



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Analizador sintáctico y semántico para lambda cálculo

Teoría de los lenguajes
Primer Cuatrimestre 2017

Integrante	LU	Correo electrónico
Blundi, Solange	336/10	solange.blundi@gmail.com
Inzaghi Pronesti, Patricio Ezequiel	255/11	patricio.inzaghi@gmail.com
Guerson, Matias Carlos	925/10	matias.guerson@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	3
2. Análisis Sintáctico	4
2.1. Lexer	4
2.2. Gramática	5
2.2.1. Ambigüedades, Correctitud y Completitud (Pequeña intuición)	5
3. Análisis Semántico	7
3.1. Chequeo de tipos	7
3.2. Evaluación de la expresión	7
4. Conclusiones	9
5. Implementación	10
5.1. Estructura del proyecto	10
5.1.1. CLambda	10
5.1.2. main.py	10
5.1.3. Makefile	10
5.1.4. README.md	10
5.1.5. requirements.txt	10
5.1.6. lambdacalc/expressions.py	11
5.1.7. lambdacalc/types.py	11
5.1.8. lambdacalc/lexer.py	11
5.1.9. lambdacalc/parser.py	11
5.1.10. tests/basicExpressionsTest.py	11
5.1.11. tests/complexExpressionsTest.py	11
5.1.12. tests/typesTest.py	11
5.2. Ejecución del programa	11
5.3. Código	12
5.3.1. main.py	12
5.3.2. lexer.py	13
5.3.3. parser.py	15
5.3.4. types.py	17
5.3.5. expressions.py	18
5.4. Tests	23
5.4.1. basicExpressionsTest.py	23
5.4.2. complexExpressionsTest.py	24
5.4.3. typesTest.py	26

1. Introducción

Este trabajo tiene como objetivo la implementación de un analizador sintáctico y semántico para el cálculo lambda extendido para Booleanos y Naturales λ^{bn} . Para poder implementar dicho analizador, es necesario llevar a cabo las siguientes tareas:

- Definición de una gramática no ambigua que permita generar el lenguaje propuesto. En nuestro caso se optó por una gramática LALR.
- Generación de un lexer que permita convertir la cadena de entrada de una secuencia de símbolos en una cadena de tokens. Estos últimos serán los símbolos terminales de nuestra gramática.
- Implementación de un analizador sintáctico: Mediante el uso de la gramática y el lexer, en esta etapa se va a implementar un parser que no solo permitirá determinar si una cadena pertenece o no al lenguaje, sino que también se generará el Abstract Syntax Tree, el cual será utilizado posteriormente para realizar un análisis semántico.
- Implementación de un analizador semántico: En este trabajo, el analizador semántico tendrá dos objetivos:
 - Realizar el chequeo de tipos, esto es, dada una expresión del cálculo lambda decidir si la misma tipa correctamente o no.
 - Realizar la evaluación de la expresión. En este paso la idea es reducir la expresión a un valor.

Durante este trabajo utilizaremos la herramienta *PLY*, que nos permite definir las reglas para el lexer, las del parser y luego generar el árbol sintáctico. Una vez generado el mismo, podremos utilizarlo para el análisis semántico de la expresión dada.

2. Análisis Sintáctico

Para este trabajo decidimos utilizar una gramática LALR (Look-Ahead, Left to Right, RightMost derivation), es decir, definimos una gramática que al ser procesada por un parser LALR no presenta conflictos de ningún tipo.

Los parsers LALR son de la familia de parsers ascendentes. En particular, son una simplificación de los parsers LR(1), y esto se debe a que unifica los estados que poseen el mismo kernel uniendo todos sus look-ahead.

Esta simplificación tiene como objetivo reducir el espacio de memoria utilizado por los LR(1), problema que en algún caso puede llegar a ser crítico. Los parsers LALR utilizan la misma cantidad de estados que los parsers SLR pudiendo obtener un mayor poder expresivo que estos.

Es necesario aclarar que, al realizar esta unificación, se pueden introducir conflictos del tipo reduce/reduce que no existían en la versión del parser LR(1), pero es importante notar también, que no se pueden introducir otro tipo de conflictos si estos no existían previamente.

2.1. Lexer

Antes de definir la gramática, procederemos a mostrar las reglas utilizadas para generar el lexer. La herramienta *PLY* nos permite definir las reglas mediante la utilización de expresiones regulares.

A continuación detallamos las reglas generadas:

Cuadro 1: Lexer Rules

Token	Expresión Regular	Símbolo Terminal
BACKSLASH	\\	\
VAR	[a-z][a-zA-Z0-9]*	Nombres de las variables
COLON	\:	:
TYPE	(Bool Nat)	Tipos de variables
DOT	\.	.
IF	if	if
THEN	then	then
ELSE	else	else
SUCC	succ	succ
PRED	pred	pred
IS_ZERO	isZero	isZero
TRUE	true	true
FALSE	false	false
ZERO	0	0
L_PAREN	\((
R_PAREN	\))
ARROW	->	→
ignore	'\t '	Espacios y tabs

Detalles de implementación:

A la hora de definir las reglas del lexer, teníamos como intención que los nombres de las variables pudieran ser cualquier string que comenzara con una letra en minúscula y siguiera con cualquier otra letra, ya sea mayúscula o minúscula, o números. El problema que nos topamos en ese momento, es que de esta forma podrían producirse conflictos entre los nombres de las variables y algunos símbolos del lenguaje, como el *true*, *false*, *isZero*, *pred* y *succ*.

