int 3) (Int 4)) [Assign "a" (Int
3)] [Assign "a" (Int 4)])]
            let expected = [("a", 3)]
            obtained <- eval expression emptyDataStore
            obtained `shouldBe` expected


        it "Evaluate and ifelse (condition false)" $ do
            let expression = [IfExp (IfElse (Greater (Int 3) (Int 4)) [Assign "a"
(Int 3)] [Assign "a" (Int 4)])]
            let expected = [("a", 4)]
            obtained <- eval expression emptyDataStore
            obtained `shouldBe` expected


        it "Print something!" $ do
            let expression = [Print (Int 2)]
            let expected = []
            obtained <- eval expression emptyDataStore
            obtained `shouldBe` expected

        it "While expression finished with value expected" $ do
            let expression = [Assign "a" (Int 3), WhileExp (Greater (Sym "a") (Int
0)) [Assign "a" (Minus (Sym "a") (Int 1))]]
            let expected = [("a", 0)]
            obtained <- eval expression emptyDataStore
            obtained `shouldBe` expected
```

ANEXO D

(Código main para ejecutar las pruebas anteriores)

```
1. import Grammar
2. import LexerSpec
3. import GrammarSpec
4. import LibSpec
5.
6. main :: IO ()
7. main = do
8.     lexerSpec
9.     grammarSpec
10.    libSpec
```