DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Analizador Sintáctico y Semántico para λ^{bn}

Trabajo Práctico

Primer Cuatrimestre de 2017, Grupo "Jon y Meñique" — Teoría de Lenguajes

Integrante	LU	Correo electrónico
Costa, Manuel José Joaquín	035/14	manucos94@gmail.com
Maddonni, Axel Ezequiel	200/14	axel.maddonni@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo se presenta una implementación de un analizador sintáctico y semántico para un subconjunto del cálculo lambda tipado, sobre booleanos y naturales. El mismo analiza si una expresión sigue la sintaxis esperada, y en caso de que sea válida, indica el tipo y el valor de la misma.

Se soportan los siguientes términos:
M ::= x | true | false | if M1 then M2 else M3 | \x:T.M | M1 M2 | 0 | succ(M) | pred(M) | iszero(M)

y los siguientes tipos:
T ::= Bool | Nat | T1 → T2

Los valores para el lenguaje son los siguientes:
V ::= true | false | \x:T.M | n

2. Gramática LALR No Ambigua

Rule 0	S' → exp
Rule 1	exp → app
Rule 2	exp → IF exp THEN exp ELSE exp
Rule 3	exp → LAMBDA VAR : type . exp
Rule 4	app → app term
Rule 5	app → term
Rule 6	term → nat
Rule 7	term → bool
Rule 8	term → VAR
Rule 9	nat → ZERO
Rule 10	nat → SUC (exp)
Rule 11	nat → PRED (exp)
Rule 12	bool → BOOL
Rule 13	bool → IS_ZERO (exp)
Rule 14	term → (exp)
Rule 15	type → atype
Rule 16	type → atype TO type
Rule 17	atype → TBOOL
Rule 18	atype → TNAT
Rule 19	atype → (type)

3. Tokens

TOKEN	**Regex**
ZERO	0
SUC	$succ$
PRED	$pred$
BOOL	$true \vert false$
IS_ZERO	$isZero$
IF	if
THEN	$then$
ELSE	$else$
TO	$->$
TBOOL	$Bool$
TNAT	Nat
VAR	$[a-z]$
LAMBDA	\

Además, se utilizan los siguientes tokens literales: '(', ')', ':', '.'

4. Implementación

Para realizar la implementación se crearon clases para representar los distintos tipos de términos generados por la gramática. A medida que se parsean las expresiones, se van generando los objetos correspondientes según el tipo de término. Una vez parseada la cadena, se imprime el *value* y *type* de la expresión generada. Estas funciones calculan el valor y el tipo recursivamente, bindeando los tipos según las lambdas encontradas y resolviendo las aplicaciones. Esto no es posible hacerlo a medida que se parsea la cadena ya que podrían encontrarse variables libres que no se pueden evaluar hasta que no se les bindee su valor más adelante en la ejecución. En las clases se definen además funciones para imprimir los términos por pantalla, y chequeo de tipos.

4.1. Lexer

```
#! coding: utf-8
import ply.lex as lex
from classes import *

literals = ['(', ')', ':', '.']

tokens = [
   'IF',
   'THEN',
   'ELSE',
   'TO',
   'BOOL',
   'VAR',
   'ZERO',
   'SUC',
   'PRED',
   'IS_ZERO',
   'TBOOL',
   'TNAT',
   'LAMBDA'
]

t_ignore = ' '

def t_ZERO(t):
   r'0'
   t.value = Nat(0)
   return t

def t_SUC(t):
   r'succ'
   return t

def t_PRED(t):
   r'pred'
   return t

def t_BOOL(t):
   r'true|false'
   t.value = Bool(t.value == 'true')
   return t

def t_IS_ZERO(t):
   r'isZero'
   return t
```

```
def t_IF(t):
  r'if'
  return t

def t_THEN(t):
  r'then'
  return t

def t_ELSE(t):
  r'else'
  return t

def t_TO(t):
  r'\->'
  return t

def t_TBOOL(t):
  r'Bool'
  return t

def t_TNAT(t):
  r'Nat'
  return t

def t_VAR(t):
  r'[a-z]'
  t.value = Var(t.value)
  return t

def t_LAMBDA(t):
  r'\\'
  return t

# Error handling rule
def t_error(t):
    sys.stderr.write("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

def apply_lexer(string):
    """Aplica el lexer al string dado."""
    lexer.input(string)

    return list(lexer)
```

