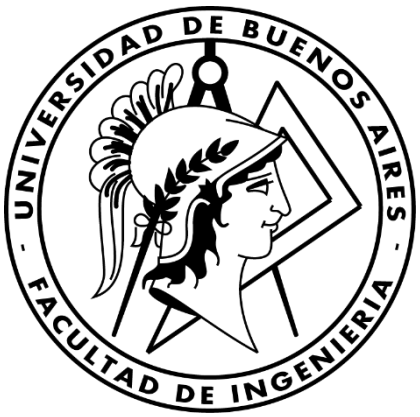
Trabajo Práctico 2:
Intérprete de Cálculo Lambda

Facultad de Ingeniería de la Universidad de Buenos Aires
Paradigmas de Programación
Cátedra Cano



Brero, Joaquín Nicolás
Padrón: 110916
email: jbrero@fi.uba.ar

Llanos Pontaut, Valentina
Padrón: 104413
email: vllanos@fi.uba.ar

Janampa Salazar, Mario
Padrón: 108344
email: mjanampa@fi.uba.ar

Novaro, Santiago Héctor
Padrón: 110938
email: snovaro@fi.uba.ar

Introducción

En el presente trabajo práctico se modela un intérprete de expresiones de cálculo Lambda. Los detalles del enunciado se expresan en CONSIGNA.md del repositorio donde se encuentra el desarrollo del programa.

Para llevar adelante el proyecto, se utilizó Scala debido a sus capacidades en el Paradigma Funcional. Dado que Scala se ejecuta en la Máquina Virtual de Java (JVM), se requiere tener instalados tanto el JDK de Java como el SDK de Scala. En este proyecto, se utilizaron las versiones 22.0.1 del JDK de Java y 3.3.3 del SDK de Scala que son las últimas versiones LTS actualmente.

El IDE utilizado fue IntelliJ (se puede obtener gratuitamente en la página oficial de JetBrains).

Estructura del proyecto

El diseño del proyecto fue dividido en tres módulos: Lexer, Parser y Reductor. Además, se creó una carpeta con casos de testeo.

Lexer

Esta sección se encarga de recibir la expresión que ingrese por input el usuario y devolver una secuencia de tokens.

```
def leerInputDeUsuario(expresionLambda: String): List[Token] = {
  val exps = expresionLambda.toList.map(x => x.toString)
  _leerInputUsuarioRec(exps)
}

private def _leerInputUsuarioRec(tokens: List[String]): List[Token] = { tokens match {
    case Nil => List()
    case x::xs => val op = x match {
        case "λ" => Lambda
        case "." => Dot
        case " " => Space
        case "(" => Lpar
        case ")" => Rpar
        case _ => Var(x)
    }
      op::_leerInputUsuarioRec(xs)
    }
}
```

La función `leerInputDeUsuario` crea una lista con cada uno de los elementos del input y le asigna un Token de los definidos a continuación:

```
sealed trait Token

case object Lambda extends Token
case object Dot extends Token
case class  Var(value: String) extends Token
case object Space extends Token
case object Lpar extends Token
case object Rpar extends Token
```

Parser

El Parser recibe la secuencia de tokens proveniente del Lexer y los transforma en un Árbol de Sintaxis Abstracta *(Abstract Syntax Tree - AST)*.

```
private def _parsearTokensRec(tokens: List[Token]): (Expresion, List[Token]) = tokens match {
  case Lambda :: Var(value) :: Dot :: restoDeTokensAParsear =>
    val (contenido, restoDeTokensRestantes) = _parsearTokensRec(restoDeTokensAParsear)
    (Abs(Variable(value), contenido), restoDeTokensRestantes)
  case Lpar :: restoDeTokensAParsear =>
    val (expresion1, tokensDespuesDeE1) = _parsearTokensRec(restoDeTokensAParsear)
    tokensDespuesDeE1 match {
      case Space :: tokensDespuesDelEspacio =>
        val (expresion2, tokensDespuesDeE2) = _parsearTokensRec(tokensDespuesDelEspacio)
        tokensDespuesDeE2 match {
          case Rpar :: remainingTokens => (Aps(expresion1, expresion2), remainingTokens)
        }
    }
  case Var(value) :: restoDeTokensAParsear => (Variable(value), restoDeTokensAParsear)
}
def parsearTokens(tokens: List[Token]): Expresion = {  Mario
  val (expr, _) = _parsearTokensRec(tokens)
  expr
}
```

La función **`parsearTokens`** va armando el AST utilizando pattern matching y las siguientes expresiones:

```
sealed trait Expresion

case class Abs(variable: Expresion,contenido: Expresion) extends Expresion
case class Aps(e1:Expresion, e2: Expresion) extends Expresion
case class Variable(e: String) extends Expresion
```

