DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Analizador Sintáctico y Semántico para Cálculo Lambda

Teoría de Lenguajes

Integrante	LU	Correo electrónico
Martinez, Federico		fedomartinez@hotmail.com
Medina Raco, Juan		juan.medina.raco@gmail.com
Harari, Ignacio		nachotee@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

El objetivo de este trabajo es crear un analizador sintáctico y semántico para un subconjunto del cálculo lambda tipado sobre booleanos y naturales (λ bn). Este debe poder parsear validando la sintaxis, y en caso de ser correcta evaluar la expresión y mostrar el resultado de su evaluación. Se dan como alternativas 2 tipos de parsers (LL1, y LALR) para la resolución del problema. Una vez definida la herramienta o tipo de parser a utilizar, se definió una gramática, junto con sus tokens, y luego se definieron las estructuras necesarias para poder realizar un análisis semántico de las expresiones que resultasen validas a nivel sintáctico.

2. Desarrollo

2.1. Tokens

Dado el problema planteado se definieron los siguientes tokens para la gramática encargada de parsear el conjunto de expresiones q conforma el cálculo lambda. Las expresiones regulares utilizadas para parsear cada uno de los tokens se pueden con mayor claridad en la sección de Código Fuente, en el archivo lexer.py.

- **TRUE** r'(?i)true'
- **FALSE** r'(?i)false'
- **ZERO** r'0'
- **ISZERO** r'(?i)iszero'
- **LPAREN** r'
  ('
- **RPAREN** r'
)'
- **SUCC** r'(?i)succ'
- **PRED** r'(?i)pred'
- **IF** r'(?i)if'
- **THEN** r'(?i)then'
- **ELSE** r'(?i)else'
- **ARROW** r'-¿'
- **LAMBDA** r'
  '
- **TYPE** r'(Nat—Bool)'
- **DOT** r'¡'
- **DOBLEDOT** r':'
- **VARIABLE** r'([jvwxyz])'

2.2. Gramática

Una vez definidos los tokens, se trabajó sobre varias versiones de gramáticas. Como resultante de un proceso iterativo en el cual se fueron adaptando, e incluso llegando a rehacer de cero las grámaticas, se llegó a la gramática que se presenta a continuación.

```
Rule 0     S' -> expression
Rule 1     expression -> IF expression THEN expression ELSE expression
Rule 2     expression -> nat
Rule 3     expression -> bool
Rule 4     expression -> LPAREN lambda RPAREN subexp
Rule 5     expression -> lambda
Rule 6     expression -> variable
Rule 7     expression -> variable variable
Rule 8     subexp -> LPAREN lambda RPAREN subexp
Rule 9     subexp -> nat subexp
Rule 10    subexp -> bool subexp
Rule 11    subexp -> <empty>
Rule 12    lambda -> LAMBDA variable DOBLEDOT type DOT expression
Rule 13    atomictype -> TYPE
Rule 14    type -> atomictype ARROW type
Rule 15    type -> atomictype
Rule 16    variable -> VARIABLE
Rule 17    bool -> ISZERO LPAREN expression RPAREN
Rule 18    bool -> TRUE
Rule 19    bool -> FALSE
Rule 20    nat -> SUCC LPAREN expression RPAREN
Rule 21    nat -> PRED LPAREN expression RPAREN
Rule 22    nat -> ZERO
Rule 23    expression -> nat nat
```

El formato en el que se presenta la gramática es el utilizado por la libreria PLY para mostrar el resultado de las reglas de parseo que se definieron. O sea, es la gramática generada en el archivo parser.out de la librería

2.3. Tipo de gramática

Dado el problema planteado, la creación de un analizador sintáctico y semántico para el cálculo lambda, y las alternativas en cuanto al tipo de parser que se podía utilizar, se optó por la implementación de un parser de tipo LALR generado con la herramienta PLY, sobre el lenguaje de programacion Python.