4.2. Parser

```
# coding=utf-8

"""Parser LALR de lambdaCalc."""
import ply.yacc as yacc
from .lexer import tokens
from classes import *

def p_exp_app(p):
    '''exp : app '''
    p[0] = p[1]

def p_exp_if(p):
    '''exp : IF exp THEN exp ELSE exp '''
    p[0] = IfThenElse(p[2], p[4], p[6])

def p_exp_abs(p):
    '''exp : LAMBDA VAR ':' type '.' exp '''
    p[0] = Abstraction(p[2], p[4], p[6])

def p_app(p):
    '''app : app term '''
    p[0] = Application(p[1], p[2])

def p_app_term(p):
    '''app : term '''
    p[0] = p[1]

def p_term_base(p):
    '''term : nat
            | bool
            | VAR '''
    p[0] = p[1]

def p_nat_zero(p):
    '''nat : ZERO'''
    p[0] = p[1]

def p_nat_suc(p):
    '''nat : SUC '(' exp ')' '''
    p[0] = Suc(p[3])

def p_nat_pred(p):
    '''nat : PRED '(' exp ')' '''
    p[0] = Pred(p[3])
```

```
def p_bool(p):
    '''bool : BOOL'''
    p[0] = p[1]

def p_bool_iszero(p):
    '''bool : IS_ZERO '(' exp ')' '''
    p[0] = IsZero(p[3])

def p_term_exp(p):
    '''term : '(' exp ')' '''
    p[0] = p[2]

def p_type_atype(p):
    '''type : atype '''
    p[0] = p[1]

def p_type_to(p):
    '''type : atype TO type '''
    p[0] = Type([p[1].typesArray(), p[3].typesArray()])

def p_atype_base(p):
    '''atype : TBOOL
             | TNAT '''
    p[0] = Type([p[1]])

def p_atype_type(p):
    '''atype : '(' type ')' '''
    p[0] = Type(p[2].typesArray())

def p_error(p):
    sys.stderr.write("Syntax Error. Asegurese que la cadena se ingreso
    entre commilas dobles.\n")
    sys.exit(1)

# Build the parser
parser = yacc.yacc(debug=True)

def apply_parser(str):
    return parser.parse(str)
```