Donde **`Abs`** representa una abstracción, **`Aps`** representa una aplicación y **`Variable`** una variable sola. Por ejemplo, si quisiera obtener el AST de la expresión $(\lambda x.\lambda y.x\ z)$, el Lexer devolvería la lista $[Lpar, Lambda, Var(x), Dot, Lambda, Var(y), Dot, Var(x), Space, Var(z), Rpar]$ y el Parser interpretaría el siguiente AST:

$$Aps(Abs(Variable(x), Abs(Variable(y), Variable(x))), Variable(z))$$

El Parser es capaz también de recibir un AST y transformarlo como String de esa expresión Lambda:

```
def parsearExpresionAString(expr: Expresion): String = expr match {
  case Abs(variable:Variable,contenido) => s"λ${parsearExpresionAString(variable)}.${parsearExpresionAString(contenido)}"
  case Aps(exprs1,exprs2) => s"(${parsearExpresionAString(exprs1)} ${parsearExpresionAString(exprs2)})"
  case Variable(value) => value
}
```

En este caso **`parsearExpresionAString`** realiza un pattern matching que verifica qué tipo de Expresión es y devuelve el String correspondiente.

Reductor

El reductor recibe el AST proporcionado por el Parser y devuelve la expresión reducida. Por default devuelve la expresión reducida por call-by-name. También se lo puede setear al programa para que realice una reducción call-by-value y para que halle las variables libres de la expresión:

```
BIENVENIDO AL INTERPRETE DE CALCULO LAMBDA! :)
Actualmente se encuentra en el modo: set call-by-name
Por favor, ingrese una EXPRESION LAMBDA o SETEE UN NUEVO MODO (ingrese 'help' para conocer los modos disponibles):
help

set call-by-name: Para reducir la expresión mediante el método call-by-name
set call-by-value : Para reducir la expresión mediante el método call-by-value
set free-variables : Para hallar las variables libres de la expresión
Actualmente se encuentra en el modo: set call-by-name
Por favor, ingrese una EXPRESION LAMBDA o SETEE UN NUEVO MODO (ingrese 'help' para conocer los modos disponibles):
```

```
def reductorSegunModo(modo:String, expresion:Expresion): String = modo match {
    case FREEVARIABLES =>
      // Funciones para hacer el free variable
      val fvariables = freeVariables(expresion)
      "Variables libres: " + imprimirListaVariables(fvariables)
    case CALLBYNAME =>
      // Funciones para hacer reduccion call-by-name
      val reducida = reductorCallByName(expresion)
      "Expresión reducida: " + parsearExpresionAString(reducida)
    case CALLBYVALUE =>
      // Funciones para hacer reduccion call-by-value
      val reducida = reductorCallByValue(expresion)
      "Expresión reducida: " + parsearExpresionAString(reducida)
}
```

Free Variables

Halla las variables libres de la expresión Lambda. La función **`freeVariables`** recibe el AST y lo desglosa teniendo en cuenta que cuando se encuentra con una Abstracción, va a llamar a la función **`quitarVariable`** y cuando sea una Aplicación llama a **`unirVariables`**. Esto se debe a que las abstracciones definen variables ligadas, porque están definidas por parámetro. Las que no sean variables ligadas serán entonces variables libres:

```
def freeVariables(exp: Expresion): List[String] = {
  exp match
    case Variable(v) => List(v)
    case Aps(exp1, exp2) => unirVariables(freeVariables(exp1), freeVariables(exp2))
    case Abs(vLigada, contenido) => quitarVariable(freeVariables(vLigada), freeVariables(contenido))
}
def unirVariables(vExp1: List[String], vExp2: List[String]): List[String] = {
  vExp1.appendedAll(vExp2)
    .groupBy(x => x)
    .keys
    .toList
}
def quitarVariable(vLigada: List[String], variables: List[String]): List[String] = {
  variables
    .filter(x => !vLigada.contains(x))
}
```

Reducción Call-by-name y Call-by-value

Para esto se implementaron las funciones de **`conversionAlfa`** y **`reduccionBeta`** que implementarían ambas reducciones:

`conversionAlfa`

```
def conversionAlfa(expresion: Expresion, vLigada: String): Expresion = {
  val nuevoNombre = vLigada + SIGNOCONVALFA
  expresion match {
    case Variable(valor) if valor == vLigada => Variable(nuevoNombre)
    case Abs(variable, contenido) if parsearExpresionAString(variable) == vLigada =>
      Abs(Variable(nuevoNombre), conversionAlfa(contenido, vLigada))
    case Abs(variable, contenido) => Abs(variable, conversionAlfa(contenido, vLigada))
    case Aps(e1, e2) => Aps(conversionAlfa(e1, vLigada), conversionAlfa(e2, vLigada))
    case _ => expresion
  }
}

def conversionAlfaGeneral(expresion: Expresion, VLibres2daExpresion: List[String]): Expresion = {
  expresion match {
    case Variable(valor) => Variable(valor)
    case Abs(variable, contenido) if VLibres2daExpresion.contains(parsearExpresionAString(variable)) =>
      val nuevoNombreVLigada = parsearExpresionAString(variable) + SIGNOCONVALFA
      Abs(Variable(nuevoNombreVLigada), conversionAlfaGeneral(conversionAlfa(contenido, parsearExpresionAString(variable)), VLibres2daExpresion))
    case Abs(variable, contenido) => Abs(variable, conversionAlfaGeneral(contenido, VLibres2daExpresion))
    case Aps(e1, e2) => Aps(conversionAlfaGeneral(e1, VLibres2daExpresion), conversionAlfaGeneral(e2, VLibres2daExpresion))
  }
}
```

`reduccionBetaCBN`

```
def reduccionBetaCBN(vLigada: String, expresion: Expresion, reemplazo: Expresion): Expresion = expresion match {
  case Variable(valor) if valor == vLigada => reemplazo
  case Variable(valor) => expresion
  case Abs(variable, contenido) if parsearExpresionAString(variable) == vLigada => expresion
  case Abs(variable, contenido) =>
    Abs(variable, reduccionBetaCBN(vLigada, contenido, reemplazo))
  case Aps(e1, e2) => Aps(reduccionBetaCBN(vLigada, e1, reemplazo), reduccionBetaCBN(vLigada, e2, reemplazo))
}
```

`reduccionBetaCBV`

```
def reduccionBetaCBV(expr: Expresion, variable: Variable, reemplazo: Expresion): Expresion = expr match {
  case Variable(valor) if valor == variable.e => reemplazo
  case Abs(vLigada, contenido) if vLigada != variable => Abs(vLigada, reduccionBetaCBV(contenido, variable, reemplazo))
  case Aps(e1, e2) => Aps(reduccionBetaCBV(e1, variable, reemplazo), reduccionBetaCBV(e2, variable, reemplazo))
  case _ => expr
}
```

Luego, cada llamada hará uso de estas funciones según lo necesite:

CBN

```
def callByName(e1: Expresion, e2: Expresion): Expresion = e1 match {
    case Variable(v) => Aps(e1, reductorCallByName(e2))
    case Abs(vLigada, contenido) =>
      val nuevoContenido = conversionAlfaGeneral(contenido, freeVariables(e2))
      reductorCallByName(reduccionBetaCBN(parsearExpresionAString(vLigada), nuevoContenido, e2))
    case Aps(e1, e3) =>
      val expresion = callByName(e1, e3)
      expresion match {
        case Aps(exp1, exp2) => Aps(expresion, reductorCallByName(e2))
        case _ => callByName(expresion, e2)
      }
}
def reductorCallByName(expresion: Expresion): Expresion = expresion match {
  case Variable(_) => expresion
  case Aps(exp1, exp2) => callByName(reductorCallByName(exp1), exp2)
  case Abs(vLigada, contenido) => Abs(vLigada, reductorCallByName(contenido))
}
```

CBV

```
def reductorCallByValue(expr: Expresion): Expresion = expr match {
  case Aps(Abs(variable:Variable, contenido), arg) =>
    val reducedArg = reductorCallByValue(arg)
    val nuevoContenido = conversionAlfaGeneral(contenido, freeVariables(reducedArg))
    reductorCallByValue(reduccionBetaCBV(nuevoContenido, variable, reducedArg))
  case Abs(vLigada, contenido) => Abs(vLigada, reductorCallByValue(contenido))
  case Aps(e1, e2) =>
    val reducedE1 = reductorCallByValue(e1)
    val reducedE2 = reductorCallByValue(e2)
    (reducedE1, reducedE2) match {
      case (Abs(variable, contenido), argumento) => reductorCallByValue(Aps(reducedE1, reducedE2))
      case _ => Aps(reducedE1, reducedE2)
    }
  case Variable(_) => expr
}
```

Conclusión

La propuesta de la cátedra de resolver el programa en tres módulos fue de gran ayuda para la organización del proyecto. Propuso una buena metodología de ordenamiento de tareas y permitió al equipo una buena dinámica de trabajo.

Una de las dificultades encontradas con el paradigma fue la legibilidad de código que resta realizar la mayoría de operaciones recursivamente.

Links de interés

- Repositorio de github del proyecto: https://github.com/Paradigmas-1C2024/Inter-Lambda
- Guía de instalación de JetBrains: https://www.jetbrains.com/help/idea/installation-guide.html