2.4. Implementación de la solución

Como se puede ver en el punto anterior donde se lista la gramatica utilizada para intepretar expresiones del calculo lambda, se definio que las expresiones de tipo abstraccion deben ser encerradas entre parentesis para poder ser agregarlas a una aplicacion. Esta fue la forma de solucionar los conflictos shif/reduce y reduce/reduce q se producian con gramaticas alternativas, asi como los problemas relacionados con la interpretacion de la abstraccion y su alcance.

2.5. Requerimientos de software

Para poder ejecutar el parser es necesario tener instalado Python 2.7 junto con libreria ply(version 3.7). Es necesario, además, ejecutar el archivo requirements.txt con pip tal cual como se hizo en el taller. El trabajo se desarrolló sobre una plataforma linux y para poder testear una expresión en una terminal esta debe ser pasada por stdin al script 'CLambda.sh'.

EJ: echo ."^{ex}presion lambda"— ./CLambda.sh

2.6. Casos de prueba

Se trabajó sobre un conjunto de expresiones que nos permitió ir construyendo el parser de manera iterativa. Desarrollamos los casos con una estrategia bottom-up, arrancando por los casos más básicos y simples como

true, 0 o succ(0) y, una vez que confirmamos que funcionaran correctamente, pasamos a probar expresiones más complejas como las Lambdas y las aplicaciones.

A continuación, mostramos algunos de los casos que testeamos. El conjunto completo de nuestros tests se puede observar en la sección de Código Fuente, en el archivo test.py.

```
Testeos orientados a tokens y casos básicos.
  0
  true
  false
  isZero(0)
  succ(0)
Testeos orientados a la semántica
  succ(succ(succ(succ(0))))
  pred(pred(succ(0)))
  if true then 0 else succ(pred(0))
  if true then 0 else false
  \z:Nat.z 0
  \z:Nat.if z then 0 else succ(0) true
  (\x:Bool.(\z:Nat.if x then z else succ(z))) 0 true
Testeos orientados a las aplicaciones y lambdas
  \z:Nat.z 0
  (\x:Bool.(\z:Nat.if x then z else succ(z))) 0 true
  \x:Nat->Nat. \y:Nat. (\z:Bool. if z then x y else 0)
  \x:Nat.x
  \z:Nat.if z then 0 else succ(0) iszero(0)
  (\z:Nat.iszero((\x:Nat. pred(x)) z)) succ(0)
```

2.7. Conclusiones

Se pudo experimentar el desarrollo de un parser y sus complejidades. Dada la herramienta utilizada(PLY) resulto facil generar una gramatica para poder realizar el anailisis sintactico de las expresiones que forman el calculo Lamda. Lo que la herramienta no soluciona y en lo que se encontraron dificultades es en el analisis semantico de las expresiones. En ese punto fue necesario replantear varias veces la implementacion para ir mejorandola y realizando abstracciones para poder describir e interpretar correctamente el significado de las expresiones mas complejas. El analisis de las expresiones de tipo Lambda y las aplicaciones, tanto en el tipo como de valor resultante fueron los puntos en los que se debio reconsiderar desiciones y mejorar la implementacion varias veces. Tambien la forma en la que se iban arrastrando los contextos y como se van ligando las variables en cada expresion y sub expresion permitieron ejercitar la tematica de gramatica de atributos.

3. Código Fuente

3.1. lexer.py

```
tokens = [
    'TRUE',
    'FALSE',
    'ZERO',
    'ISZERO',
    'LPAREN',
    'RPAREN',
    'SUCC',
    'PRED',
    'IF',
    'THEN',
    'ELSE',
    'ARROW',
    'LAMBDA',
    'TYPE',
    'DOT',
    'DOBLEDOT',
    'VARIABLE'
]

t_LPAREN = r'\\('
t_RPAREN = r'\\)'

def t_VARIABLE(t):
    r'([jvwxyz])'
    return t

def t_ARROW(t):
    r'->'
    return t

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

def t_TYPE(t):
    r'(Nat|Bool)'
    return t

def t_DOT(t):
    r'\.'
    return t

def t_DOBLEDOT(t):
    r':'
    return t

def t_IF(t):
    r'(?i)if'
    return t

def t_THEN(t):
    r'(?i)then'
```

```
    return t

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

def t_SUCC(t):
    r'(?i)succ'
    return t

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

def t_TRUE(t):
    r'(?i)true'
    return t

def t_FALSE(t):
    r'(?i)false'
    return t

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

def t_ISZERO(t):
    r'(?i)iszero'
    return t

t_ignore = ' \t\n'
```