4.3. Classes

```
# coding=utf-8

import traceback
import sys

### Definimos las clases para los tipos de terminos ###

class Expression(object):
  def bindType(self, varName, varType):
    return self

  def bindValue(self, varName, varValue):
    return self

class Nat(Expression):
  def __init__(self, n):
    self._value = n

  def __repr__(self):
    start = ''
    end = ''
    for x in xrange(0,self._value):
      start += 'succ('
      end += ')'
    return start + str(0) + end

  def __eq__(self, other):
    return self._value == other._value

  def value(self):
    return self

  def type(self):
    return Type(['Nat'])

  def suc(self):
    return Nat(self._value + 1)

  def pred(self):
    if self._value > 0:
      return Nat(self._value - 1)
    else:
      return Nat(0)

  def isZero(self):
```

```
    return Bool(self.value() == Nat(0))

  def isFree(self):
    return False

class Bool(Expression):
  def __init__(self, b):
    self._value = b

  def __repr__(self):
    if self._value:
      return 'true'
    else:
      return 'false'

  def value(self):
    return self

  def type(self):
    return Type(['Bool'])

  def isTrue(self):
    if self._value:
      return True
    else:
      return False

  def isFree(self):
    return False

class Suc(Expression):
  def __init__(self, exp):
    self._exp = exp

  def __repr__(self):
    return 'succ(' + str(self._exp) + ')'

  def value(self):
    if (not (self._exp.type().typesArray() == ['Nat'])):
      sys.stderr.write('ERROR succ espera un valor de tipo Nat' + "\n")
      sys.exit(1)
    return self._exp.value().suc()

  def type(self):
    if (not (self._exp.type().typesArray() == ['Nat'])):
      sys.stderr.write('ERROR succ espera un valor de tipo Nat' + "\n")
      sys.exit(1)
```

```
    return Type(['Nat'])

  def bindValue(self, varName, value):
    self._exp = self._exp.bindValue(varName, value)
    return self

  def bindType(self, varName, valueType):
    self._exp = self._exp.bindType(varName, valueType)
    return self

  def isFree(self):
    return self._exp.isFree()

class Pred(Expression):
  def __init__(self, exp):
    self._exp = exp

  def __repr__(self):
    if (self.isFree()):
      return 'pred(' + str(self._exp) + ')'
    else:
      return str(self.value())

  def value(self):
    if (not (self._exp.type().typesArray() == ['Nat'])):
      sys.stderr.write('ERROR pred espera un valor de tipo Nat' + "\n")
      sys.exit(1)
    return self._exp.value().pred()

  def type(self):
    if (not (self._exp.type().typesArray() == ['Nat'])):
      sys.stderr.write('ERROR pred espera un valor de tipo Nat' + "\n")
      sys.exit(1)
    return Type(['Nat'])

  def bindValue(self, varName, value):
    self._exp = self._exp.bindValue(varName, value)
    return self

  def bindType(self, varName, valueType):
    self._exp = self._exp.bindType(varName, valueType)
    return self

  def isFree(self):
    return self._exp.isFree()

class IsZero(Expression):
```

```
  def __init__(self, exp):
    self._exp = exp

  def __repr__(self):
    if (self._exp.isFree()):
      return 'isZero(' + str(self._exp) + ')'
    else:
      return str(self.value())

  def value(self):
    if (self._exp.type().typesArray() == ['Nat']):
      return self._exp.value().isZero()
    else:
      sys.stderr.write('ERROR isZero espera un valor de tipo Nat' + "\n")
      sys.exit(1)

  def type(self):
    if (self._exp.type().typesArray() == ['Nat']):
      return Type(['Bool'])
    else:
      sys.stderr.write('ERROR isZero espera un valor de tipo Nat' + "\n")
      sys.exit(1)

  def bindValue(self, varName, value):
    self._exp = self._exp.bindValue(varName, value)
    return self

  def bindType(self, varName, valueType):
    self._exp = self._exp.bindType(varName, valueType)
    return self

  def isFree(self):
    return self._exp.isFree()

class IfThenElse(Expression):
  def __init__(self, cond, ifTrue, ifFalse):
    self._cond = cond
    self._ifTrue = ifTrue
    self._ifFalse = ifFalse

  def __repr__(self):
    if (self._cond.isFree()):
      return 'if ' + str(self._cond) + ' then ' + str(self._ifTrue)
      + ' else ' + str(self._ifFalse)
    else:
      return str(self.value())
```

```
  def value(self):
    if (not (self._ifTrue.type() == self._ifFalse.type())):
      sys.stderr.write('ERROR: Las dos opciones del if deben tener
      el mismo tipo' + "\n")
      sys.exit(1)
    if (not (self._cond.type().typesArray() == ['Bool'])):
      sys.stderr.write('ERROR: La condicion del if debe ser de tipo
      Bool' + "\n")
      sys.exit(1)
    if (self._cond.value().isTrue()):
      return self._ifTrue.value()
    else:
      return self._ifFalse.value()

  def type(self):
    if (not (self._ifTrue.type() == self._ifFalse.type())):
      sys.stderr.write('ERROR: Las dos opciones del if deben tener
      el mismo tipo' + "\n")
      sys.exit(1)
    if (not (self._cond.type().typesArray() == ['Bool'])):
      sys.stderr.write('ERROR: La condicion del if debe ser de tipo
      Bool' + "\n")
      sys.exit(1)
    return self._ifTrue.type()

  def bindValue(self, varName, value):
    self._cond = self._cond.bindValue(varName, value)
    self._ifTrue = self._ifTrue.bindValue(varName, value)
    self._ifFalse = self._ifFalse.bindValue(varName, value)
    return self

  def bindType(self, varName, valueType):
    self._cond = self._cond.bindType(varName, valueType)
    self._ifTrue = self._ifTrue.bindType(varName, valueType)
    self._ifFalse = self._ifFalse.bindType(varName, valueType)
    return self

  def isFree(self):