Para evitar estos posibles conflictos, se implementó una solución ¹ que antes de crear un token correspondiente a una variable, chequea que la cadena no se corresponda con un token reservado. En caso de conflicto, opta por crear el token reservado y no una variable.

¹<https://stackoverflow.com/questions/5022129/ply-lex-parsing-problem>

2.2. Gramática

Una vez definidas las reglas del lexer y utilizando los tokens generados, definimos la siguiente gramática:

```

S → expression
expression → absExpression
absExpression → BACKSLASH varExpression COLON typeExpression DOT expression
absExpression → ifExpression
ifExpression → IF expression THEN expression ELSE expression
ifExpression → appExpression
appExpression → appExpression termExpression
appExpression → termExpression
termExpression → varExpression
termExpression → functionExpression
varExpression → VAR
functionExpression → SUCC L_PAREN expression R_PAREN
functionExpression → PRED L_PAREN expression R_PAREN
functionExpression → IS_ZERO L_PAREN expression R_PAREN
functionExpression → valExpression
valExpression → TRUE
valExpression → FALSE
valExpression → ZERO
valExpression → L_PAREN expression R_PAREN
typeExpression → TYPE ARROW typeExpression
typeExpression → TYPE

```

2.2.1. Ambigüedades, Correctitud y Completitud (Pequeña intuición)

Si bien hablar de algunos de estos conceptos requiere de demostraciones formales que van por fuera del alcance de la materia, podemos dar algunas pequeñas nociones o intuiciones:

Ambigüedades:

Mediante la herramienta *PLY* pudimos generar el parser LALR y, al no presentarse ningún conflicto, podemos asegurar que nuestra gramática no es ambigua.

Completitud:

Lamentablemente, la gramática dada no es completa, es decir, hay cadenas válidas del lenguaje que nuestro parser rechazará. Un ejemplo de esto se puede dar en las aplicaciones. Por como fue definida nuestra gramática, no se aceptarán cadenas que sean aplicaciones, donde la expresión de la derecha sea algo menos precedente que la aplicación misma. Es decir, por ejemplo, expresiones como las siguientes serán rechazadas:

$$(\backslash f : Nat \rightarrow Nat.f\ 0) \backslash x : Nat.x$$

$$(\backslash f : Nat \rightarrow Nat.f\ 0) \text{ if true then } \backslash x : Nat.x \text{ else } \backslash y : Nat.succ(y)$$

Aún así, estos casos pueden llegar a expresarse mediante la utilización de paréntesis, por ejemplo, siguiendo los casos recientes, las expresiones que nuestro parser si acepta deberían estar expresadas de la siguiente forma:

$$(\backslash f : Nat \rightarrow Nat.f\ 0) (\backslash x : Nat.x)$$

$$(\backslash f : Nat \rightarrow Nat.f\ 0) (if\ true\ then\ \backslash x : Nat.x\ else\ \backslash y : Nat.succ(y))$$

Correctitud:

Para demostrar correctitud tendríamos que poder asegurar que absolutamente todas las cadenas que nuestro parser acepte sean válidas en el lenguaje.

Si bien no estamos en condiciones de realizar dicha demostración, hemos realizado un conjunto de tests sobre el parser que nos permite intuir que la gramática dada sería correcta.

3. Análisis Semántico

En esta parte del trabajo tenemos que realizar el análisis semántico de la expresión recibida. Como mencionamos previamente, dicho análisis consta de dos etapas:

- Chequeo de tipos
- Evaluación de la expresión

Una vez realizado el análisis sintáctico, si la cadena de entrada no fue rechazada, no solo sabemos que es una cadena válida del lenguaje, sino que también disponemos del Abstract Syntax Tree.

Luego, la idea consiste básicamente en diseñar e implementar clases y/o estructuras de datos que serán los valores de cada nodo del árbol y que nos permitirán realizar operaciones más complejas y de un modo más simple e intuitivo sobre el árbol.

Para esto, definimos dos grandes clases:

```
class Type
class Expression
```

La clase *Type* nos permite generar instancias que representan los tipos básicos Bool y Nat. Para poder tener una abstracción del tipo de datos *flecha* (llamada por nosotros tipo función), creamos una subclase de *Type* llamada *FunctionType*, de esta manera podemos generar instancias del tipo función que a su vez pueden tener como dominio o imagen tipos básicos, como otras funciones.

En cuanto a la clase *Expression*, la idea es definir una clase que debieran extender todas las clases que quieran representar algún tipo de expresión válida dentro del lenguaje.

Para esto, todas las subclases deberán implementar los métodos *value* y *type*, que serán los métodos que nos ayudaran a obtener tanto el tipo de cada expresión como su valor.

3.1. Chequeo de tipos

Para poder realizar el chequeo de tipos, cada nodo del árbol (es decir cada expresión), tiene que poder determinar cuál es su propio tipo pero también esto puede depender de su o sus subexpresiones, tanto para definir su tipo, como para poder realizar las validaciones de tipado pertinentes.

La idea entonces es que cada subexpresión pueda retornar su propio tipo, de esta forma, se podría ir determinando el tipo de toda la expresión y validando si el tipado es correcto. Pero esto solo no alcanza, recordemos que dentro del lenguaje disponemos de términos que son las variables. Estos términos heredan su tipo por el contexto y dado que *PLY* no dispone de atributos heredados, la solución que decidimos llevar a cabo es que, en cada llamado al método *type*, se pase como parámetro el contexto Γ .

