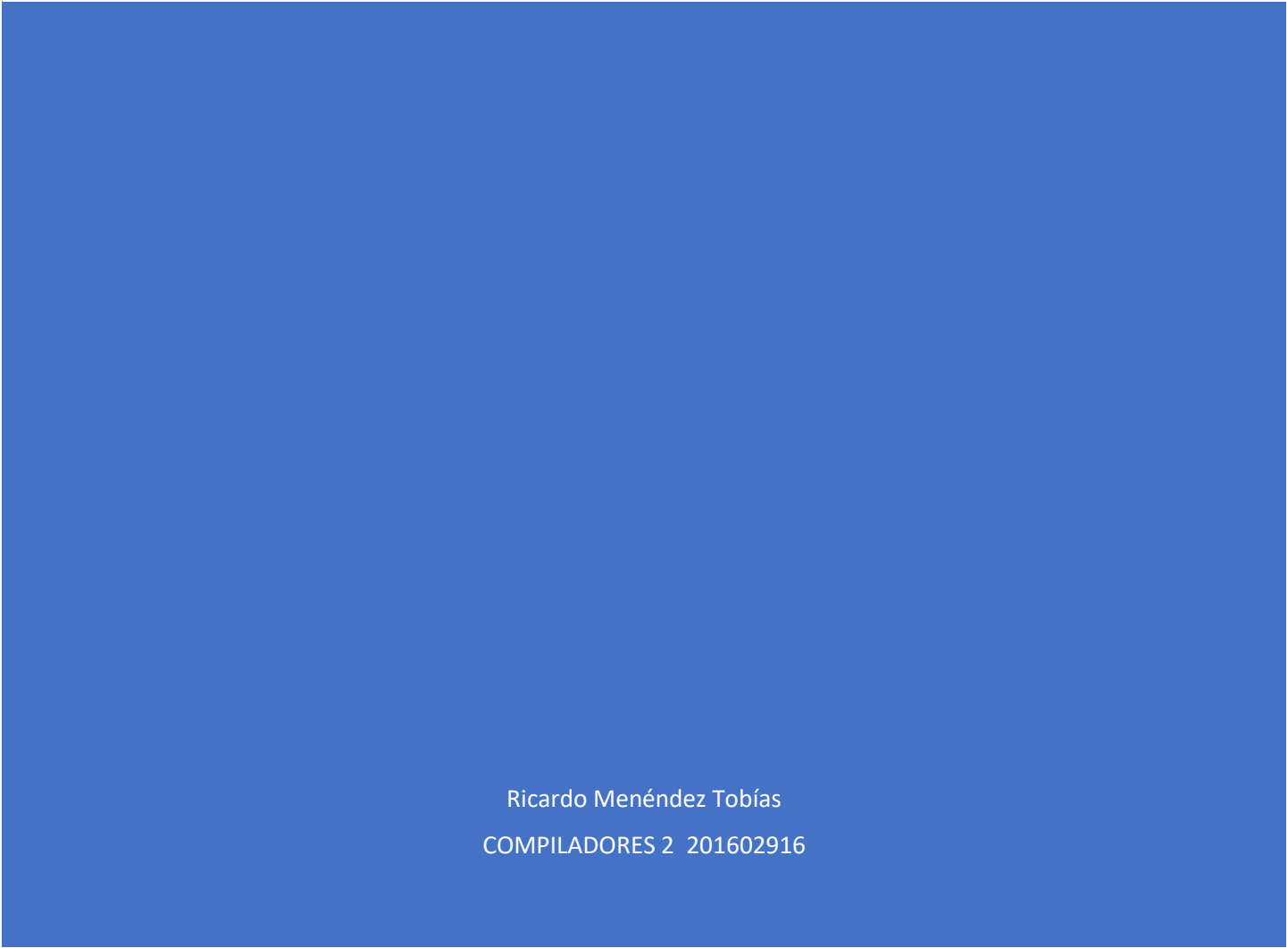




MANUAL TECNICO



Ricardo Menéndez Tobías
COMPILADORES 2 201602916

Contenido

MinorC	2
Herramientas Utilizadas:.....	2
PLY.....	2
Análisis Lexico	2
Análisis Sintactico.....	4
Patrón Interprete	5
Estructura.....	5
Funciones:	6
Clases:	6
Interfaz	8

MinorC

MinorC es un lenguaje de programación que es una versión simplificada de C. Este es capaz de realizar diferentes ciclos, sentencias de control, manejo de ámbitos, estructuras, arreglos y llamado a métodos. Su creación fue con la finalidad de poder dar un lenguaje fuente, el cual se pueda traducir a Augus, un lenguaje que simula el código intermedio.

Herramientas Utilizadas:

- Python 3
- PLY
- Tkinter
- PyQt

PLY

Análisis Lexico

PLY es una librería de Python la cual sirve para el análisis léxico y sintactico.

