DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Teoría de Lenguajes
Analizador Sintáctico y Semántico para λ^{bn}

4 de julio de 2017

Grupo: *Ullman, Sethi y los demás*

Integrante	LU	Correo electrónico
Gabriel Eric Thibeault	114/13	gabriel.eric.thibeault@gmail.com
Gonzalo Ciruelos Rodríguez	063/14	gonzalo.ciruelos@gmail.com
Luis Agustín Nieto	46/01	lnieto@dc.uba.ar

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

1. Introducción

En el presente trabajo práctico se realiza un analizador sintáctico y semántico para un subconjunto del cálculo lambda tipado sobre booleanos y naturales λ^{bn}, para dicha implementación se utilizó PLY que es una implementación en Python de las clásicas herramientas *lex* y *yacc*.

El lexer está implementado en *lexer.py*, en el mismo definimos las expresiones regulares y tokens que sirven para definir si cadenas de entrada son o no válidas acorde a la gramática creada. El parser,donde definimos las producciones de nuestra gramática, está implementado en *parser.py*.
Se realizaron una serie de test para probar la correcta implementacion del analizador, los mismos se encuentran en *test.py*.

2. Gramática

La mayor dificultad del trabajo fue armar una gramática que sirviese para para el lenguaje pedido, recordemos que el mismo es:

```
M ::=  x | true | false | if M1 then M2 else M3 | \x:T.M | M1 M2 | 0 | succ(M)
            | pred(M) | iszero(M)

T ::= Bool | Nat | T1 -> T2
```

Luego de varias pruebas se llegó a esta gramática:

$G = \langle \{E, S, C, L, T'\}, \{var, true, false, iszero, succ, pred, (,), 0, Bool, Nat, if, then, else, \backslash, :, .\}, P, E \rangle$, con P:

$$
\begin{array}{rcl}
E & \rightarrow & S\ L \\
 & | & S \\
S & \rightarrow & S\ C \\
 & | & \lambda \\
C & \rightarrow & (E) \\
 & | & var \\
 & | & true \\
 & | & false \\
 & | & 0 \\
 & | & iszero(E) \\
 & | & succ(E) \\
 & | & pred(E) \\
 & | & if\ E\ else\ E\ then\ C \\
L & \rightarrow & \backslash\ \mathrm{V} : \mathrm{T}\ .\ \mathrm{E} \\
T & \rightarrow & T'\ arrow\ T \\
 & | & T' \\
T' & \rightarrow & (T\ arrow\ T') \\
 & | & Bool \\
 & | & Nat
\end{array}
\tag{1}
$$

Expliquemosla brevemente:

- E. La idea es que este no terminal represente a todas las expresiones, que o bien pueden ser un S, o bien un S aplicado a un L.
- S. Estos terminos seran una lista de aplicaciones a cosas distintas que un L (puede haber un L, pero deberá estar entre paréntesis, siguiendo la reducción $SC \rightarrow S(E) \rightarrow S(SL) \rightarrow S(L)$). Este no-terminal nos permite forzar que la aplicación asocie a izquierda (notemos que la producción es recursiva a izquierda).

- C. Estos serán todos los términos sin paréntesis, excepto los lambda. Necesitamos que los lambda vayan entre paréntesis en general, porque por ejemplo el string $nx\ :\ Nat\ .\ x\ y$ no queda claro si es una función aplicada a y, o si $x\ y$ es el cuerpo de la abtracción.
- L. Básicamente las abstracciones lambda.
- T. Como los tipos flecha asocian a derecha ($Nat \to Nat \to Nat$ es $Nat \to (Nat \to Nat)$), necesitamos tener dos no terminales para que la grámatica no nos quede ambigua.
- T'. Los tipos de la derecha de una flecha tienen que, o bien ser un tipo flecha entre paréntesis, o bien ser tipos básicos.

Veamos además las expresiones regulares de cada token:

- var. "[abcjxyz]", o sea cualquier letra del conjunto a, b, c, j, x, y, z, para tener una cantidad suficiente de variables, pero necesitamos prohibir ciertos nombres (por ejemplo "i", "e"), para que no haya conflictos en el parser, porque colisionarian con las sentencias como "else" o "if".
- $arrow$. " $->$ ".
- Todas el resto la expresión regular es igual al nombre del token.