    return self._cond.isFree() or self._ifFalse.isFree() or
    self._ifTrue.isFree()

class Var(Expression):
  def __init__(self, name):
    self._name  = name
    self._value = None
    self._type  = None
```

```
  def __repr__(self):
    if (self.isFree()):
      return self._name
    else:
      return str(self.value())

  def value(self):
    if (self.isFree()):
      sys.stderr.write('ERROR: El termino no es cerrado
      (' + self._name + ' esta libre))' + "\n")
      sys.exit(1)
    else:
      return self._value.value()

  def type(self):
    if self._type is not None:
      return self._type
    else:
      sys.stderr.write('ERROR: El termino no es cerrado
      (' + self._name + ' esta libre))' + "\n")
      sys.exit(1)

  def bindValue(self, varName, value):
    if (str(self._name) == str(varName)):
      self._value = value
      self._type = value.type()
    return self

  def bindType(self, varName, valueType):
    if (str(self._name) == str(varName)):
      self._type = valueType
    return self

  def isFree(self):
    return self._value is None

class Abstraction(Expression):
  def __init__(self, varname, vartype, body):
    self._var = varname
    self._type = vartype
    self._body = body
    self._body.bindType(varname, self._type)

  def __repr__(self):
    return '\\' + str(self._var) + ':' + str(self._type)
    + '.' + str(self._body)
```

```
  def paramName(self):
    return self._var

  def paramType(self):
    return self._type

  def bodyType(self):
    return self._body.type()

  def apply(self, value):
    self._body.bindValue(self._var, value)
    return self._body.value()

  def type(self):
    return Type([self.paramType().typesArray(),
    self.bodyType().typesArray()])

  def value(self):
    return self

  def bindValue(self, varName, value):
    self._body = self._body.bindValue(varName, value)
    return self

  def bindType(self, varName, valueType):
    self._body = self._body.bindType(varName, valueType)
    return self

  def isFree(self):
    return True

class Application(Expression):
  def __init__(self, absTerm, paramTerm):
    self._absTerm = absTerm
    self._paramTerm = paramTerm

  def __repr__(self):
    if (self.isFree()):
      return str(self._absTerm) + ' ' + str(self._paramTerm)
    else:
      return str(self.value())

  def value(self):
    absTermValue = self._absTerm.value()
    paramTermValue = self._paramTerm
    if (isinstance(absTermValue, Abstraction) and
      paramTermValue.type() == absTermValue.paramType()):
```

```
      return absTermValue.apply(paramTermValue)
    else:
      sys.stderr.write('ERROR: La parte izquierda de la aplicacion
      ('+str(self._absTerm.value())+') no es una funcion con
      dominio en '+str(self._paramTerm.type())+"\n")
      sys.exit(1)

  def type(self):
    return self._absTerm.type().applyType(self._paramTerm.type())

  def bindValue(self, varName, value):
    self._absTerm = self._absTerm.bindValue(varName, value)
    self._paramTerm = self._paramTerm.bindValue(varName, value)
    return self

  def bindType(self, varName, valueType):
    self._absTerm = self._absTerm.bindType(varName, valueType)
    self._paramTerm = self._paramTerm.bindType(varName, valueType)
    return self

  def isFree(self):
    return self._absTerm.isFree() or self._paramTerm.isFree()

class Type:
  def __init__(self, typesArray):
    while (len(typesArray) == 1 and isinstance(typesArray[0], list)):
      typesArray = typesArray[0]
    self._typesArray = []
    for aType in typesArray:
      if (isinstance(aType, list) and len(aType) == 1):
        aType = aType[0]
      self._typesArray.append(aType)

  def __repr__(self):
    if (len(self.typesArray()) > 1):
      return "->".join(map(lambda x:
        '('+str(Type([x]))+')' if isinstance(x, list)
        else str(Type([x]))
        , self._typesArray))
    else:
      elem = self.typesArray()[0]
      return str(elem)

  def __eq__(self, other):
    if isinstance(other, self.__class__):
      if (len(self._typesArray) == 1 and
      isinstance(self._typesArray[0], list)):
```

```
        myType = self._typesArray[0]
      else:
        myType = self._typesArray
      if (len(other._typesArray) == 1 and
      isinstance(other._typesArray[0], list)):
        otherType = other._typesArray[0]
      else:
        otherType = other._typesArray
      return myType == otherType
    return False

  def typesArray(self):
    return self._typesArray

  def applyType(self, paramType):
    if (len(self.typesArray()) == 1
    and isinstance(self.typesArray()[0], list)):
      return Type(self.typesArray()[0]).applyType(paramType)

    if (len(paramType.typesArray()) == 1
    and isinstance(paramType.typesArray()[0], list)):
      return self.applyType(paramType.typesArray()[0])

    absType = self.typesArray()

    if (absType[0] == paramType.typesArray() and len(absType) > 1):
      return Type(absType[1:len(absType)])

    if (len(absType) <= len(paramType.typesArray())):
      sys.stderr.write('ERROR de tipos en aplicacion,
      no se puede aplicar un termino de tipo ' + str(self)
      + ' a otro de tipo ' + str(paramType) + "\n")
      sys.exit(1)

    for i in xrange(0,len(paramType.typesArray())):
      myType = absType[i]
      otherType = paramType.typesArray()[i]
      if (myType != otherType):
        sys.stderr.write('ERROR de tipos en aplicacion,
        no se puede aplicar un termino de tipo ' + str(self)
        + ' a otro de tipo ' + str(paramType) + "\n")
        sys.exit(1)
    return Type(absType[len(paramType.typesArray()):len(absType)])
```