3.2. parser.py

```
import ply.yacc as yacc
from .lexer import tokens
import objetoParseado as op
import sys,traceback

def p_if_exp_then_exp_else_exp(p):
    'expression : IF expression THEN expression ELSE expression'
    p[0] = op.construirIfThenElse(p[2], p[4], p[6])

def p_exp_nat(p):
    'expression : nat'
    p[0] = p[1]

def p_exp_bool(p):
    'expression : bool'
    p[0] = p[1]

def p_exp_apply(p):
    'expression : LPAREN lambda RPAREN subexp'
    p[0] = op.construirAplicacion(p[2], p[4])
```

```
def p_exp_expression_lambda(p):
    'expression : lambda'
    p[0] = p[1]

def p_exp_variable_expresion(p):
    'expression : variable'
    p[0] = p[1]

def p_exp_variable_variable(p):
    'expression : variable variable'
    p[0] = op.construirAplicacion(p[1], [p[2]])

def p_subexp_paren_lambda(p):
    'subexp : LPAREN lambda RPAREN subexp'
    p[0] = [p[2]] + p[4]

def p_subexp_nat(p):
    'subexp : nat subexp'
    p[0] = [p[1]] + p[2]

def p_subexp_bool(p):
    'subexp : bool subexp'
    p[0] = [p[1]] + p[2]

def p_subexp_empty(p):
    'subexp : '
    p[0] = []

def p_exp_lambda(p):
    'lambda : LAMBDA variable DOBLEDOT type DOT expression'
    p[0] = op.construirLambda(p[2], p[4], p[6])

def p_exp_atomic_type(p):
    'atomictype : TYPE'
    p[0] = p[1]

def p_exp_type_arrow(p):
    'type : atomictype ARROW type'
    p[0] = '('+p[1]+p[2]+p[3]+')'

def p_exp_type(p):
    'type : atomictype'
    p[0] = p[1]

def p_exp_variable(p):
    'variable : VARIABLE'
    p[0] = op.construirVariable(p[1])

def p_exp_iszero(p):
    'bool : ISZERO LPAREN expression RPAREN'
    p[0] = op.construirIsZero(p[3])

def p_exp_true(p):
    'bool : TRUE'
    p[0] = op.construirBool(True)

def p_exp_false(p):
    'bool : FALSE'
    p[0] = op.construirBool(False)
```

```
def p_exp_succ(p):
    'nat : SUCC LPAREN expression RPAREN'
    p[0] = op.construirSucc(p[3])

def p_exp_pred(p):
    'nat : PRED LPAREN expression RPAREN'
    p[0] = op.construirPred(p[3])

def p_exp_zero(p):
    'nat : ZERO'
    p[0] = op.construirZero()

def p_exp_nat_nat(p):
    'expression : nat nat'
    p[0] = op.construirError('La parte izquierda de la aplicacion ('+p[1].getExpresion()+') no es u


def p_error(p):
    print "Hubo un error en el parseo."
    print p
    parser.restart()

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

def apply_parser(string):
    try:
        parseado = parser.parse(string)
        if parseado is not None and parseado.getTipo() == 'Var':
            return 'Error: el termino no es cerrado ('+parseado.getExpresion()+' esta libre)'

        if parseado is not None and not isinstance(parseado.getExpresion(), op.EError) and not isin
            return str(parseado.getExpresion()) + ':'+ parseado.getTipo()
        elif not isinstance(parseado, op.EError):
            return parseado.getExpresion().getExpresion() + parseado.getExpresion().getTipo()
        else:
            return str(parseado.getExpresion()) + parseado.getTipo()

    except:
        traceback.print_exc(file=sys.stdout)
```