3. Código

3.1. Explicación

En el código basicamente tenemos una clase para cada tipo de término. Todas estas clases son polimorficas y saben responder los mismos mensajes (evaluate, dame tu tipo, etc). La idea es que el parseo nos devuelva un árbol de todas estas clases, y simplemente le pidamos el valor y el tipo a la raiz, que se ocupará de hacer todas las llamadas necesarias a sus hijos, y así sucesivamente.

O sea, lo que sucede es que al terminar de parsear tenemos el AST de la cadena, y luego la evaluamos siguiendo las reglas del cálculo lambda.

Un paso intermedio no menor (luego de obtener el AST pero antes de evaluar) es que le asignamos "contextos" a todos los subtérminos de la cadena. Por ejemplo, en la cadena "(\x : Nat . x)", nos gustaría que la clase que representara a la x del cuerpo de la abstracción supiera que esa x tiene tipo Nat.

Esto lo logramos con una función llamada "add_judgement", que simplemente lo que hace es tomar el contexto de un término, y pasárselo a todos sus hijos. Además, si el término es un lambda, le va a pasar a los hijos su contexto sumandole un juicio que dice que el parámetro tiene el tipo dado.

Luego, antes de evaluar el AST, necesitamos propagar todos estos valores hacia abajo. Logramos esto haciendo una llamada al "add_judgement" de la raíz con valores dummy.

3.1.1. Conflicto

En las reglas de los tipos ($T \to T' -> T$), PLY nos decía que había un conflicto. Esto se soluciona de forma tan fácil como indicarle que la flecha asocia a derecha. Sin embargo, en algunas versiones de PLY no parecía tomar este comando e igualmente salía el Warning, pero lo resolvía de la forma correcta.

3.2. Uso y Tests

Para poder ejecutar el código necesitamos instalar varias dependencias, como utilizamos de base el código del taller de **PLY** usamos el mismo archivo `requirements.txt`, la única diferencia con el taller es que el trabajo está implementado en Python3 por lo que tenemos que usar *pip3* en lugar de *pip*.

```
pip3 install -r requirements.txt
```

El comando para evaluar expresiones es **CLambda** y su sintaxis es `./CLambda [EXPRESION]`.
Si la expresión es correcta debe devolver su evaluación, caso contrario devuelve un mensaje de error detallando que fue lo que pasó, por ejemplo:

```
tlen-tp1$ ./CLambda.py '(\x:Nat. if iszero(x) then succ(x) else pred(x)) 0'
1  :  Nat
```

O si queremos ver un error de tipado:

```
tlen-tp1$ ./CLambda.py '(\x:Nat. if iszero(x) then succ(x) else true) 0'
```

```
Ilegal: distintos tipos en el cuerpo del if: 1 y true tienen distintos tipos (Nat y
Bool respectivamente).
```

Para testar el correcto funcionamiento de la implementación se armaron varios casos de test dentro del archivo *test.py*, en el mismo se define la expresion el resultado y el tipo esperado de la evaluación. Se hicieron tanto casos satisfactorios como casos de evaluaciones fallidas.
Para probarlo solo tenemos que ejecutarlo:

```
tlen-tp1$ ./tests.py
PASSED 0
PASSED true
PASSED if true then 0 else false
PASSED \x:Bool.if x then false else true
PASSED \x:Nat.succ(0)
PASSED \z:Nat.z
PASSED (\x:Bool.succ(x)) true
PASSED succ(succ(succ(0)))
PASSED x
PASSED succ(succ(pred(0)))
PASSED 0 0
PASSED \x:Nat->Nat.\y:Nat.(\z:Bool.if z then x y else 0)
PASSED (\x:Nat->Nat.\y:Nat.(\z:Bool.if z then x y else 0)) (\j:Nat.succ(j))
       succ(succ(succ(succ(succ(succ(succ(succ(0)))))))) true
PASSED (\z:Nat. pred(z)) ((\x:Nat->Nat. x succ(succ(0))) \y:Nat. y)
PASSED (\x:Nat.succ(x)) 0 0
PASSED (\x:Nat->Nat. x succ(succ(0))) \y:Nat. y
PASSED (\x:Nat->Nat. x pred(succ(0))) \y:Nat. y
PASSED (\x:Nat. if iszero(x) then succ(x) else pred(x)) pred(0)
PASSED (\x:Nat. if iszero(x) then succ(x) else pred(x)) succ(0)
PASSED (\x:Nat. if iszero(x) then succ(x) then pred(x)) succ(0)
PASSED (\x:Nat. if iszero(x) then succ(x) then true) succ(0)
```