Como decisión de implementación entendimos que con un diccionario alcanzaba para representar dicho contexto. Definimos como clave a la expresión de la variable y como valor su tipo.

Así, cuando se quiera saber el tipo de una variable, esta deberá chequear si se encuentra definida en el contexto o no. En caso de no estarlo, significará que la misma se encuentra libre, arrojando entonces la excepción pertinente.

3.2. Evaluación de la expresión

Una vez que sabemos que la expresión es sintácticamente válida y que hemos podido validar que el tipado es correcto, es hora de poder reducir la expresión a un valor.

Para esto, el procedimiento a realizar es similar al del chequeo de tipos. Utilizando el árbol sintáctico, comenzaremos desde la raíz a pedir a los nodos hijos sus valores e ir realizando las reducciones necesarias.

Nuevamente nos encontramos con el problema que representan las variables, dado que tenemos que poder determinar cuál es la expresión que esa variable representa.

Sin embargo, en este caso el problema es un poco más complejo, y esto se debe a que, si tenemos una expresión que es una abstracción, el valor es la abstracción en si misma, y en este caso no nos interesa reemplazar las variables dentro.

Pero si la abstracción es parte de una aplicación, entonces debemos poder reducir toda la expresión al valor correspondiente, pero esta vez tendremos que poder determinar el valor de las variables.

Para poder lidiar con este problema, decidimos definir un nuevo método en las abstracciones, llamado *apply*, de esta forma, si solo se quiere saber el valor de la abstracción, está retornará una referencia a si misma (Recordar que la abstracción en sí ya es un valor del cálculo lambda). En cambio, si tenemos una aplicación, lo que haremos es, dentro del método *value* de la misma, invocar a la funcionalidad *apply* de la expresión de la izquierda, que por haber superado el chequeo de tipos, sabemos que es una abstracción.

El método *apply* recibirá como parámetros tanto el contexto como el valor de la expresión de la derecha de la aplicación. Lo que hacemos es agregar al contexto la variable y su valor, para luego invocar el método *value* de la subexpresión con el nuevo contexto Γ . Una vez más, cuando se invoque el método *value* de la variable, esta deberá buscar su expresión en el contexto y pedirle el valor, en caso de no estar definida deberá arrojar la excepción pertinente.

4. Conclusiones

Durante este trabajo pudimos llevar a cabo un analizador sintáctico y semántico para el Cálculo Lambda tipado, pasando por distintas etapas en el proceso de diseño e implementación.

Comenzamos pensando una **gramática** que no sea **ambigua**, que preserve las **asociatividades y precedencias** que el lenguaje exigía y que pudieramos generar un **parser LALR sin conflictos**.

Luego tuvimos que definir las reglas del **lexer**, para lo que utilizamos **expresiones regulares**, y de esta forma generamos los **tokens** que luego serían necesarios para el parser.

En la etapa del analizador sintáctico, plasmamos tanto la gramática diseñada como los tokens y con todo eso definimos las reglas de parsing.

Al finalizar en análisis sintáctico, la ventaja de estos parsers es que no solo podemos determinar si una **cadena ingresada es válida o no**. A diferencia de los **autómatas**, con los parsers, en esta etapa tenemos generado el **Abstract Syntax Tree**, que será fundamental para el paso siguiente, el análisis semántico de la expresión recibida.

Llegando a la etapa del análisis semántico, teníamos como objetivo tanto, chequear que la expresión tipe correctamente, como poder reducirla a un valor. Aquí fue fundamental pensar, diseñar e implementar distintas clases que nos permitieran llevar a cabo las tareas recientemente mencionadas de una manera clara y simple. En esta etapa nos encontramos con problemas como por ejemplo no disponer de **atributos heredados** y tener que encontrar la forma de poder, aún así, completar satisfactoriamente el análisis semántico.

Como conclusión final, durante este trabajo tuvimos que plasmar gran parte de los contenidos aprendidos durante toda la materia, en algunos casos por separado, pero en otros, los mismos se entremezclaron, por lo que pudimos terminar de comprender y redondear los conceptos de la asignatura.

5. Implementación

Este analizador sintáctico y semántico del cálculo lambda λ^{bn} fue realizado en Python 2.7 utilizando la herramienta *PLY* para la generación del Lexer, Parser y posterior uso del Abstract Syntax Tree.

El proyecto dispone también de tests unitarios realizados mediante la utilización del módulo *unittest* de Python.

5.1. Estructura del proyecto

```
tp1
├── CLambda
├── main.py
├── Makefile
├── README.md
├── requirements.txt
├── lambdacalc
│   ├── expressions.py
│   ├── lexer.py
│   ├── parser.py
│   └── types.py
└── tests
    ├── basicExpressionsTest.py
    ├── complexExpressionsTest.py
    └── typesTest.py
```

5.1.1. CLambda

Binario para utilizar el programa. El mismo puede ser invocado de dos formas:

- `./CLambda <Expresión a evaluar entre comillas>` : Evaluará la expresión pasada por parámetro y finalizará la ejecución
- `./CLambda` : Modo interactivo, se irán ingresando expresiones por stdin. Ingresando la cadena *exit* se finaliza la ejecución.

5.1.2. main.py

Programa principal que es invocado desde *CLambda*.

5.1.3. Makefile

Makefile que permite tanto instalar los requirements como correr los tests.

El modo de uso es:

- *make*: Instala los requirements y corre los tests
- *make test*: Corre los tests
- *make install*: Instala los requirements

5.1.4. README.md

Descripción del proyecto.