3.3. objetoParseado.py

```
def construirBool(valor):
    return EBool(valor)

def construirZero():
    return EZero()

def construirIsZero(param):
    return EIsZero(param)

def construirSucc(param):
    return ESucc(param)
```

```
def construirPred(param):
    return EPred(param)

def construirVariable(var):
    return EVariable(var)

def construirIfThenElse(cond, exprIf, exprElse):
    return EIfThenElse(cond, exprIf, exprElse)

def construirLambda(var, tipo, expr):
    return ELambda(var, tipo, expr)

def construirAplicacion(lamb, param):
    return EAplicacion(lamb, param)

def construirError(mensaje):
    return EError(mensaje)


class objetoParseado(object):

    def __init__(self, expresion, tipo, valor):
        self.expresion = expresion
        self.tipo = [tipo]
        self.valor = valor

    def getExpresion(self):
        return self.expresion

    def getTipo(self):
        return self.tipo

    def getValor(self):
        return self.valor


class EAplicacion(objetoParseado):

    def __init__(self, lamb, param):
        self.lamb = lamb
        self.param = param

    def getLamb(self):
        return self.lamb

    def setParam(self, param):
        self.param = param

    def getExpresion(self):
        if self.getValor().getTipo() != 'Indefinido':
            return self.getValor().getExpresion()

        return  '%s %s' % (self.lamb.getExpresion(), self.param[0].getExpresion())

    def getTipo(self):
        if self.getValor().getTipo() != 'Indefinido':
            return self.getValor().getTipo()

        return self.lamb.getTipo()
```

```
    def getValor(self, scope={}):
        if len(self.param) == 0 and len(scope) == 0:
            return EValor('Indefinido', None)

        s = {
            'assignment': self.param[0].getValor(scope),
            'passDown': self.param[1:]
        }
        s.update(scope)

        if self.lamb.getTipo() == 'Var' and self.lamb.getValor(scope).getValor() is not None:
            return self.lamb.getValor(scope).getValor().getValor(s)

        return self.lamb.getValor(s)


class ELambda(objetoParseado):
    def __init__(self, var, tipo, expr):
        self.var = var
        self.tipo = tipo
        self.expr = expr

    def getExpresion(self):
        return  '\\%s:%s.%s' % (self.var.getExpresion(), self.tipo, self.expr.getExpresion())

    def getTipo(self):
        if self.expr.getTipo() == 'Var':
            return '%s->%s' % (self.tipo, self.tipo)

        return '%s->%s' % (self.tipo, self.expr.getTipo())

    def getValor(self, scope={}):
        if scope.has_key('assignment'):
            param = scope['assignment']

            if isinstance(self.expr, EAplicacion):
                self.expr.setParam(scope['passDown'])

            scope.pop('assignment', None)
            scope.pop('passDown', None)

            newScope = scope
            newScope.update({self.var.getExpresion(): param})

            if self.expr.getValor(newScope).getTipo() != 'Indefinido':
                return self.expr.getValor(newScope)

        return EValor('Lambda', self)


class EVariable(objetoParseado):
    def __init__(self, var):
        self.var = var

    def getExpresion(self):
        return self.var

    def getTipo(self):
```

```
        return 'Var'

    def getValor(self, scope={}):
        if scope.has_key(self.var):
            if scope[self.var].getTipo() != 'Indefinido':
                return scope[self.var]

        return EValor('Indefinido', None)


class EIfThenElse(objetoParseado):
    def __init__(self, cond, exprIf, exprElse):
        self.cond = cond
        self.exprIf = exprIf
        self.exprElse = exprElse

    def getExpresion(self):
        if isinstance(self.getValor(), EError):
            return self.getValor()

        if self.getValor().getTipo() != 'Indefinido' and self.getValor().getTipo() != 'Var':
            return self.getValor().getExpresion()

        return  'if %s then %s else %s' % (self.cond.getExpresion(), self.exprIf.getExpresion(), se

    def getTipo(self):
        if self.cond.getTipo() == 'Var':
            if self.exprIf.getTipo() == 'Var':
                return self.exprElse.getTipo()

            return self.exprIf.getTipo()

        return self.exprIf.getTipo()

    def getValor(self, scope={}):
        if self.cond.getTipo() == 'Var' and self.cond.getValor(scope).getValor() is None:
            return self.cond

        if not isinstance(self.cond.getValor(scope).getValor(), bool):