5. Modo de Uso

El programa se ejecuta con el comando:

./CLambda "EXPRESION"

El comando recibe una cadena entre comillas dobles con la expresión a evaluar, y devuelve por stdout el resultado de la evaluación y su tipo. Si no se especifica la cadena en la llamada, se esperará recibirla por stdin. En caso de que hubiera algún inconveniente al ejecutar el programa, éste termina y se muestran los errores por stderr.

Para correr el parser se deberán instalar los requerimientos de PLY 3.6, herramienta seleccionada que genera código Python:

pip install -r –user requirements.txt

El archivo requirements.txt se encuentra adjunto junto con el código.

6. Casos de Prueba

ENTRADA	**SALIDA**
0	0:Nat
true	true:Bool
if true then 0 else false	ERROR: Las dos opciones del if deben tener el mismo tipo
\x:Bool.if x then false else true	\x:Bool.if x then false else true:Bool→Bool
\x:Nat.succ(0)	\x:Nat.succ(0):Nat→Nat
\z:Nat.z	\z:Nat.z:Nat→Nat
x	ERROR: El término no es cerrado (x está libre))
(\x:Bool.succ(x)) true	ERROR succ espera un valor de tipo Nat
0 0	ERROR: La parte izquierda de la aplicación (0) no es una función con dominio en Nat
succ(succ(pred(0))) succ(succ(0)):Nat	
\x:Nat.succ(x)	\x:Nat.succ(x):Nat→Nat
\x:Nat→Nat.\y:Nat.(\z:Bool.if z then x y else 0)	\x:Nat→Nat.\y:Nat.\z:Bool.if z then x y else 0:(Nat→Nat)→(Nat→(Bool→Nat)
(\x:Nat→Nat.\:Nat.(\z:Bool.if z then x y else 0)) (\j:Nat.succ(j)) 0 true	succ(0):Nat
(\x:Bool→Bool.x true)	\x:Bool→Bool.x true:(Bool→Bool)→Bool
\n:Nat.isZero(n)	\n:Nat.isZero(n):Nat→Bool
\x:Nat.isZero(y)	ERROR: El término no es cerrado (y está libre))
\x:Nat→Nat.((\y:Nat.isZero(y)) 0)	\x:Nat→Nat.true:(Nat→Nat)→Bool
(\x:Nat→Nat.(\y:Nat.x succ(0))) (\x:Nat.succ(x))	\y:Nat.succ(succ(0)):Nat→Nat
(\x:Nat.x) (\z:Bool.succ(0)) true	ERROR: La parte izquierda de la aplicación (\x:Nat.x) no es una función con dominio en Bool→Nat
(\x:Nat.x) ((\:Bool.succ(0)) true)	succ(0):Nat
\x:Bool.\y:Bool.\z:Bool.x	\x:Bool.\y:Bool.\z:Bool.x :Bool→(Bool→(Bool→Bool))
\x:Nat.\y:(Bool→Nat)→Bool.\z:Bool.0	\x:Nat.\y:(Bool→Nat)→Bool.\z:Bool.0 :Nat→(((Bool→Nat)→Bool)→(Bool→Nat))

7. Conclusiones

El proceso de implementación del parser fue un proceso iterativo, comenzando desde expresiones más simples hasta expresiones más complejas. A medida que aumentaba la complejidad de las mismas, nos encontrábamos con nuevos problemas al evaluar o asociar las expresiones correctamente, que nos llevaban a realizar cambios en la gramática o en el modo en que procesamos cada producción. Para el conjunto de casos de tests generados, se obtuvieron los resultados deseables, aunque notamos que hubo varios casos bordes aparecieron mientras avanzábamos con la implementación. Como conclusión, la implementacion de parsers nos resultó un tema muy interesante pero de arduo y constante trabajo.