



Tecnológico
de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Desarrollo de Aplicaciones Avanzadas

José Manuel Castillo Guajardo
A01284149

Profesores

Elda Quiroga

Entregable

Entrega #1: Little Duck

Fecha de entrega

29 / mayo / 2024

Entrega #1

Gramática

<TYPE> → int

<TYPE> → float

<list_ids> → id<mas_ids>

<mas_ids> → ,<list_ids>

<mas_ids> → ε

<VARS> → var <variables> : <TYPE> ; <mas_var>

<variables> → <list_ids> :

<mas_var> → ε

<mas_var> → <variables>

<Programa> → program id ; <dec_vars> <dec_f> main <Body> end

<dec_v> → ε

<dev_v> → <VARS>

<dec_f> → ε

<dec_f> → <FUNCS> <mas_f>

<mas_f> → ε

<mas_f> → <FUNCS> <mas_f>

<Body> → { <mas_statement> }

<mas_statement> → ε

<mas_statement> → <statement><mas_statement>

<STATEMENT> → <ASSIGN>

<STATEMENT> → <CONDITION>

<STATEMENT> → <CYCLE>

<STATEMENT> → <F_CALL>

<STATEMENT> → <PRINT>

<ASSIGN> → id = <EXPRESION>;

<EXPRESION> → <EXP> <mas_expresion>

<mas_expresion> → > <EXP>

<mas_expresion> → < <EXP>

<mas_expresion> → != <EXP>

<mas_expresion> → ε

<EXP> → <TERMINO> <mas_exp>

<mas_exp> → + <TERMINO>

<mas_exp> → - <TERMINO>

<mas_exp> → ε

<TERMINO> → <FACTOR> <mas_termino>

<mas_termino> → * <FACTOR>
<mas_termino> → / <FACTOR>
<mas_termino> → ϵ

<CTE> → cte_int
<CTE> → cte_float

<PRINT> → print <EXPRESION> <mas_print>);
<mas_print> → , <EXPRESION> <mas_print>
<mas_print> → cte.string <mas_print>
<mas_print> → ϵ

<CYCLE> → while <BODY> do (<EXPRESION>);

<CONDITION> → if (<EXPRESION>) <BODY> <mas_condition> ;
<mas_condition> → ϵ
<mas_condition> else <BODY>

<F_Call> → id (<EXPRESION> <mas_expresion>);
<mas_expresion> → , <EXPRESION>
<mas_expresion> → ϵ

<FUNCS> → void id (<PARAMS>) [<VARS> <BODY>] ;

<params> → , id : <TYPE> <params>
<params> → , id : <TYPE> <mas_params>
<params> → ϵ
<mas_params> → , <params>

<FACTOR> → <mas_factor>

<mas_factor> → (<EXPRESION>)
<mas_factor> → <signo> id
<mas_factor> → <signo> <CTE>
<mas_factor> → + <CTE>
<mas_factor> → - <CTE>

<signo> → +
<signo> → -
<signo> → ϵ

PLY (Python Lex-Yacc)

PLY es una implementación en Python de las herramientas lex y yacc que son comúnmente utilizadas para escribir parsers y compiladores. El parsing está basado en el algoritmo LALR utilizado en muchas herramientas yacc. PLY tiene un reporte y diagnóstico de errores robusto que ayuda a en la construcción efectiva del parser. Tiene soporte para lenguajes con cientos de reglas, como lo podría ser un lenguaje como C.

Para la realización de este proyecto se tomó en cuenta PLY y ANTLR para el desarrollo del scanner y el parser. Se decidió utilizar PLY porque es una librería que fue creada específicamente para el curso de introducción a los compiladores de la Universidad de Chicago en el 2001. Tiene únicamente las cualidades del core de LALR, esto ofreciendo una mayor rigidez en cuanto al diseño de la gramática. Esto ayuda a los estudiantes a aprender de manera más profunda el cómo verdaderamente funciona un compilador.

