Teoria de Lenguajes

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

TP1

Analizador Sintáctico y Semántico para λ^{bn}
July 5, 2017

Grupo:

Integrante	LU	Correo electrónico
Francisco Leto	249/09	leto.francisco@gmail.com
Ignacio Lebrero Rial		
Federico Beuter	827/13	federicobeuter@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Introduccion

En este infome se describe la implementacion de un analizador sintactico y semantico para λ^{bn}. En base al lenguaje presentado se procedio a construir una gramatica acorde, la cual luego fue utilizada para implementar el correspondiente Lexer y Parser. La implementacion fue realizado empleando Python junto con la libreria PLY (**P**ython **L**ex-**Y**acc) [1].

2 Gramatica

3 Lexer

El primer paso para la construcción de nuestra gramática fue la identificación de los *tokens* y palabras reservadas. Utilizamos las siguientes palabras reservadas y literales: **if, then, else, succ, pred, iszero, true, false, '(', ')', '.', '\', 0**. Definimos, ademas, los siguientes tokens, que se identificaron con las expresiones regulares detalladas a continuación.

Token	Expresion Regular
VAR_DECLARATION	[a-z|A-Z]+:((Nat|Bool)->(Nat|Bool)|Nat|Bool)
VAR_USAGE	[a-z|A-Z]+

Ademas, el lexer ignora los caracteres en blanco y descompone el token de VAR_DECLARATION en un objeto con el valor de la variable y su tipo. En la implementación, los literales se definieron como tokens para facilitar la implementación.

La gramática propuesta intenta ser lo mas simple posible, teniendo en cuenta la precedencia y asociatividad de los operadores, y evitando ambigüedades.

$$
\begin{aligned}
E &\rightarrow (E) \mid A \\
A &\rightarrow \backslash\ VAR\_DECLARATION.E \mid C \\
C &\rightarrow if\ E\ then\ E\ else\ E \mid P \\
P &\rightarrow E\ Q \mid F \\
Q &\rightarrow E \\
F &\rightarrow succ(E) \mid pred(E) \mid iszero(E) \mid G \\
G &\rightarrow 0 \mid true \mid false \mid V \\
V &\rightarrow VAR\_USAGE
\end{aligned}
$$

4 Analisis Semantico

Nuestro codigo, utilizando *PLY* construye un parser LARL para la gramática propuesta. Luego le agregamos semantica, que permite realizar el chequeo e inferencia de tipo, junto con la evaluación de las cadenas bien formadas. Parte de lo implementado consiste en:

- Asegurar que la guarda de la condición sea del tipo booleano.
- Asegurar que ambas ramas de la condición tengan el mismo tipo.

[1] PLY (Python Lex-Yacc): http://www.dabeaz.com/ply/ply.html

- Se asegura de que el tipo de la variable en la abstracción lambda sea valido.
- Se asegura de que `succ` y `pred` y `iszero` se apliquen solamente a valores de tipo `Nat`.
- Infiere los tipos cuando esto es posible.
- Respeta la precedencia y asociatividad al evaluar la cadenas de entrada.

5 Evaluaciones

5.1 Expresiones correctas

Expresion	Evaluacion
if true then (\x:Bool.false) else (\x:Bool.true)	\x:Bool.false:Bool->Bool
\x:Nat.iszero(x)	\x:Nat.iszero(x):Nat->Bool
pred(succ(succ(0)))	succ(0):Nat

5.2 Expresiones incorrectas

Expresion	Salida
if true then false else (\x:Bool.true)	ERROR: Las dos opciones del if deben tener el mismo tipo
\x.Not.succ(x)	Hubo un error en el parseo. Sintaxis invalida
succ(iszero(true))	ERROR: iszero espera un valor de tipo Nat

6 Manual de Usuario

Para poder ejecutar el parser, es necesario emplear Python 3 con PLY instalado. Una vez que se tienen todas las dependencias, se puede ejecutar con cualquiera de los siguientes dos comandos:

- `python CLambda.py`
- `python CLambda.py expresionLambda`

El segundo recibe la expresion lambda directamente y la evalua, mientras que el primero recibe la expresion por `stdin` mediante la consola. Los resultados de la evaluacion se imprimen por `stdout` en caso de que sea satisfactoria, en el caso contrario se imprime el motivo de error por `stderr`.