5.1.5. requirements.txt

Dependencias que deben ser instaladas para la ejecución del programa.

5.1.6. `lambdacalc/expressions.py`

Módulo que contiene las clases que representan todas las expresiones del lenguaje.

5.1.7. `lambdacalc/types.py`

Módulo que contiene las clases que representan todos los tipos del cálculo lambda.

5.1.8. `lambdacalc/lexer.py`

Módulo que contiene las reglas del lexer para generar los tokens.

5.1.9. `lambdacalc/parser.py`

Módulo que contiene las reglas del parser para generar el Abstract Syntax Tree.

5.1.10. `tests/basicExpressionsTest.py`

Tests básicos sobre las expresiones. Simplemente se corrobora que se puedan generar correctamente los términos básicos del lenguaje.

5.1.11. `tests/complexExpressionsTest.py`

Tests que evalúan expresiones más complejas, además de poner a prueba las propiedades de asociatividad y precedencia.

5.1.12. `tests/typesTest.py`

Tests sobre el chequeo de tipos de las expresiones.

5.2. Ejecución del programa

Como se mencionó en la sección de la estructura del proyecto, para ejecutar el programa es necesario invocar el binario *CLambda*.

Su uso es el siguiente:

- `./CLambda <Expresión a evaluar entre comillas>` : Evaluará la expresión pasada por parámetro y finalizará la ejecución
- `./CLambda` : Modo interactivo, se irán ingresando expresiones por stdin. Ingresando la cadena *exit* se finaliza la ejecución.

5.3. Código

5.3.1. main.py

```
""" Archivo_principal_de_lambdacalc. """
from lambdacalc import parse
import sys
from string import join

def execute(expression):
    try:
        parsed = parse(expression)
        expressionType = str(parsed.type({}))
        print(str(parsed.value({})) + ":" + expressionType)
    except Exception as e:
        sys.stderr.write(str(e) + "\n")
        exit(1)

def main():
    if len(sys.argv) < 2:
        while True:
            expression = raw_input('expression>_')
            if expression == "exit":
                break
            else:
                execute(expression)
    else:
        execute(join(sys.argv[1:]))

if __name__ == '__main__':
    main()
```

5.3.2. lexer.py

```
#!/ coding: utf-8
import ply.lex as lex

"""_Lista_de_tokens_"""
tokens = [

    'BACKSLASH',
    'VAR',
    'COLON',
    'TYPE',
    'DOT',

    'ZERO',

    'R_PAREN',
    'L_PAREN',
    'ARROW'

]

# https://stackoverflow.com/questions/5022129/ply-lex-parsing-problem
reserved_tokens = {
    'if' : 'IF',
    'then' : 'THEN',
    'else' : 'ELSE',
    'succ' : 'SUCC',
    'pred' : 'PRED',
    'isZero' : 'IS_ZERO',
    'true' : 'TRUE',
    'false' : 'FALSE'
}

tokens += reserved_tokens.values()

t.BACKSLASH = r'\'\'

def t.VAR(t):
    r'[a-z][a-zA-Z0-9]*'
    if t.value in reserved_tokens:
        t.type = reserved_tokens[t.value]
    return t

t.COLON = r':'
t.TYPE = r'(Bool|Nat)'
t.DOT = r'\.'

t.ZERO = r'0'

t.L_PAREN = r'\('
t.R_PAREN = r'\)'
t.ARROW = r'→'

t.ignore = '_\t'

def t_error(t):
    raise Exception("Expresión_inválida_ '%s'" % t.value)

# Build the lexer
lexer = lex.lex()

def apply_lexer(string):
```

```
"""Aplica el lexer al string dado."""  
lexer.input(string)  
  
return list(lexer)
```

5.3.3. parser.py

```
#!/ coding: utf-8
""" Parser.lambdacalculo. """
import ply.yacc as yacc
from .lexer import tokens
from types import *
from expressions import *

def p_expression(subexpression):
    'expression::absExpression'
    subexpression[0] = subexpression[1]

def p_abstraction_expression(subexpression):
    'absExpression::BACKSLASH_varExpression_COLON_typeExpression_DOT_expression'
    subexpression[0] = AbstractionExpression(subexpression[2], subexpression[4], subexpression[6])

def p_abstraction_if_expression(subexpression):
    'absExpression::ifExpression'
    subexpression[0] = subexpression[1]

def p_if_expression(subexpression):
    'ifExpression::IF_expression_THEN_expression_ELSE_expression'
    subexpression[0] = IfExpression(subexpression[2], subexpression[4], subexpression[6])

def p_if_application_expression(subexpression):
    'ifExpression::appExpression'
    subexpression[0] = subexpression[1]

def p_application_expression(subexpression):
    'appExpression::appExpression_termExpression'
    subexpression[0] = ApplicationExpression(subexpression[1], subexpression[2])

def p_application_term_expression(subexpression):
    'appExpression::termExpression'
    subexpression[0] = subexpression[1]

def p_term_var_expression(subexpression):
    'termExpression::varExpression'
    subexpression[0] = subexpression[1]

def p_term_function_expression(subexpression):
    'termExpression::functionExpression'
    subexpression[0] = subexpression[1]

def p_var_expression(subexpression):
    'varExpression::VAR'
    subexpression[0] = VarExpression(subexpression[1])

def p_function_succ_expression(subexpression):
    'functionExpression::SUCC_L_PAREN_expression_R_PAREN'
    subexpression[0] = SuccExpression(subexpression[3])

def p_function_pred_expression(subexpression):
    'functionExpression::PRED_L_PAREN_expression_R_PAREN'
    subexpression[0] = PredExpression(subexpression[3])

def p_function_is_zero_expression(subexpression):
    'functionExpression::IS_ZERO_L_PAREN_expression_R_PAREN'
    subexpression[0] = IsZeroExpression(subexpression[3])

def p_function_val_expression(subexpression):
    'functionExpression::valExpression'
    subexpression[0] = subexpression[1]

def p_val_true_expression(subexpression):
```

```
'valExpression': _TRUE'
subexpression[0] = TRUE

def p_val_false_expression(subexpression):
    'valExpression': _FALSE'
    subexpression[0] = FALSE

def p_val_zero_expression(subexpression):
    'valExpression': _ZERO'
    subexpression[0] = ZERO

def p_val_expression_expression(subexpression):
    'valExpression': _L_PAREN_expression_R_PAREN'
    subexpression[0] = subexpression[2]

def p_type_function_expression(subexpression):
    'typeExpression': _TYPE_ARROW_typeExpression'
    subexpression[0] = FunctionType(Type(subexpression[1]), subexpression[3])

def p_type_basic_expression(subexpression):
    'typeExpression': _TYPE'
    subexpression[0] = Type(subexpression[1])

def p_error(p):
    message = "Hubo un error durante el parseo.\n"
    if p is not None:
        message += "Expresión '{0}' incorrecta en la posición {1}.".format(p.value, str(p.
            lexpos))
    else:
        message += "La expresión no puede ser parseada por la producción {0} de {1}:{2}.".
            format(parser.symstack, __file__.split("/")[1], parser.state)

    raise Exception(message)

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

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


5.3.4. types.py