            return EError('la condicion del if no es un valor booleano')

        if self.exprIf.getValor(scope).getTipo() != self.exprElse.getValor(scope).getTipo():
            if self.exprIf.getValor(scope).getTipo() != 'Indefinido' and self.exprElse.getValor(sco
                return EError('las dos opciones del if deben tener el mismo tipo')

        if self.cond.getValor(scope).getValor():
            if self.exprIf.getValor(scope).getTipo() != 'Indefinido':
                return self.exprIf.getValor(scope)
        else:
            if self.exprElse.getValor(scope).getTipo() != 'Indefinido':
                return self.exprElse.getValor(scope)

        return EValor('Indefinido', 'None')


class EPred(objetoParseado):
    def __init__(self,param):
        self.hijo = param
```

```
    def getExpresion(self):
        if self.getValor().getTipo() == 'Nat':
            return self.getValor().getValor()

        return 'pred(%s)' % self.hijo.getExpresion()

    def getTipo(self):
        return 'Nat'

    def getValor(self, scope={}):
        if self.hijo.getValor(scope).getTipo() == 'Nat':
            if self.hijo.getValor(scope).getValor() > 0:
                return EValor('Nat', self.hijo.getValor(scope).getValor()-1)
            else:
                return EValor('Nat', self.hijo.getValor(scope).getValor())

        return EValor('Indefinido', None)


class ESucc(objetoParseado):
    def __init__(self, param):
        self.hijo = param

    def getExpresion(self):
        if self.getValor().getTipo() == 'Nat':
            return self.getValor().getExpresion()

        return 'succ(%s)' % self.hijo.getExpresion()

    def getTipo(self):
        return 'Nat'

    def getValor(self, scope={}):
        if self.hijo.getValor(scope).getTipo() == 'Nat':
            return EValor('Nat', self.hijo.getValor(scope).getValor()+1)

        return EValor('Indefinido', None)


class EIsZero(objetoParseado):

    def __init__(self, param):
        self.hijo = param

    def getExpresion(self):
        if self.getValor().getTipo() == 'Bool':
            return self.getValor().getExpresion()

        if self.getValor().getTipo() == 'Var':
            return 'iszero('+self.getValor().getExpresion()+')'

        return EError('iszero espera un valor de tipo Nat')

    def getTipo(self):
        return 'Bool'

    def getValor(self, scope={}):
        if self.hijo.getValor(scope).getTipo() == 'Nat':
            return EValor('Bool', not bool(self.hijo.getValor(scope).getValor()))
```

```
        if self.hijo.getTipo() == 'Var':
            return self.hijo

        return EValor('Indefinido', 'None')


class EZero(objetoParseado):
    def __init__(self):
        pass

    def getExpresion(self):
        return '0'

    def getTipo(self):
        return 'Nat'

    def getValor(self, scope={}):
        return EValor('Nat', 0)


class EBool(objetoParseado):
    def __init__(self, valor):
        self.valor = valor

    def getExpresion(self):
        if self.valor:
            return 'true'
        else:
            return 'false'

    def getTipo(self):
        return 'Bool'

    def getValor(self, scope={}):
        return EValor('Bool', bool(self.valor))


class EValor(object):
    def __init__(self, tipo, valor):
        self.tipo = tipo
        self.valor = valor

    def getExpresion(self):
        if self.tipo == 'Nat':
            return self.printNat()
        elif self.tipo == 'Bool':
            return self.printBool()
        elif self.tipo == 'Lambda':
            return self.valor.getExpresion()
        else:
            return self.printIndefinido()

    def getTipo(self):
        if self.tipo == 'Lambda':
            return '('+self.valor.getTipo()+')'

        return self.tipo
```

```
    def getValor(self):
        return self.valor

    def printBool(self):
        return  'true' if self.valor else 'false'

    def printNat(self):
        num = '0'
        for _ in range(max(0, int(self.valor))):
            num = 'succ(%s)' % num
        return '%s' % num

    def printIndefinido(self):
        return 'Indefinido'

    def __str__(self):
        return 'DEBUG: %s : %s ' % (self.tipo, self.valor)


class EError(object):
    def __init__(self, mensaje):
        self.mensaje = mensaje

    def getExpresion(self):
        return 'Error: '

    def getTipo(self):
        return self.mensaje

    def getValor(self):
        return 'Explota'
```