7 Código fuente

7.1 CLambda.py

```
#pylint: disable=C0103,C0111
"Script principal para ejecutar el parser."

import sys
from parser_processing import process_entry

if __name__ == "__main__":
```

```
    script_input = ""
    if len(sys.argv) > 1:
        #Uno la entrada con espacios para evitar problemas en el shell.
        script_input = " ".join(sys.argv[1:])
    else:
        script_input = input("lambda> ")

    try:
        print(process_entry(script_input))
    except KeyError as e:
        print("ERROR: El termino no es cerrado (%s esta libre)." %(e), file=sys.stderr)
        sys.exit(1)
    except TypeError as e:
        print("ERROR: " + str(e), file=sys.stderr)
        sys.exit(1)
```

7.2 parser_processing.py

```
#pylint: disable=C0103,C0111
import re

from ply.lex import lex
from ply.yacc import yacc

import lambda_lexer
import lambda_parser
from expressions import get_type_str, Lambda
from lambda_lexer import global_context

def hack_numbers(text, result):
    "cambia los numeros por combinación de succ cuando corresponde"
    def calc_succ(match):
        i = int(match.group(0))
        res = "0"
        for _ in range(i):
            res = "succ(" + res
        res += ")"*i
        return res
    if not re.match(r".*[1-9]\d*", text):
        return re.sub(r"([1-9]\d*)", calc_succ, result)
    else:
        return result

def process_entry(text):
    lexer = lex(module=lambda_lexer)
    parser = yacc(module=lambda_parser, debug=True)

    expression = parser.parse(text, lexer)
    result = expression.evaluate()
    if isinstance(result, Lambda):
```

```
        res_type = result.type
    else:
        res_type = type(result)
    global_context.clear()
    #Hago las conversiones necesarias para mostrar bien los tipos
    if isinstance(res_type, tuple):
        a = get_type_str(res_type[0])
        b = get_type_str(res_type[1])
        if isinstance(res_type[0], tuple):
            a = "(" + a + ")"
        if isinstance(res_type[1], tuple):
            b = "(" + b + ")"
        res_type = "%s->%s" %(a, b)
    else:
        res_type = get_type_str(res_type)
    return "%s:%s" %(hack_numbers(text, str(result)), res_type)
```

7.3 lambda_lexer.py

```
#pylint: disable=C0103,C0111
"Lexer para calculo lambda con Bool y Nat"

global_context = {}

def convert(var_type):
    if var_type == "Nat":
        return int
    elif var_type == "Bool":
        return bool
    else:
        a, b = var_type.split("->")
        return (convert(a.strip()), convert(b.strip()))

#Palabras reservadas
reserved_keywords = {
    'if'     : 'IF',
    'then'   : 'THEN',
    'else'   : 'ELSE',
    'succ'   : 'SUCC',
    'pred'   : 'PRED',
    'iszero' : 'ISZERO',
    'true'   : 'TRUE',
    'false'  : 'FALSE'
}

#Defino los tokens
tokens = [
    "VAR_DECLARATION",
    "VAR_USAGE",
    "LAMBDA",
```

```
    "POINT",
    "L_BRACKET",
    "R_BRACKET",
    "ZERO",
] + list(reserved_keywords.values())

t_LAMBDA = r"\\"
t_POINT = r"\."
t_L_BRACKET = r"\("
t_R_BRACKET = r"\)"

# Espacios y tabs
t_ignore_WHITESPACES = r"[ \t]+"

def t_ZERO(t):
    r"0"
    t.value = int(t.value)
    return t

def t_error(t):
    message  = "\n------------------ Illegal character ------------------"
    message += "\ntype:"  + t.type
    message += "\nvalue:" + str(t.value)
    message += "\nline:"  + str(t.lineno)
    message += "\nposition:"  + str(t.lexpos)
    message += "\n-------------------------------------------------------\n"

    print(message)
    t.lexer.skip(1)

def t_VAR_DECLARATION(t):
    r"[a-z|A-Z]+:((Nat|Bool)->(Nat|Bool)|Nat|Bool)"
    var, var_type = t.value.split(":")
    var_type = var_type.strip()
    t.value = var
    t.var_type = convert(var_type)
    global_context[t.value] = t.var_type
    return t

def t_VAR_USAGE(t):
    r"[a-z|A-Z]+"
    t.type = reserved_keywords.get(t.value,'VAR_USAGE')
    if t.value in global_context:
        t.var_type = global_context[t.value]
    return t
```