```
class Type(object):
    def __init__(self, typeName):
        self.typeName = typeName

    def __repr__(self):
        return "Type(typeName:_{0})".format(self.typeName)

    def __str__(self):
        return self.typeName

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.typeName == other.typeName

    def __ne__(self, other):
        return not (self == other)

    def isFunction(self):
        return False;

BOOL_TYPE = Type("Bool")
NAT_TYPE = Type("Nat")

class FunctionType(Type):
    def __init__(self, domainType, rangeType):
        Type.__init__(self, "Function")
        self.domainType = domainType
        self.rangeType = rangeType

    def __repr__(self):
        return "FunctionType(domainType:_{0}_|_rangeType:_{1})".format(self.domainType
        , self.rangeType)

    def __str__(self):
        domainStr = "(" + str(self.domainType) + ")" if (self.domainType.isFunction())
        else self.domainType
        return "{0}->{1}".format(domainStr, self.rangeType)

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.domainType == other.
        domainType and self.rangeType == other.rangeType

    def isFunction(self):
        return True;

    def domain(self):
        return self.domainType

    def range(self):
        return self.rangeType
```

5.3.5. expressions.py

```
#!/ coding: utf-8
from types import *

class Expression(object):
    """
    La clase Expression tiene como objetivo representar una abstracción de las expresiones
    del calculo lambda.
    Cualquier expresión válida de este lenguaje deberá extender esta clase y sobrescribir
    los métodos de la misma.
    """

    def value(self, context):
        """
        El método value debe evaluar la expresión y retornar el valor que se obtiene al
        realizar dicha evaluación.

        Parameters:
        context: dict<VarExpression, Expression>
        Diccionario con el contexto de la expresión. La clave debe ser la
        expresión correspondiente a la variable
        y su valor debe ser la expresión por la cual debe ser sustituida.

        Returns:
        Expression
        Expresión obtenida al haber realizado la evaluación
        """
        raise NotImplementedError("Este método lo deben implementar las clases que
        hereden de Expression")

    def type(self, context):
        """
        El método type debe realizar el chequeo de tipos de la expresión y retornar el
        tipo que se obtiene al realizar dicho chequeo.

        Parameters:
        context: dict<VarExpression, Type>
        Diccionario con el contexto de la expresión. La clave debe ser la
        expresión correspondiente a la variable
        y su valor debe ser el tipo que debe tomar la misma.

        Returns:
        Type
        Tipo obtenido al haber realizado el chequeo de tipos
        """
        raise NotImplementedError("Este método lo deben implementar las clases que
        hereden de Expression")

class BoolExpression(Expression):
    def __init__(self, expression):
        self.expression = expression

    def __repr__(self):
        return "BoolExpression({0})".format(self.expression)

    def __str__(self):
        return self.expression
```

```
def type(self, context):
    return BOOL_TYPE

def value(self, context):
    return self

TRUE = BoolExpression("true")
FALSE = BoolExpression("false")

class ZeroExpression(Expression):
    def __repr__(self):
        return "ZeroExpression"

    def __str__(self):
        return "0"

    def type(self, context):
        return NAT_TYPE

    def value(self, context):
        return self

ZERO = ZeroExpression()

class ApplicationExpression(Expression):
    def __init__(self, firstExpression, secondExpression):
        self.firstExpression = firstExpression
        self.secondExpression = secondExpression

    def __repr__(self):
        return "ApplicationExpression(Expression1:_{0}_{1})".format(
            self.firstExpression, self.secondExpression)

    def __str__(self):
        return "{0}_{1}".format(self.firstExpression, self.secondExpression)