3.4. test.py

```
from analyzer import parse

print 'Testeando casos basicos:\n'

assert parse('0') == '0:Nat'
print 'Caso: 0 OK!'
assert parse('true') == 'true:Bool'
print 'Caso: true OK!'
assert parse('false') == 'false:Bool'
print 'Caso: false OK!'
assert parse('iszero(0)') == 'true:Bool'
print 'Caso: iszero(0) OK!'
assert parse('iszero(true)') == 'Error: iszero espera un valor de tipo Nat'
print 'Caso: iszero(true) OK!'
assert parse('succ(0)') == 'succ(0):Nat'
print 'Caso: succ(0) OK!'
assert parse('pred(0)') == '0:Nat'
print 'Caso: pred(0) OK!'
assert parse('iszero(pred(0))') == 'true:Bool'
print 'Caso: iszero(pred(0)) OK!'
assert parse('iszero(succ(0))') == 'false:Bool'
print 'Caso: iszero(succ(0)) OK!'
```

```
assert parse('if true then 0 else succ(0)') == '0:Nat'
print 'Caso: if true then 0 else succ(0) OK!'
assert parse('if true then false else true') == 'false:Bool'
print 'Caso: if true then false else true OK!'

print '\nTesteando casos combinados:\n'

assert parse('succ(succ(0))') == 'succ(succ(0)):Nat'
print 'Caso: succ(succ(0)) OK!'
assert parse('succ(pred(0))') == 'succ(0):Nat'
print 'Caso: succ(pred(0)) OK!'
assert parse('pred(succ(0))') == '0:Nat'
print 'Caso: pred(succ(0)) OK!'
assert parse('pred(pred(0))') == '0:Nat'
print 'Caso: pred(pred(0)) OK!'
assert parse('if if false then true else false then false else true') == 'true:Bool'
print 'Caso: if if false then true else false then false else true OK!'
assert parse('if iszero(0) then succ(0) else 0') == 'succ(0):Nat'
print 'Caso: if iszero(0) then succ(0) else 0 OK!'

print '\nTesteando casos complejos:\n'

assert parse('pred(pred(succ(succ(0))))') == '0:Nat'
print 'Caso: pred(pred(succ(succ(0)))) OK!'
assert parse('succ(pred(pred(succ(succ(0)))))') == 'succ(0):Nat'
print 'Caso: succ(pred(pred(succ(succ(0))))) OK!'
assert parse('if iszero(succ(0)) then 0 else if true then pred(succ(0)) else succ(0)') == '0:Nat'
print 'Caso: if iszero(succ(0)) then 0 else if true then pred(succ(0)) else succ(0) OK!'
assert parse('succ(if iszero(0) then succ(0) else 0)') == 'succ(succ(0)):Nat'
print 'Caso: succ(if iszero(0) then succ(0) else 0) OK!'

print '\nTesteando lambdas sin aplicacion:\n'

assert parse('\\z:Nat.z') == '\\z:Nat.z:Nat->Nat'
print 'Caso: \\z:Nat.z OK!'
assert parse('\\z:Nat.succ(z)') == '\\z:Nat.succ(z):Nat->Nat'
print 'Caso: \\z:Nat.succ(z) OK!'