La documentación se puede encontrar en el siguiente enlace:

<https://ply.readthedocs.io/en/latest/>

Gramática en PLY

```
# Palabras reservadas
reserved = {
    'if'      : 'IF',
    'else'    : 'ELSE',
    'program' : 'PROGRAM',
    'main'    : 'MAIN',
    'end'     : 'END',
    'var'     : 'VAR',
    'int'     : 'INT',
    'float'   : 'FLOAT',
    'print'   : 'PRINT',
    'void'    : 'VOID',
    'while'   : 'WHILE',
    'do'      : 'DO',
    'if'      : 'IF',
    'else'    : 'ELSE',
}

# Lista de tokens
tokens = [
    'ID',
    'SEMI_COLON',
    'OPEN_PAR',
    'CLOSE_PAR',
    'DIVIDE',
    'MULTIPLY',
    'ADD',
    'SUBTRACT',
    'LESS_THAN',
    'MORE_THAN',
    'NOT_EQUAL',
    'COLON',
    'L_BRACKET',
    'R_BRACKET',
    'LEFT_CURLY',
    'RIGHT_CURLY',
    'COMMA',
    'EQUAL',
    'CTE_STRING',
    'CTE_FLOAT',
    'CTE_INT',
] + list(reserved.values())
```

Las palabras reservadas y los tokens se definen en el lenguaje como se aprecia en las figuras. Definiendo en un diccionario cada palabra reservada y en una lista los tokens que se utilizan para el lenguaje.

```
# Define reglas para cada token
t_SEMI_COLON = r';'
t_OPEN_PAR = r'\('
t_CLOSE_PAR = r'\)'
t_DIVIDE = r'/'
t_MULTIPLY = r'\*'
t_ADD = r'\+'
t_SUBTRACT = r'\-'
t_LESS_THAN = r'<'
t_MORE_THAN = r'>'
t_NOT_EQUAL = r'!='
t_COLON = r':'
t_L_BRACKET = r'\['
t_R_BRACKET = r'\]'
t_LEFT_CURLY = r'\{'
t_RIGHT_CURLY = r'\}'
t_COMMA = r','
t_EQUAL = r'='
```

Después de esto se tienen que definir las reglas de expresión regular para cada uno de los tokens. De esta manera el scanner los puede identificar.

```
def p_programa(p):
    """programa : PROGRAM ID SEMI_COLON dec_v dec_f MAIN body END
    """

def p_vars(p):
    """vars : VAR variables COLON type SEMI_COLON mas_var
    | empty"""

def p_variables(p):
    """variables : list_ids
    """

def p_mas_var(p):
    """mas_var : variables
    | empty"""
```

En PLY cada regla tiene su propia función en la que se escribe la gramática de la misma. Como se observa p_programa sería el equivalente a <Programa> → program id ; <dec_vars> <dec_f> main <Body> end. En esta las palabras en mayúscula se refieren a los tokens o palabras reservadas. Las palabras en minúscula se refiere a otras reglas gramaticales (otra función). En este caso también el uso de “|” indica un “or” en la regla gramatical. La palabra “empty” sería el equivalente al ϵ .

Test Plan

El test plan que se desarrolló es el siguiente, dónde se prueba el scanner y el parser desarrollado, probando las reglas gramaticales definidas.

```
program HelloWorld;

var i : int;

void myFunc(param1 : float) [
    var x : int;
    {
        x = 0;
        do {
            x = x + 1;
            print(x);
        } while (x < 6);
    }
];

void myFunc2(param1 : float) [
    {
        if (param1 > 2) {
            print("Greater than");
        } else {
            print("Not greater than");
        };
    }
];

main
{
    i = 45 * 2 + 1 / 33;
    myFunc(45);
    myFunc2(1);
    print("Hello World!");
}
end
```

Cómo resultado obtenemos la detección correcta de cada token o palabra reservada (no se muestran todos en la imagen) y en el análisis semántico obtenemos un None, que significa que no hubo errores ya que es un test correcto.

```
LexToken(CTE_INT,6,12,182)
LexToken(CLOSE_PAR,')',12,183)
LexToken(SEMI_COLON,';',12,184)
LexToken(RIGHT_CURLY,'}',13,190)
LexToken(R_BRACKET,']',14,192)
LexToken(SEMI_COLON,';',14,193)
LexToken(VOID,'void',16,196)
LexToken(ID,'myFunc',16,201)
LexToken(OPEN_PAR,'(',16,208)
LexToken(ID,'param1',16,209)
LexToken(COLON,':',16,216)
LexToken(FLOAT,'float',16,218)
LexToken(CLOSE_PAR,')',16,223)
LexToken(L_BRACKET,['',16,225)
LexToken(LEFT_CURLY,'{',17,231)
LexToken(IF,'if',18,241)
LexToken(OPEN_PAR,'(',18,244)
LexToken(ID,'param1',18,245)
LexToken(MORE_THAN,'>',18,252)
LexToken(CTE_INT,2,18,254)
LexToken(CLOSE_PAR,')',18,255)
LexToken(LEFT_CURLY,'{',18,257)
LexToken(PRINT,'print',19,271)
LexToken(OPEN_PAR,'(',19,276)
```

```
        if (param1 > 2) {
            print("Greater than");
        } else {
            print("Not greater than");
        };
    }
];

main
{
    i = 45 * 2 + 1 / 33;
    myFunc(45);
    myFunc2(1);
    print("Hello World!");
}
end
None
```