3.3. lexer.py

```
#! coding: utf-8
"""Calculator lexer example."""
import ply.lex as lex
from .ast import TNat,TBool,TArrow, LBool, LZero, LVar, LLambda, LSucc, LPred,
    LApp, LIfThenElse,LIsZero
from difflib import get_close_matches

"""
Lista de tokens

El analizador léxico de PLY (al llamar al método lex.lex()) va a buscar
para cada uno de estos tokens una variable "t_TOKEN" en el módulo actual.

Sí, es súper nigromántico pero es lo que hay.

t_TOKEN puede ser:

- Una expresión regular
- Una función cuyo docstring sea una expresión regular (bizarro).

En el segundo caso, podemos hacer algunas cosas "extras", como se
muestra aquí abajo.

"""

tokens = (
    'TRUE',
    'FALSE',
    'ZERO',
    'IF',
    'THEN',
    'ELSE',

    'VAR',
    'LAM',
    'COLON',
    'DOT',

    'LPARENS',
    'RPARENS',

    'ISZERO',
    'SUCC',
    'PRED',

    'ARROW',
    'NAT',
    'BOOL',

)


t_IF = r'if'
t_THEN = r'then'
t_ELSE = r'else'

t_LAM = r'\\'
t_COLON = r':'
t_DOT = r'\.'

```

```
60  t_LPARENS = r'\('
61  t_RPARENS = r'\)'
62
63  t_ISZERO = r'iszero'
64  t_SUCC = r'succ'
65  t_PRED = r'pred'
66
67  t_ARROW = r'->'
68  t_NAT = r'Nat'
69  t_BOOL = r'Bool'
70
71  t_ignore = ' \t'
72
73  tokenizables = [
74  'if',
75  'then',
76  'else',
77  '\\',
78  ':',
79  '.',
80  '(',
81  ')',
82  'iszero',
83  'succ',
84  'pred',
85  '->',
86  'Nat',
87  'Bool']
88
89  def t_TRUE(t):
90      r'true'
91      t.value = LBool(True)
92      return t
93
94  def t_FALSE(t):
95      r'false'
96      t.value = LBool(False)
97      return t
98
99  def t_ZERO(t):
100     r'0'
101     t.value = LZero()
102     return t
103
104 # Sin i,s,p
105 def t_VAR(t):
106     r'[abcjxyz]'
107     t.value = LVar(t.value)
108     return t
109
110 def t_error(t):
111     errStr = t.value.split(' ', 1)[0]
112     print('Error de sintaxis: la cadena "{}" no puede ser tokenizada.'
113           .format(errStr))
114     if get_close_matches(errStr, tokenizables, n = 1):
115         print('Habrás querido usar "{}"?'
116               .format(*get_close_matches(errStr, tokenizables, n = 1)))
117     exit(0)
118
119 # Build the lexer
120 lexer = lex.lex()
121 def apply_lexer(string):
```

```
122         """Aplica el lexer al string dado."""
123         lexer.input(string)
124 
125         return list(lexer)
```