assert parse('\\z:Nat.pred(z)') == '\\z:Nat.pred(z):Nat->Nat'
print 'Caso: \\z:Nat.pred(z) OK!'
assert parse('\\z:Nat.true') == '\\z:Nat.true:Nat->Bool'
print 'Caso: \\z:Nat.true OK!'
assert parse('\\z:Nat.iszero(z)') == '\\z:Nat.iszero(z):Nat->Bool'
print 'Caso: \\z:Nat.iszero(z) OK!'
assert parse('\\z:Nat.if z then 0 else succ(0)') == '\\z:Nat.if z then 0 else succ(0):Nat->Nat'
print 'Caso: \\z:Nat.\\y:Bool.if y then z else succ(z) OK!'
print parse('\\z:Nat.\\y:Bool.if y then z else succ(z)')
assert parse('\\z:Nat.\\y:Bool.if y then z else succ(z)') == '\\z:Nat.\\y:Bool.if y then z else suc
print 'Caso: \\z:Nat.\\y:Bool.if y then z else succ(z) OK!'


print '\nTesteando lambdas con aplicacion:\n'

print parse('(\\z:Nat.z) 0')
assert parse('(\\z:Nat.z) 0') == '0:Nat'
print 'Caso: (\\z:Nat.z) 0 OK!'
assert parse('(\\z:Nat.succ(z)) 0') == 'succ(0):Nat'
print 'Caso: (\\z:Nat.succ(z)) 0 OK!'
assert parse('(\\z:Nat.succ(succ(z))) 0') == 'succ(succ(0)):Nat'
print 'Caso: (\\z:Nat.succ(succ(z))) 0 OK!'
```

```
assert parse('(\\z:Nat.pred(succ(succ(z)))) 0') == 'succ(0):Nat'
print 'Caso: (\\z:Nat.pred(succ(succ(z)))) 0 OK!'
assert parse('(\\z:Nat.z)  true') == 'true:Bool'
print 'Caso: (\\z:Nat.z)  true OK!'
assert parse('(\\z:Nat.z)  false') == 'false:Bool'
print 'Caso: (\\z:Nat.z)  false OK!'
assert parse('(\\z:Nat.iszero(z)) 0') == 'true:Bool'
print 'Caso: (\\z:Nat.iszero(z)) 0 OK!'
assert parse('(\\z:Nat.iszero(succ(z))) 0') == 'false:Bool'
print 'Caso: (\\z:Nat.iszero(succ(z))) 0 OK!'
assert parse('(\\z:Nat.if z then 0 else succ(0)) false') == 'succ(0):Nat'
print 'Caso: (\\z:Nat.if z then 0 else succ(0)) false OK!'
assert parse('(\\z:Nat.if z then 0 else succ(0)) true') == '0:Nat'
print 'Caso: (\\z:Nat.if z then 0 else succ(0)) true OK!'
assert parse('(\\z:Nat.if z then 0 else succ(0)) iszero(0)') == '0:Nat'
print 'Caso: (\\z:Nat.if z then 0 else succ(0)) iszero(0) OK!'
assert parse('(\\z:Nat.if z then 0 else succ(0)) iszero(succ(0))') == 'succ(0):Nat'
print 'Caso: (\\z:Nat.if z then 0 else succ(0)) iszero(succ(0)) OK!'
assert parse('(\\z:Nat.if iszero(z) then succ(0) else succ(z)) succ(pred(succ(0)))') == 'succ(succ(
print 'Caso: (\\z:Nat.if iszero(z) then succ(0) else succ(z)) succ(pred(succ(0))) OK!'

print '\nTesteando lambdas con y sin aplicacion complejos:\n'