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.firstExpression == other.firstExpression and self.secondExpression == other.secondExpression

    def value(self, context):
        # Se evalúa la expresión de la derecha hasta obtener un valor
        parameterValue = self.secondExpression.value(context)
        # Se evalúa la expresión de la izquierda hasta obtener un valor (es necesario
        # porque antes de evaluar no necesariamente es una abstracción),
        # el cual se asume que una vez evaluado es una abstracción.
        # Luego se le indica a la abstracción que se aplique, es decir, que realice el
        # reemplazo de variables por sus valores y
        # retorne el valor final
        return self.firstExpression.value(context).apply(parameterValue, context)

    def type(self, context):
        abstractionType = self.firstExpression.type(context)
        if not abstractionType.isFunction():
            raise Exception("La expresión de la izquierda no es una abstracción")
        else:
            parameterType = self.secondExpression.type(context)
            if not abstractionType.domain() == parameterType:
                raise Exception("El parámetro pasado no se corresponde con el
                    dominio de la abstracción")
            else:
                return abstractionType.range()

class AbstractionExpression(Expression):
    def __init__(self, variable, variableType, expression):
```

```
        self.variable = variable
        self.variableType = variableType
        self.expression = expression

    def __repr__(self):
        return "AbstractionExpression(VarName:_{0}_|_VarType:_{1}_|_Expression:_{2})".format(
            self.variable, self.variableType, self.expression)

    def __str__(self):
        return "{0}:{1}.{2}".format(self.variable, str(self.variableType), self.expression)

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.variable == other.variable
            and self.variableType == other.variableType \
            and self.expression == other.expression

    def value(self, context):
        return self

    def apply(self, parameterValue, context):
        """
        El método apply debe introducir el valor de la variable en el contexto y
        retornar el valor obtenido
        al evaluar su expresión.

        Parameters:
        parameterValue: Expression
        Expresión que será el valor de la variable

        context: dict<VarExpression, Expression>
        Contexto en donde se deberá definir la expresión para la variable de
        la abstracción

        Returns:
        Expression
        Valor obtenido luego de realizar la evaluación de la expresión
        reemplazando variables por su valor.
        """
        context[self.variable] = parameterValue
        return self.expression.value(context)

    def type(self, context):
        context[self.variable] = self.variableType
        return FunctionType(self.variableType, self.expression.type(context))

class IfExpression(Expression):
    def __init__(self, condition, ifTrue, ifFalse):
        self.condition = condition
        self.ifTrue = ifTrue
        self.ifFalse = ifFalse

    def __repr__(self):
        return "IfExpression(condition:_{0}_|_ifTrue:_{1}_|_ifFalse:_{2})".format(
            self.condition, self.ifTrue, self.ifFalse)

    def __str__(self):
        return "if_{0}_then_{1}_else_{2}".format(self.condition, self.ifTrue, self.ifFalse)

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.condition == other.condition
            and self.ifTrue == other.ifTrue \
            and self.ifFalse == other.ifFalse
```

```
def value(self, context):
    return self.ifTrue.value(context) if (self.condition.value(context) == TRUE)
    else self.ifFalse.value(context)

def type(self, context):
    if (self.condition.type(context) != BOOL.TYPE):
        raise Exception("La condición debe ser de tipo Bool")
    elif (self.ifTrue.type(context) != self.ifFalse.type(context)):
        raise Exception("Las expresiones resultantes del if deben tener el mismo tipo")
    return self.ifTrue.type(context)

class VarExpression(Expression):
    def __init__(self, varName):
        self.varName = varName

    def __repr__(self):
        return "VarExpression({})".format(self.varName)

    def __str__(self):
        return self.varName

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.varName == other.varName

    def __hash__(self):
        return hash(self.varName)

    def type(self, context):
        if (context.has_key(self)):
            return context[self]
        else:
            raise Exception("La variable {} está libre".format(self.varName))

    def value(self, context):
        if (context.has_key(self)):
            return context[self].value(context)
        else:
            raise Exception("La variable {} está libre".format(self.varName))

class NatExpression(Expression):
    def __init__(self, expression):
        self.expression = expression

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.expression == other.expression

    def type(self, context):
        if (self.expression.type(context) == NAT.TYPE):
            return NAT.TYPE
        else:
            raise Exception("La subexpresión no es de tipo Nat")

class SuccExpression(NatExpression):
    def __repr__(self):
        return "SuccExpression(Expression: {})".format(self.expression)

    def __str__(self):
        return "succ({})".format(self.expression)

    def value(self, context):
        return SuccExpression(self.expression.value(context))
```

```
class PredExpression(NatExpression):
    def __repr__(self):
        return "PredExpression(Expression:_{0})".format(self.expression)

    def __str__(self):
        return "pred({0})".format(self.expression)

    def value(self, context):
        if (self.expression == ZERO):
            return ZERO
        else:
            expressionValue = self.expression.value(context)
            return ZERO if (expressionValue == ZERO) else expressionValue.value(context)

class IsZeroExpression(Expression):
    def __init__(self, expression):
        self.expression = expression

    def __repr__(self):
        return "IsZeroExpression({0})".format(self.expression)

    def __str__(self):
        return "isZero({0})".format(self.expression)

    def type(self, context):
        if (self.expression.type(context) == NAT.TYPE):
            return BOOL.TYPE
        else:
            raise Exception("La subexpresión no es de tipo Nat")

    def value(self, context):
        return TRUE if (self.expression.value(context) == ZERO) else FALSE
```

5.4. Tests

5.4.1. basicExpressionsTest.py