Por otro lado, si quitamos un punto y como de una de las líneas en main, saldría el siguiente error. Un error de semántica en myFunc2, ya que previamente debería tener ese punto y como que se eliminó.

```
main
{
    i = 45 * 2 + 1 / 33;
    myFunc(45)
    myFunc2(1);
    print("Hello World!");
}
LexToken(SEMI_COLON, ';', 31, 459)
LexToken(RIGHT_CURLY, '}', 32, 461)
LexToken(END, 'end', 33, 463) "Not greater than"
Semantic error LexToken(ID, 'myFunc2', 62, 422)
```

Entrega #2: Directorio de Funciones y Tablas de variables

Para guardar registro de las funciones creadas y las variables declaradas en ellas, se creó el siguiente esquema. Este esquema también incluye las variables globales.

```
1 class FunctionTable:
2     def __init__(self):
3         self.variables = VariableTable()
4
5 class VariableTable:
6     def __init__(self):
7         self.symbols = {}
8
9     def add_variable(self, name, data_type):
10        if name in self.symbols:
11            raise ValueError(f"Variable '{name}' is already declared in this scope")
12        self.symbols[name] = {'type': data_type, 'value': None}
13
14    def set_variable_value(self, name, value):
15        if name in self.symbols:
16            self.symbols[name]['value'] = value
17        else:
18            raise KeyError(f"Variable '{name}' not found in symbol table")
19
20    def get_variable_value(self, name):
21        if name in self.symbols:
22            return self.symbols[name]['value']
23        else:
24            raise KeyError(f"Variable '{name}' not found in symbol table")
25
```

Se tienen dos clases, una llamada “FunctionTable” y “VariableTable”. Se dividió así para una posible extensión principalmente de “FunctionTable”, esta podrá tener todas las características que se tengan que guardar de una función declarada en el programa.

La clase “VariableTable” tiene como objetivo guardar las variables (nombre, tipo, valor) en un diccionario. Se tienen varios metodos en la clase.

add_variable(self, name, data_type)

Esta función tiene como objetivo agregar una variable al diccionario. Si la variable está repetida, se lanza un error.

set_variable_value(self, name, value)

Esta función tiene como objetivo actualizar el valor de alguna variable en el diccionario. Si la variable no se encuentra en el diccionario, se lanza un error.

get_variable_value(self, name)

Esta función tiene como objetivo obtener el valor de alguna variable. Si la variable no se encuentra en el diccionario, se lanza un error.

```
Function name: global
k {'type': 'int', 'value': None}
j {'type': 'int', 'value': None}
i {'type': 'int', 'value': None}
hello {'type': 'float', 'value': None}

Function name: myFunc
z {'type': 'int', 'value': None}
y {'type': 'int', 'value': None}
x {'type': 'int', 'value': None}

Function name: myFunc2
```

Este es un ejemplo de cómo se almacenan las variables. Cada función tiene sus variables con un identificador, el tipo y el valor.

Si alguna variable se encuentra redeclarada en un mismo scope, se lanza el siguiente error.

```
ValueError: Variable 'x' is already declared in this scope
```


Entrega #3: Generación de Código Intermedio

La clase `QuadrupleGenerator` se creó principalmente para realizar los cuádruplos de expresiones. La función principal es `generate()`, que se llamaría con un string de alguna expresión y regresaría esta misma representada en cuádruplos.