3.4. parser.py

```
# coding=utf-8
"""Parser LR(1) de lambda."""
import ply.yacc as yacc
from .lexer import tokens
from .ast import TNat, TBool, TArrow, LBool, LZero, LVar, LLambda, LSucc, LPred,
    LApp, LIfThenElse, LIsZero

precedence = (
    ('right', 'ARROW'),
)

########################## EXPRESION ##############################################
#
#  E -> S L
#     | S
#
###################################################################################

def p_expr_s_lambda(p):
    'expr : S lambda'
    if p[1] is None:
        p[0] = p[2]
    else:
        p[0] = LApp(p[1], p[2])


def p_expr_s(p):
    'expr : S'
    p[0] = p[1]

########################## S ######################################################
#
#  S -> S cont
#     |
#
###################################################################################

def p_s_factor(p):
    'S : S cont'
    if p[1] is None:
        p[0] = p[2]
    else:
        p[0] = LApp(p[1], p[2])

def p_s_empty(p):
    'S : '

########################## cont ###################################################
#
#  C -> (E)
#     | var
#     | true
#     | false
#     | 0
#     | iszero(E)
#     | succ(E)
#     | pred(E)
#     | if E then E else C
#
###################################################################################
```

```
60
61  def p_cont_expr(p):
62      'cont : LPARENS expr RPARENS'
63      p[0] = p[2]
64
65  def p_cont_var(p):
66      'cont : VAR'
67      p[0] = p[1]
68
69  def p_cont_true(p):
70      'cont : TRUE'
71      p[0] = p[1]
72
73  def p_cont_false(p):
74      'cont : FALSE'
75      p[0] = p[1]
76
77  def p_cont_zero(p):
78      'cont : ZERO'
79      p[0] = p[1]
80
81  def p_cont_iszero(p):
82      'cont : ISZERO LPARENS expr RPARENS'
83      p[0] = LIsZero(p[3])
84
85  def p_cont_succ(p):
86      'cont : SUCC LPARENS expr RPARENS'
87      p[0] = LSucc(p[3])
88
89  def p_cont_pred(p):
90      'cont : PRED LPARENS expr RPARENS'
91      p[0] = LPred(p[3])
92
93  def p_cont_ifthenelse(p):
94      'cont : IF expr THEN expr ELSE cont'
95      p[0] = LIfThenElse(p[2], p[4], p[6])
96
97
98  ########################## LAMBDA ##############################################
99  #
100 #  L -> \ V : T . E
101 #
102 ################################################################################
103
104 def p_lambda(p):
105     'lambda : LAM VAR COLON type DOT expr'
106     p[0] = LLambda(p[2], p[4], p[6])
107
108
109 ########################## TIPOS ###############################################
110 #
111 #  T -> T' -> T
112 #      | T'
113 #
114 #  T' -> (T -> T')
115 #       | Bool
116 #       | Nat
117 #
118 ################################################################################
119
120
121 def p_type_arrow(p):
```

```
122     'type : typex ARROW type'
123     p[0] = TArrow(p[1], p[3])
124
125 def p_type_typex(p):
126     'type : typex'
127     p[0] = p[1]
128
129 def p_typex_arrow(p):
130     'typex : LPARENS type ARROW typex RPARENS'
131     p[0] = TArrow(p[2], p[4])
132
133 def p_typex_nat(p):
134     'typex : NAT'
135     p[0] = TNat()
136
137 def p_typex_bool(p):
138     'typex : BOOL'
139     p[0] = TBool()
140
141
142 def p_error(p):
143     if p is not None:
144         exit('Error de sintaxis: "{}" no esperado en la posicion {}.'.format(p.
                value, p.lexpos))
145     else:
146         exit('Error de sintaxis: fin de linea no esperado ("$").')
147
148
149
150 # Build the parser
151 parser = yacc.yacc(debug=True)
152
153 def apply_parser(str):
154     p = parser.parse(str)
155     p.add_judgement('h4x0r', 'turururu')  # Llamo a add judgement del padre asi
156                                           # propaga todos los judgements hacia
157                                           # abajo.
158     # print(repr(p))
159     while p is not None and not p.is_value():
160         p = p.value()
161         # print(repr(p))
162     return p
```