```
import unittest
from context import import *

class BasicTests(unittest.TestCase):

    def testTrue(self):
        self.assertEqual(parse("true").value({}), TRUE)

    def testFalse(self):
        self.assertEqual(parse("false").value({}), FALSE)

    def testZero(self):
        self.assertEqual(parse("0").value({}), ZERO)

    def testSuccZero(self):
        self.assertEqual(parse("succ(0)").value({}), SuccExpression(ZERO))

    def testPredZero(self):
        self.assertEqual(parse("pred(0)").value({}), ZERO)

    def testPredSuccZero(self):
        self.assertEqual(parse("pred(succ(0))").value({}), ZERO)

    def testPredSuccSuccZero(self):
        self.assertEqual(parse("pred(succ(succ(0)))").value({}), SuccExpression(ZERO))

    def testSuccPredSuccZero(self):
        self.assertEqual(parse("succ(pred(succ(0)))").value({}), SuccExpression(ZERO))

    def testIsZeroZero(self):
        self.assertEqual(parse("isZero(0)").value({}), TRUE)

    def testIsZeroPredZero(self):
        self.assertEqual(parse("isZero(pred(0))").value({}), TRUE)

    def testIsZeroSuccZero(self):
        self.assertEqual(parse("isZero(succ(0))").value({}), FALSE)

    def testIfTrue(self):
        self.assertEqual(parse("if true then 0 else succ(0)").value({}), ZERO)

    def testIfFalse(self):
        self.assertEqual(parse("if false then 0 else succ(0)").value({}),
                          SuccExpression(ZERO))

    def testLambda(self):
        self.assertEqual(parse("\\x:Nat.x").value({}), AbstractionExpression(
            VarExpression("x"), NAT_TYPE, VarExpression("x")))

    def testApplicationSimple(self):
        self.assertEqual(parse("(\\x:Nat.x) 0").value({}), ZERO)

    def testApplicationFunction(self):
        self.assertEqual(parse("(\\x:Nat.succ(x)) 0").value({}), SuccExpression(ZERO))

    def testApplicationLambda(self):
        self.assertEqual(parse("(\\f:Nat->Bool.f 0) (\\x:Nat.isZero(x))").value({}),
                          TRUE)

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

5.4.2. complexExpressionsTest.py

```

#!/ coding: utf-8
import unittest
from context import import *

class BasicTests(unittest.TestCase):

    def testSuccExpressionValue(self):
        self.assertEqual(parse("succ(if true then succ(0) else 0)").value({}),
            SuccExpression(SuccExpression(ZERO)))

    def testPredExpressionValue(self):
        self.assertEqual(parse("pred((\x:Nat.succ(x))_succ(0))").value({}),
            SuccExpression(ZERO))

    def testIsZeroExpressionValue(self):
        self.assertEqual(parse("isZero(if false then 0 else succ(0))").value({}),
            FALSE)

    def testIfConditionExpressionValue(self):
        self.assertEqual(parse("if isZero(pred(if true then 0 else succ(0))) then 0
            else succ(0)").value({}), ZERO)

    def testIfTrueExpressionValue(self):
        self.assertEqual(parse("if isZero(0) then (\x:Nat.succ(x))_0 else succ(succ
            (0))").value({}), SuccExpression(ZERO))

    def testIfFalseExpressionValue(self):
        self.assertEqual(parse("if isZero((\b:Bool.b)) then (\x:Nat.succ(x))_succ(0)
            else pred((\x:Nat.succ(x))_succ(0))").value({}), SuccExpression(ZERO))

    def testLambdaRigthAssociation(self):
        self.assertEqual(parse("\f:Nat->Nat.f_0").value({}), AbstractionExpression(
            VarExpression("f"), FunctionType(NAT.TYPE, NAT.TYPE), ApplicationExpression(
            VarExpression("f"), ZERO)))

    def testLambdaPrecedence(self):
        self.assertEqual(parse("\f:Nat->Nat.f_(if true then 0 else succ(0))").value
            ({}), \
            AbstractionExpression(VarExpression("f"), FunctionType(NAT.TYPE,
            NAT.TYPE), \
            ApplicationExpression(VarExpression("f"), \
            IfExpression(TRUE, ZERO, SuccExpression(ZERO)))))

    def testIfRigthAssociation(self):
        self.assertEqual(parse("if true then 0 else (\x:Nat.succ(x))_0").value({}),
            ZERO)

    def testIfPrecedence(self):
        self.assertEqual(parse("if false then 0 else (\x:Nat.succ(x))_0").value({}),
            SuccExpression(ZERO))

    def testApplicationLeftExpressionLambdaWithoutBrackets(self):
        with self.assertRaisesRegexp(Exception, r".* Expresión '\_incorrecta en la
            posición.*"):
            self.assertEqual(parse("( \f:Nat->Nat.f_0 ) \y:Nat.y").value({}), ZERO
            )

    def testApplicationLeftExpressionLambdaWithBrackets(self):
        self.assertEqual(parse("( \f:Nat->Nat.f_0 ) \y:Nat.y").value({}), ZERO)

    def testApplicationLeftExpressionIfWithoutBrackets(self):
        with self.assertRaisesRegexp(Exception, "Expresión '\_if '\_incorrecta en la
            posición"):
            parse("( \f:Nat->Nat.f_0 ) if true then \y:Nat.y else \x:Nat.succ(x)"
            ).value({})