assert parse('(\\x:Bool.(\\z:Nat.if x then z else succ(z))) true 0') == '0:Nat'
print 'Caso: (\\x:Bool.(\\z:Nat.if x then z else succ(z))) true 0 OK!'
assert parse('(\\x:Nat->Nat.(\\y:Nat. (\\z:Bool.if z then x y else 0))) (\\j:Nat.succ(j)) succ(succ
print 'Caso: (\\x:Nat->Nat.(\\y:Nat. (\\z:Bool.if z then x y else 0))) (\\j:Nat.succ(j)) succ(succ(
assert parse('(\\z:Nat.iszero(pred(z))) succ(0)') == 'true:Bool'
print 'Caso: (\\z:Nat.iszero(pred(z))) succ(0) OK!'
assert parse('(\\x:Nat->Nat.x) (\\j:Nat.succ(j))') == '\\j:Nat.succ(j):(Nat->Nat)'
print 'Caso: (\\x:Nat->Nat.x) (\\j:Nat.succ(j)) OK!'
assert parse('(\\x:Nat->Nat.(\\y:Nat. (\\z:Bool.if z then x y else 0))) (\\j:Nat.succ(j)) succ(succ
print 'Caso: (\\x:Nat->Nat.(\\y:Nat. (\\z:Bool.if z then x y else 0))) (\\j:Nat.succ(j)) succ(succ(

print '\nTesteando ejemplos enunciado:\n'

assert parse('0') == '0:Nat'
print 'Caso: 0 OK!'
assert parse('true') == 'true:Bool'
print 'Caso: true OK!'
assert parse('if true then 0 else false') == 'Error: las dos opciones del if deben tener el mismo t
print 'Caso: if true then 0 else false OK!'
assert parse('\\x:Bool.if x then false else true') == '\\x:Bool.if x then false else true:Bool->Boo
print 'Caso: \\x:Bool.if x then false else true OK!'
assert parse('\\x:Nat.succ(0)') == '\\x:Nat.succ(0):Nat->Nat'
print 'Caso: \\x:Nat.succ(0) OK!'
assert parse('\\z:Nat.z') == '\\z:Nat.z:Nat->Nat'
print 'Caso: \\z:Nat.z OK!'
assert parse('succ(succ(succ(0)))') == 'succ(succ(succ(0))):Nat'
print 'Caso: succ(succ(succ(0))) OK!'
assert parse('x') == 'Error: el termino no es cerrado (x esta libre)'
print 'Caso: x OK!'
assert parse('succ(succ(pred(0)))') == 'succ(succ(0)):Nat'
print 'Caso: succ(succ(pred(0))) OK!'
assert parse('\\x:Nat.succ(x)') == '\\x:Nat.succ(x):Nat->Nat'
print 'Caso: \\x:Nat.succ(x) OK!'
assert parse('0 0') == 'Error: La parte izquierda de la aplicacion (0) no es una funcion con domini
print 'Caso: 0 0 OK!'
assert parse('\\x:Nat->Nat.\\y:Nat.\\z:Bool.if z then x y else 0') == '\\x:(Nat->Nat).\\y:Nat.\\z:B
```

```
print 'Caso: \\x:Nat->Nat.\\y:Nat.\\z:Bool.if z then x y else 0 OK!'
assert parse('(\\x:Nat->Nat.(\\y:Nat. (\\z:Bool.if z then x y else 0))) (\\j:Nat.succ(j)) succ(succ
print 'Caso: (\\x:Nat->Nat.(\\y:Nat. (\\z:Bool.if z then x y else 0))) (\\j:Nat.succ(j)) succ(succ(

print '\nTesting finalizado, todos los casos correctos!\n'
```