7.4 lambda_parser.py

```
#pylint: disable=C0103,C0111,W0611
"El parser para calculo lambda"
```

```
import sys
import ply.yacc as yacc

from operator import add, is_

from lambda_lexer import tokens
from expressions  import *

######################## Expresiones ########################

precedence = [
    ('right', 'APP')
]

def p_brackets(p):
    "exp : L_BRACKET exp R_BRACKET"
    p[0] = Brackets(p[2])

def p_exp_abs(p):
    "exp : abs"
    p[0] = p[1]

def p_abs(p):
    "abs : LAMBDA dec POINT exp"
    p[0] = Lambda(p[2], p[4])

def p_abs_con(p):
    "abs : con"
    p[0] = p[1]

def p_conditional(p):
    "con : IF exp THEN exp ELSE exp"
    p[0] = ConditionalOperation(p[2], p[4], p[6])

def p_con_app(p):
    "con : app"
    p[0] = p[1]

def p_application(p):
    "app : exp exp %prec APP"
    p[0] = Application(p[1], p[2])

#def p_exp2_exp(p):
#    "exp2 : exp"
#    p[0] = p[1]

def p_app_fun(p):
    "app : fun"
    p[0] = p[1]
```

```
def p_is_zero(p):
    "fun : ISZERO L_BRACKET exp R_BRACKET"
    p[0] = NatOperation(p[3], Number(0), is_)

def p_pred(p):
    "fun : PRED L_BRACKET exp R_BRACKET"
    p[0] = NatOperation(p[3], Number(-1), add)

def p_succ(p):
    "fun : SUCC L_BRACKET exp R_BRACKET"
    p[0] = NatOperation(p[3], Number(1), add)

def p_fun_val(p):
    "fun : val"
    p[0] = p[1]

def p_cero(p):
    "val : ZERO"
    p[0] = Number(p[1])

def p_true(p):
    "val : TRUE"
    p[0] = Boolean(p[1])

def p_false(p):
    "val : FALSE"
    p[0] = Boolean(p[1])

def p_val_var(p):
    "val : var"
    p[0] = p[1]

def p_var(p):
    "dec : VAR_DECLARATION"
    p[0] = Var(p[1])

def p_var_from_exp(p):
    "var : VAR_USAGE"
    p[0] = Var(p[1])

def p_error(_):
    print("Hubo un error en el parseo. Sintaxis invalida", file=sys.stderr)
    sys.exit(1)
```