```



```
def testApplicationLeftExpressionIfWithBrackets(self):
    self.assertEqual(parse("(\\f:Nat->Nat.f_0)_(if_true_then_\\y:Nat.y_else_\\x:
        Nat.succ(x))").value({}), ZERO)

def testComplexExpressionWithMultiplePrecedencesAndAssociations(self):
    self.assertEqual(parse("(\\x:Nat->Nat.\\y:Nat.(\\z:Bool.if_z_then_x_y_else_0))
        _((\\j:Nat.succ(j))_succ(succ(0))_true").value({}), \
        SuccExpression(SuccExpression(SuccExpression(ZERO))))

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

5.4.3. typesTest.py

```
#!/ coding: utf-8
import unittest
from context import import *

class BasicTests(unittest.TestCase):

    def testTrueBool(self):
        self.assertEqual(parse("true").type({}), BOOL_TYPE)

    def testFalseBool(self):
        self.assertEqual(parse("false").type({}), BOOL_TYPE)

    def testZeroNat(self):
        self.assertEqual(parse("0").type({}), NAT_TYPE)

    def testSuccZeroNat(self):
        self.assertEqual(parse("succ(0)").type({}), NAT_TYPE)

    def testPredZeroNat(self):
        self.assertEqual(parse("pred(0)").type({}), NAT_TYPE)

    def testPredSuccZeroNat(self):
        self.assertEqual(parse("pred(succ(0))").type({}), NAT_TYPE)

    def testPredSuccSuccZeroNat(self):
        self.assertEqual(parse("pred(succ(succ(0)))").type({}), NAT_TYPE)

    def testSuccPredSuccZeroNat(self):
        self.assertEqual(parse("succ(pred(succ(0)))").type({}), NAT_TYPE)

    def testIsZeroZeroBool(self):
        self.assertEqual(parse("isZero(0)").type({}), BOOL_TYPE)

    def testIsZeroPredZeroBool(self):
        self.assertEqual(parse("isZero(pred(0))").type({}), BOOL_TYPE)

    def testIsZeroSuccZeroBool(self):
        self.assertEqual(parse("isZero(succ(0))").type({}), BOOL_TYPE)

    def testSuccSubExpressionNotNat(self):
        with self.assertRaisesRegexp(Exception, 'La subexpresión no es de tipo Nat'):
            parse("succ(true)").type({})

    def testPredSubExpressionNotNat(self):
        with self.assertRaisesRegexp(Exception, 'La subexpresión no es de tipo Nat'):
            parse("pred(true)").type({})

    def testIsZeroSubExpressionNotNat(self):
        with self.assertRaisesRegexp(Exception, 'La subexpresión no es de tipo Nat'):
            parse("isZero(false)").type({})

    def testIfConditionNotBool(self):
        with self.assertRaisesRegexp(Exception, 'La condición debe ser de tipo Bool'):
            parse("if 0 then 0 else succ(0)").type({})

    def testIfDifferentExpressionsType(self):
        with self.assertRaisesRegexp(Exception, 'Las expresiones resultantes del if deben tener el mismo tipo'):
            parse("if true then false else succ(0)").type({})

    def testIfTrueType(self):
        self.assertEqual(parse("if true then 0 else succ(0)").type({}), NAT_TYPE)

    def testIfFalseType(self):
        self.assertEqual(parse("if false then true else false").type({}), BOOL_TYPE)
```

```
def testLambdaType1(self):
    self.assertEqual(parse("\\x:Nat.succ(x)").type({}), FunctionType(NAT.TYPE,
        NAT.TYPE))

def testLambdaType2(self):
    self.assertEqual(parse("\\x:Bool.x").type({}), FunctionType(BOOL.TYPE,
        BOOL.TYPE))

def testLambdaType3(self):
    self.assertEqual(parse("\\x:Nat.isZero(x)").type({}), FunctionType(NAT.TYPE,
        BOOL.TYPE))

def testLambdaType4(self):
    self.assertEqual(parse("\\x:Nat->Bool.x").type({}), FunctionType(FunctionType(
        NAT.TYPE, BOOL.TYPE), FunctionType(NAT.TYPE, BOOL.TYPE)))

def testApplicationSimpleType(self):
    self.assertEqual(parse("(\\x:Nat.x)_0").type({}), NAT.TYPE)

def testApplicationFunctionType(self):
    self.assertEqual(parse("(\\x:Nat.succ(x))_0").type({}), NAT.TYPE)

def testApplicationLambdaType(self):
    self.assertEqual(parse("(\\f:Nat->Bool.f_0)_ (\\x:Nat.isZero(x))").type({}),
        BOOL.TYPE)

def testApplicationLambdaParameterTypeDifferentFromVariableType(self):
    with self.assertRaisesRegexp(Exception, 'El parámetro pasado no se corresponde
        con el dominio de la abstracción'):
        self.assertEqual(parse("(\\x:Nat.x)_true").type({}), NAT.TYPE)

def testApplicationWithoutLambdaType(self):
    with self.assertRaisesRegexp(Exception, 'La expresión de la izquierda no es
        una abstracción'):
        self.assertEqual(parse("(true)_true").type({}), NAT.TYPE)

def testVariableFree(self):
    with self.assertRaisesRegexp(Exception, 'La variable x está libre'):
        self.assertEqual(parse("x").type({}), NAT.TYPE)

def testVariableNotFree(self):
    self.assertEqual(parse("x").type({VarExpression("x"): NAT.TYPE}),
        NAT.TYPE)

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

Referencias

- [1] Sitio oficial de Paradigmas de Programación <http://www.dc.uba.ar/materias/plp/cursos/2017/cuat1/index.html>
- [2] Documentación de PLY <http://www.dabeaz.com/ply>