La estructura de un archivo de estos, es crear primero todas las reglas léxicas. Para esto se deben definir las palabras reservadas en un diccionario, con el lexema como clave de la palabra.

```
reservadas = {  
    'main': 'main',  
    'goto': 'goto',  
    'int': 'int',  
    'float': 'float',  
    'char': 'char',  
    'print': 'print',  
    'array': 'array',  
    'unset': 'unset',  
    'if': 'if',  
    'abs': 'abs',  
    'xor': 'xor',  
    'exit': 'exit',  
    'read': 'read'  
}
```

Y luego se define en una lista, el nombre de todos los Tokens declarados en el análisis léxico, concatenando las palabras reservadas.

```
'DOUBLE',  
'INTEGER',  
'VAR',  
'STR'  
] + list(reservadas.values())
```

Luego, se define los tokens con expresiones regulares. Estos deben tener como nombre la siguiente sintaxis. `t_NombreDefinidoEnLaLista`.

```
t_RESTA = r'-'
t_SUMA = r'\+'
t_MULT = r'\*'
t_DIV = r'\/'
t_PORCENTAJE = r'\%'
```

También se pueden definir como funciones, siendo el parámetro t el lexema enviado.

```
def t_DOUBLE(t):
    r'\d+\.\d+'
    try:
        t.value = float(t.value)
    except ValueError:
        print("Valor no es parseable a decimal %d",t.value)
        t.value = 0
    return t

def t_STR(t):
    r'\'.*?\''
    t.value = t.value[1:-1]
    return t
```

Como ultimo paso, tenemos que definir los lexemas a ignorar, algo que querramos tener en cuenta (como saltos de línea) o los errores, en el caso de este proyecto, se define la nueva línea para contarlas, el comentario, el error y se ignora el \t

```
def t_COMENTARIO(t):
    r'\#.*\n'
    t.lexer.lineno += 1

t_ignore = " \t"

def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")

def t_error(t):
    print("Caracter irreconocible! '%s'" % t.value[0])
    #meter a tabla de errores!
    erroresLexicos.append("Error Lexico: "+t.value[0]+" en ")
    t.lexer.skip(1)
```

Ahora para utilizar todas estas reglas gramaticales, debemos importar el LEX de PLY. Esto con el comando

Import ply.lex as lex

Y con esto podremos inicializar el lexer.

Lexer = lex.lex()

Con esto, ya tenemos generado nuestro analizador Léxico.

Análisis Sintactico

Para el análisis sintactico pasa algo parecido con la declaración de tokens, solo que ahora los declaramos con `p_nombreDeLaProduccion`, luego en un string ponemos la producción gramatical que queremos definir, y enviando un parámetro, accedemos a cada token. Este parámetro tiene en la posición [0] el lado izquierdo de la producción y en las siguientes posiciones los terminales o no terminales definidos en la gramática del string.

Siendo algo así:

```
def p_s_tag(t):
    '''s : ltag'''
    t[0] = Root(t[1])

def p_lista_tag(t):
    '''
        ltag : ltag tag linst
        | tag linst
    '''
    if(len(t)==3):
        t[0] = {}
        t[0][t[1]] = (Tag(t[1],t[2],t.lineno(1)))
    if(len(t)==4):
        t[0] = t[1]
        t[0][t[2]] = (Tag(t[2],t[3],t.lineno(1)))
```

El primer método declarado, va a ser el símbolo inicial del análisis.

Ahora para la ejecución del parser sintactico, se debe importar el yacc de PLY. Eso con el siguiente código:

```
Import ply.yacc as yacc
```

Y ya para ejecutarlo, solo inicializamos una variable con el yacc y llamamos al método parser.

```
Parser = yacc.yacc()
```

```
Parser.parse("CADENA A ANALIZAR", tracking=True)
```

*este segundo parámetro sirve para darle seguimiento al # de línea.

Patrón Interprete

Estructura

El Patrón Interprete consiste de una serie de clases, que heredan todos de una interfaz, que representa un nodo terminal o no terminal de una gramática. En el caso del proyecto se maneja la interfaz Nodo.

Esta interfaz posee 2 metodos abstractos el cual cada nodo debe cumplir. Una función la cual ejecuta la funcioanlidad de cada Nodo, y una función la cual devuelve el Nodo en lenguaje DOT para la generación del reporte AST, otra que devuelve el string del reporte gramatical y otra que da, la del GDA.

A diferencia de la ejecución de Augus, este código se enfrasca en sintentizar código, no en la ejecución. Con la finalidad de ir evaluando como se pueden transformar diferentes instrucciones de alto nivel, en un código intermedio.

Luego este es desglosado en 2 principales ramas de la gramática. En instrucciones y Expresiones. Ya que son las 2 derivaciones que puede tomar el nodo en la gramática.

En Expresiones va cualquier operación que puede resolver el lenguaje, así como los tipos de datos.

Por el otro lado, las instrucciones son partes de un cuerpo, el cual pertenece a cada etiqueta, un cuerpo.