7.5 expressions.py

```
#pylint: disable=C0103,C0111,W0611
"Clases que utiliza el parser para identificar las expresiones"
from operator import is_, add
from lambda_lexer import global_context
```

```
def get_type_str(t):
    if hasattr(t, "type"):
        t = t.type
    if t == bool:
        return "Bool"
    elif t == int:
        return "Nat"
    else:
        # Es una tupla
        a = b = None
        if isinstance(t[0], tuple):
            a = "(%s)" %(get_type_str(t[0]))
        else:
            a = get_type_str(t[0])
        if isinstance(t[1], tuple):
            b = "(%s)" %(get_type_str(t[1]))
        else:
            b = get_type_str(t[1])
        return "%s->%s"%(a, b)

def get_input_type_of_function(t):
    if t[0] == int:
        return Number(0)
    elif t[0] == bool:
        return Boolean("true")
    elif isinstance(t[0], tuple):
        return Function(t[0])

def checkType(a, b, err):
    if isinstance(a, Function):
        for i, _ in enumerate(a.type):
            checkType(a.type[i], b.type[i], err)
    else:
        try:
            if a.type != b.type:
                raise TypeError(err)
        except AttributeError:
            if a != b:
                raise TypeError(err)

class Expression(object):
    def evaluate(self):
        raise NotImplementedError

class Brackets(Expression):
    def __init__(self, exp):
        self.exp = exp
        self.type = self.exp.type
```

```
    def evaluate(self):
        return self.exp.evaluate()

    def __str__(self):
        return "(" + str(self.exp) + ")"

class Application(Expression):
    def __init__(self, expression1, expression2):
        exp1_type = None
        try:
            exp1_type = get_input_type_of_function(expression1.type)
        except TypeError:
            raise TypeError("La parte izquierda de la aplicación (%s) no es una función con dominio %s"
                %(str(expression1), get_type_str(expression2.type)))
        checkType(exp1_type, expression2,
            "La función lambda espera un parametro de tipo %s. Recibio %s"
            %(get_type_str(exp1_type.type), get_type_str(expression2.type)))
        self.expression1 = expression1
        self.expression2 = expression2
        self.type = expression1.type[1]

    def evaluate(self):
        self.expression1 = self.expression1.evaluate()
        return self.expression1.evaluate(self.expression2.evaluate())

    def __str__(self):
        return str(self.expression1) + " " + str(self.expression2)

class ConditionalOperation(Expression):
    def __init__(self, condition, left_branch, right_branch):
        checkType(left_branch, right_branch, "Las dos opciones del if deben tener el mismo tipo")
        checkType(condition, Boolean("true"), "La condición debe ser booleana")

        self.condition = condition
        self.left_branch = left_branch
        self.right_branch = right_branch
        self.type = left_branch.type

    def evaluate(self):
        if self.condition.evaluate():
            return self.left_branch.evaluate()
        else:
            return self.right_branch.evaluate()

    def __str__(self):
        return "if " + str(self.condition) + " then " + \
             str(self.left_branch) + " else " + str(self.right_branch)

class NatOperation(Expression):
    def __init__(self, val, added_val, op):
```

```
        if op == is_:
            self.type = bool
        else:
            self.type = int

        self.op = op
        self.val = val
        self.added_val = added_val

        self.op_str = None
        if op == is_:
            self.op_str = "iszero"
        elif op == add and self.added_val.value > 0:
            self.op_str = "succ"
        else:
            self.op_str = "pred"

        checkType(val, added_val, self.op_str + " espera un valor de tipo Nat")

    def evaluate(self):
        val1 = self.val.evaluate()
        val2 = self.added_val.evaluate()
        evaluation = self.op(val1, val2)
        self.type = type(evaluation)
        self.value = evaluation
        return self.value

    def __str__(self):
        return self.op_str + "(" + str(self.val) + ")"

############### Atomic expressions ###############

class Boolean(Expression):
    def __init__(self, value):
        self.type = bool
        self.value = (value == "true")

    def evaluate(self):
        return self.value

    def __str__(self):
        if self.value:
            return "true"
        else:
            return "false"

class Number(Expression):
    def __init__(self, value):
        self.type = int
        self.value = int(value)
```

```
    def evaluate(self):
        return self.value

    def __str__(self):
        return str(self.value)

class Function(Expression):
    def __init__(self, tuple):
        self.type = tuple

class Var(Expression):
    def __init__(self, value):
        self.value = value
        self.type = global_context[value]

    def evaluate(self):
        return global_context[self.value]

    def __str__(self):
        return self.value

class Lambda(Expression):
    def __init__(self, variable, expression):
        self.type = (variable.type, expression.type)
        self.variable = variable.value
        self.expression = expression

    def evaluate(self, x=None):
        if x:
            global_context[self.variable] = x
            return self.expression.evaluate()
        else:
            return self

    def __str__(self):
        return "\\" + self.variable + ":" + get_type_str(self.type[0]) + \
            "." + str(self.expression)
```