3.5. tests.py

```
#!/usr/bin/python3
from lambda_calculus import parse, lex

tests = [
    ('0', '0', 'Nat'),
    ('true', 'true', 'Bool'),
    ('if true then 0 else false', None, None),
    ('\\x:Bool.if x then false else true', '\ x : Bool . if x then false else true', 'Bool -> Bool'),
    ('\\x:Nat.succ(0)', '\ x : Nat . 1', 'Nat -> Nat'),
    ('\\z:Nat.z', '\ z : Nat . z', 'Nat -> Nat'),
    ('(\\x:Bool.succ(x)) true', None, None),
    ('succ(succ(succ(0)))', '3', 'Nat'),
    ('x', None, None),
    ('succ(succ(pred(0)))', '2', 'Nat'),
    ('0 0', None, None),
    ('\\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.\\y:Nat.(\\z:Bool.if z then x y else 0)) (\\j:Nat.succ(j)) succ(succ(succ(succ(succ(succ(succ(succ(0))))))))) true', '9', 'Nat'),
    # Nuestros tests.
    ('(\\z:Nat. pred(z)) ((\\x:Nat->Nat. x succ(succ(0))) \\y:Nat. y)', '1', 'Nat'),
    ('(\\x:Nat.succ(x)) 0 0', None, None),
    ('(\\x:Nat->Nat. x succ(succ(0))) \y:Nat. y', '2', 'Nat'),
    ('(\\x:Nat->Nat. x pred(succ(0))) \y:Nat. y', '0', 'Nat'),
    ('(\\x:Nat. if iszero(x) then succ(x) else pred(x)) pred(0)', '1', 'Nat'),
    ('(\\x:Nat. if iszero(x) then succ(x) else pred(x)) succ(0)', '0', 'Nat'),
    ('(\\x:Nat. if iszero(x) then succ(x) then pred(x)) succ(0)', None, None),
    ('(\\x:Nat. if iszero(x) then succ(x) then true) succ(0)', None, None),
]



for test in tests:
    try:
        p = parse(test[0])
        if str(p.value()) == test[1] and str(p.type()) == test[2]:
            print('PASSED', test[0])
        else:
            print('FAILED', test[0])
            if str(p.value()) != test[1]:
                print('Got', str(p.value()), 'but expected', test[1])
            if str(p.type()) != test[2]:
                print('Got', str(p.type()), 'but expected', test[2])
    except:
        if test[1] is None:
            print('PASSED', test[0])
        else:
            print('FAILED', test[0])
```

4. Conclusiones

- *PLY* demostró ser una herramienta muy útil que nos permitió llevar, casi sin inconvenientes, la implementación del papel al código.
- La mayor dificultad fue armar la gramática de forma que no quede ambigua, especialmente con la aplicación de función. Además, fue dificultoso "pelearnos" con *PLY* para que nos tome la asociatividad de la flecha. Se tuvo que crear la noción de *contexto* para poder usarla al evaluar las expresiones.
- También podemos decir que los temas vistos durante la cursada como gramática de atributos, gramática LALR, parsers, etc. fueron útiles para poder entender y resolver los problemas presentados y que el trabajo práctico fue un buena aplicación de conceptos que de otra forma hubiesen quedado solo en el plano teórico.

Teoría de Lenguajes - Trabajo Práctico
Analizador Sintáctico y Semántico para λ^{bn}

Versión 1.1

1er cuatrimestre 2017

Fecha de entrega: miércoles 5 de julio

1. Introducción

Se desea crear un analizador sintáctico y semántico para un subconjunto del cálculo lambda tipado, sobre booleanos y naturales (λ^{bn}). Así, se deberá analizar si una expresión sigue la sintaxis esperada, y, en caso de que sea válida, evaluarla adecuadamente e indicar de qué tipo es la misma. Para la evaluación se puede tomar como referencia la semántica operacional descripta en la 1era clase teórica y práctica de la materia Paradigmas de Lenguajes de Programación[1], aunque no es necesario seguir el proceso de evaluación paso a paso.

2. Descripción del lenguaje de entrada

A efectos de este trabajo, el cálculo lambda tipado sobre booleanos y naturales deberá soportar 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)
```

Siendo,

- `true` y `false`: las constantes de verdad,
- `if M1 then M2 else M3`: el condicional, por lo que si `M1` se evalua a `true`, resulta en `M1` y, si no, en `M2`, y tanto `M1` como `M2` deben ser del mismo tipo,
- `M1 M2`: la aplicación de la función denotada por el término `M1` al argumento `M2`, por ejemplo, `(\x:Nat.succ(succ(x))) 3`, se evalúa a 5, siendo este operador asociativo a izquierda (i.e., `M N P` se evalúa como `(M N) P`),
- `\x:T.M`: una función anónima cuyo parámetro formal es `x`, de tipo `T`, y tiene a `M` como cuerpo, siendo la operación de menor precedencia,
- `x`: una variable de términos, que puede estar asociada a cualquier término válido,
- `succ(M)`: el término para el cual se evaluará M hasta obtener un número, y se lo incrementará,
- `pred(M)`: el término para el cual se evaluará M hasta obtener un número, y se lo decrementará,
- `iszero(M)`: el término para el cual se evaluará M hasta obtener un número, y evaluará a `true` o `false` según sea cero o no.

A su vez, cada expresión bien formada del lenguaje puede tomar alguno de los siguientes tipos:

```
T  ::=  Bool | Nat | T1 -> T2
```

Siendo,

- `Bool`: el tipo para los booleanos (e.g., `isZero(0): Bool`),
- `Nat`: el tipo para los naturales, (e.g., `succ(pred(0)): Bool`),
- `T1 -> T2`: el tipo para las funciones que van de `T1` a `T2`,
  por ejemplo, `\x:Bool.if x then succ(0) else pred(succ(succ(0))): Bool -> Nat`

Una expresión del cálculo lambda está en forma normal si no puede evaluarse más. Así, los valores están en forma normal, aunque no todops los términos que no pueden evaluarse más son valores. Finalmente, si tenemos un término cerrado (i.e., que no tiene variables libres) y que está bien tipado, podremos obtener un valor a partir de él.

Los valores para nuestro lenguaje serán los siguientes:

```
V  ::=  true | false | \x:T.M | n
```

con `n` como macro de $\texttt{succ}^n\texttt{(0)}$

3. Descripción del lenguaje de salida

Dada una cadena de entrada, se deberá evaluar si efectivamente respeta la sintaxis esperada de Cálculo Lambda y, si no hay algún de error de tipado[1], se deberá evaluar la expresión hasta llegar a un valor, e indicar el tipo del mismo.

3.1. Ejemplos

Entrada	Resultado	Salida
`0`	OK	`0:Nat`
`true`	OK	`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`	OK	`\x:Bool.if x then false else true:Bool->Bool`
`\x:Nat.succ(0)`	OK	`\x:Nat.succ(0):Nat->Nat`
`\z:Nat.z`	OK	`\z:Nat.z:Nat->Nat`
`(\x:Bool.succ(x)) true`	ERROR: succ espera un valor de tipo Nat	
`succ(succ(succ(0)))`	OK	`succ(succ(succ(0))):Nat`
`x`	ERROR: El término no es cerrado (x está libre)	
`succ(succ(pred(0)))`	OK	`succ(succ(0)):Nat`
`\x:Nat.succ(x)`	OK	`\x:Nat.succ(x):Nat->Nat`
`0 0`	ERROR: La parte izquierda de la aplicación (0) no es una función con dominio en Nat	
`\x:Nat->Nat.\y:Nat. (\z:Bool.if z then x y else 0)`	OK	`\x:Nat->Nat.\y:Nat.(\z:Bool.if z then x y else 0): (Nat->Nat)->(Nat->(Bool->Nat))`
`(\x:Nat->Nat.\y:Nat. (\z:Bool.if z then x y else 0)) (\j:Nat.succ(j)) 8 true`	OK	`9:Nat`

[1] Esto es, no se corresponde el tipo esperado con el recibido

4. Modo de uso

El programa deberá poder ejecutarse como un comando de consola del sistema operativo, con los siguientes requerimientos:

Sinopsis:

./CLambda [EXPRESION]

Descripción:

El programa deberá recibir una cadena con la expresión a evaluar (EXPRESION), y deberá devolver por *standard output* (`stdout`) el resultado de la evaluación.
Si no se especificara la cadena en la llamada, se esperará recibirla por *standard input* (`stdin`). En caso de que hubiera algún inconveniente al ejecutar el programa, se deberá terminar el programa con código de salida de error (esto es, un código mayor a 0), y mostrar los detalles por standard error (`stderr`).
En caso de que la llamada fuera correcta, se deberá devolver el resultado, y, si no fuera correcta la expresión, incluir todos los detalles del error por `stderr` y no retornar nada por `stdout`.

5. Implementación

Para el lenguaje descripto se solicita crear un analizador léxico, generando los tokens necesarios de acuerdo a la gramática que se haya diseñado, y un analizador sintáctico, que deberá generar el árbol sintáctico para la cadena procesada, en caso de que sea válida, o un mensaje de error adecuado, en caso de que no lo sea. Para este último caso, será deseable indicar el lugar preciso en el cual se encuentra el problema detectado.

Hay dos grupos de herramientas que se pueden utilizar para generar los analizadores léxicos y sintácticos:

- uno utiliza expresiones regulares y autómatas finitos para el análisis lexicografico, y la técnica LALR para el análisis sintáctico. Ejemplos de esto son lex y yacc, que generan código C o C++, JLex y CUP, que generan código Java y ply, que genera código Python. flex y bison son implementaciones libres y gratuitas de lex y yacc.
  En particular, permitiremos utilizar PLY 3.6[2] (o superior).
- el otro grupo utiliza la técnica ELL(k), tanto para el análisis léxico como para el sintáctico, generando parsers descendentes iterativos recursivos. Ejemplos son JavaCC y ANTLR, que están escritos en Java. JavaCC puede generar código Java o C++. ANTLR puede generar Java, C#, Python y JavaScript.
  En particular, para este trabajo se podrá usar ANTLR 4.X[3], y se recomienda utilizar el plugin[4] para Eclipse.

6. Detalles de la entrega

Se deberá enviar el código a la dirección de e-mail tptleng@gmail.com, satisfaciendo lo siguiente:

- el **asunto de mail** debe ser `[TL-TP]` seguido por el nombre del grupo (e.g., "[TL-TP] The Chomsky Boys").
- en el mail deberán estar copiados todos los/as integrantes del grupo.

[2] PLY: Python Lex & Yacc. Disponible en https://pypi.python.org/pypi/ply
[3] ANTLR (ANother Tool for Language Recognition): http://www.antlr.org
[4] ANTLR plugin for Eclipse: http://antlreclipse.sourceforge.net

La entrega debe incluir lo descripto a continuación:

- un programa que cumpla con lo solicitado,
- el código fuente del mismo, adecuadamente documentado,
- informe enviado por e-mail y entregado impreso, con los siguientes contenidos:
  - carátula con datos de integrantes del grupo y nombre del grupo,
  - breve introducción al problema a resolver,
  - los tokens, con sus expresiones regulares, y la gramática no ambigua definidos a partir del lenguaje propuesto,
  - indicación del tipo de la gramática definida, de acuerdo a los vistos en clase,
  - el código de la solución, y, si se usaron herramientas generadoras de código, imprimir la fuente ingresada a la herramienta, no el código generado,
  - descripción de cómo se implementó la solución, con decisiones que hayan tenido que tomar y justificación de las mismas,
  - información y requerimientos de software para ejecutar y recompilar el tp (versiones de compiladores, herramientas, plataforma, etc), como un pequeño manual del usuario, que además de los requerimientos, contenga instrucciones para compilarlo, ejectuarlo, información de parámetros y lo que consideren necesario,
  - casos de prueba con expresiones sintácticamente correctas e incorrectas (al menos tres para cada caso) y
  - un resumen de los resultados obtenidos, y conclusiones del trabajo.

Es parte de lo que se espera de la resolución del trabajo práctico la detección de puntos no especificados en el enunciado y su resolución. En cualquier caso, siempre pueden realizar consultas al respecto.

Referencias

[1] Descargas de la materia Paradigmas de Lenguajes de Programación, 1er cuatrimestre de 2017. Disponible en http://www.dc.uba.ar/materias/plp/cursos/2017/cuat1/descargas/.

[2] Aho, A.V., Lam, M.S., Sethi, R., *Compilers: Principles, Techniques and Tools - Second Edition*, Pearson Education, 2007.

[3] Goyvaerts, J., Levithan, S., *Regular Expressions Cookbook*, O'Reilly, 2009.

[4] Grune, D., Jacobs, C.J.H, *Parsing Techniques: A Practical Guide - Second Edition*, Springer, 2008.

[5] Hopcroft, J.E., Motwani, R., Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation - Third Edition*, Addison Wesley, 2007.

[6] Parr, T., *The Definitive ANTLR 4 Reference*, The Pragmatic Programmers, 2012.

[7] PLY (Python Lex-Yacc). Documentación. Disponible en http://www.dabeaz.com/ply/ply.html