Luego, cada ejecutar, recibe como parámetro, a una de las clases mas importantes de todo el proyecto, que es Entorno. Esta posee la tabla de simbolos, de métodos, lista de mensajes y errores. De modo que esta se va pasando en cada nodo y va transmitiendo la información a lo largo de todo el programa.

Luego tenemos los valores que existen en el programa, que están representados por una clase abstracta Valor, todos estos valores pueden ser:

Valores:

- Integer: Numero entero. Esta representado por un int de Python.
- Numeric: Numero con valores decimales. Esta representado por un float de Python, aunque en C minor esta representado por el Float y Double.
- String: Una cadena de caracteres empezada y terminada por el carácter '.

Estructuras de Datos (También valores): C Minor ofrece una Estructuras de Datos para guardar información.

- Array: El vector es un conjunto representado linealmente de N datos primitivos. Todos los datos deben ser del mismo tipo. Este esta representado en Python como un diccionario.
- Structs: Colección de tipos de datos.

Las instrucciones son:

Bloques de Sentencias :

- If: Esta sentencia recibe una expresión, y un cuerpo al menos. Puede verse acompañado de un grupo de ELSE IF, que contienen también una expresión y un cuerpo, y aparte, puede venir con un else, el cual solo tiene su respectiva lista de instrucciones.
- Asignación: Se indica una variable y una expresión. Se ejecuta la expresión y se inserta la variable en la tabla de símbolos.
- Declaración: Se indica una variable y se ingresa a la tabla de símbolos.
- While: Un ciclo el cual recibe una condición por medio de una expresión. El cuerpo del ciclo se ejecuta mientras esta sea cierta. En el caso del código intermedio, se crea un salto condicional a la evaluación de la condición, si esta se cumple, se repite el proceso, si no, se salta a la siguiente instrucción.
- DoWhile: Funciona igual que el while, solo que la condición se evalúa luego de ejecutar el cuerpo.
- Switch: Un ciclo de control el cual nos permite dar con una variable, la elección de un cuerpo de código.

Funciones:

mINORc tiene sus propias funciones, las cuales son:

- Readf() : Lee una entrada del usuario. Si es un número o double, los devuelve de ese tipo.
- Printf(CADENA [, EXPRESION]*): Imprime en consola el resultado de la expresión. Esto añadiendo un mensaje a la tabla de símbolos.
-

Clases:

- Tabla de Símbolos

La tabla de símbolos contiene todos los datos que se deben guardar durante la ejecución del código, con la finalidad de proveer al código, una fuente de donde extraer datos como las variables, métodos o guardar información para reportes.

```

class Error():
    def __init__(self, valor, linea):
        self.valor = valor
        self.linea = linea

class Simbolo():
    def __init__(self, name, valor, temp, linea):
        self.nombre = name
        self.tipo = valor
        self.linea = linea
        self.temp = temp

class Metodos():
    def __init__(self, name, params, body, linea):
        self.nombre = name
        self.params = params
        self.body = body
        self.linea = linea

class Extra():
    def __init__(self, retorno, ciclo, ex):
        self.retorno = retorno
        self.ciclo = ciclo
        self.ex = ex

class TablaSimbolos():
    def __init__(self):
        self.contadorTemp = 0
        self.contadorParam = 0
        self.contadorTag = 0
        self.errorSemantico = []
        self.mensajes = []
        self.tsReport = []
        self.variables = []
        self.variables.append(dict())
        self.metodos = {}
        self.optimizacion = []

```

- Estructura de un Nodo:

```

class ValorEntero():
    def __init__(self, valor):
        self.valor = valor

    def ejecutar(self, ts, ex):
        try:
            v = int(self.valor)
            return valorTemporal(TIPO_TEMP.VALOR, "", str(self.valor), TIPO.ENTERO, "", "")
        except:
            return valorTemporal(TIPO_TEMP.ERROR, "No se pudo volver Entero", "", TIPO.ERROR, "", "")

    def ast(self):
        node = str(getHash(self))
        v = "n"+node+"\n n"+node+"[label = \" Entero : "+str(self.valor)+"\"]"
        return v

    def gda(self, nodo):
        node = str(nodo) + str(getHash(self.valor))
        v = "n"+node+"\n n"+node+"[label = \" Entero : "+str(self.valor)+"\"]"
        return v

    def grammar(self):
        v = (''
        <tr>
        <td> <p>EXP => entero <br/> </p></td>
        <td><p>
        t[0] = ValorEntero(t[1]) </p></td>
        </tr>'')
        return v

```


Interfaz

La interfaz fue desarrollada con TKinter. Esta posee un Menu, con varias opciones, donde cada opción, llama a un método.

Y un widget tipo Text, de donde se obtiene todo el texto. De misma forma se muestra en una consola del mismo tipo.

Y se utilizaron las dependencias de las siguientes librerías:

- PyQt5
- time
- threading
- sys
- re
- tkinter
- webbrowser
- os