$A + B * C \rightarrow [* B C t1], [+ A t1 t2]$

```
class QuadrupleGenerator:
    def __init__(self):
        self.temp_count = 0

    def new_temp(self):
        self.temp_count += 1
        return f't{self.temp_count}'

    def generate(self, expr):
        expr = expr.replace(" ", "")
        quadruples = []
        self.parse_assignment(expr, quadruples)
        return quadruples

    def parse_assignment(self, expr, quadruples):
        if '=' in expr:
            var, expr = expr.split('=')
            result = self.parse_expression(expr, quadruples)
            quadruples.append(['=', result, '', var])
        else:
            return self.parse_expression(expr, quadruples)

    def parse_expression(self, expr, quadruples):
        if '(' in expr:
            return self.parse_parentheses(expr, quadruples)

        for operators in ['><', '+-', '* /']:
            result = self.split_at_operator(expr, operators)
            if result:
                left, op, right = result
                left_result = self.parse_expression(left, quadruples)
                right_result = self.parse_expression(right, quadruples)
                temp_var = self.new_temp()
                quadruples.append([op, left_result, right_result, temp_var])
                return temp_var
        return expr

    def parse_parentheses(self, expr, quadruples):
        stack = []
        start = None
        for i, char in enumerate(expr):
            if char == '(':
                stack.append(i)
                if start is None:
                    start = i
            elif char == ')':
                start = stack.pop()
                if not stack:
                    sub_expr = expr[start + 1:i]
                    temp_var = self.parse_expression(sub_expr, quadruples)
                    new_expr = expr[:start] + temp_var + expr[i + 1:]
                    return self.parse_expression(new_expr, quadruples)
        return self.parse_expression(expr, quadruples)
```

La siguiente función se implementó como una extensión de la clase Variables. Esta recibe una lista con todas las expresiones que encuentra el parser en formato de lista de strings. Posteriormente si encuentra alguna expresión, manda el string para generar los cuádruplos a la clase QuadruplesGenerator con el método generate(). Se utiliza una pila (stack) para llevar registro de estatutos que requieren aún una dirección de memoria, ya sea IF, ELSE o DO WHILE.

```
def generate_quadruples(self):
    generator = QuadrupleGenerator()
    count = 0
    stack = []
    while self.expresiones:
        exp = self.expresiones.popleft()
        print(exp)
        if exp == 'IF':
            self.cuadрупlos.append(['gotoF', self.cuadрупlos[count-1][-1], '', ''])
            stack.append(count)
            count += 1
        elif exp == 'ELSE':
            self.cuadрупlos.append(['goto', '', '', ''])
            self.cuadрупlos[stack.pop()][3] = str(count + 2)
            stack.append(count)
            count += 1
        elif exp == 'ENDIF':
            count += 1
            self.cuadрупlos[stack.pop()][3] = str(count)
        elif exp == 'DO':
            stack.append(count)
        elif exp == 'WHILE':
            print(count)
            self.print_cuadрупlos()
            self.cuadрупlos.append(['gotoV', self.cuadрупlos[count-2][-1], '', str(stack.pop())])
            count += 1
        else:
            q = generator.generate(exp)
            self.cuadрупlos.extend(q)
            count += len(q)
    self.print_cuadрупlos()
```

Pruebas Realizadas

En las siguientes pruebas se muestra el código y el resultado de este expresado en cuádruplos.

Prueba #1

```
program HelloWorld;  
  
var A, B, C, D, x : int;  
  
main  
{  
    if(A+B > C*D) {  
        A = B + D;  
    };  
    B = A * C;  
    x = 0;  
    do {  
        x = x + 1;  
        A = 2;  
    } while (x < 6);  
}  
end
```

```
1. ['+', 'A', 'B', 't1']  
2. ['*', 'C', 'D', 't2']  
3. ['>', 't1', 't2', 't3']  
4. ['gotoF', 't3', '', '7']  
5. ['+', 'B', 'D', 't4']  
6. ['=', 't4', '', 'A']  
7. ['*', 'A', 'C', 't5']  
8. ['=', 't5', '', 'B']  
9. ['=', '0', '', 'x']  
10. ['+', 'x', '1', 't6']  
11. ['=', 't6', '', 'x']  
12. ['=', '2', '', 'A']  
13. ['<', 'x', '6', 't7']  
1. ['+', 'A', 'B', 't1']  
2. ['*', 'C', 'D', 't2']  
3. ['>', 't1', 't2', 't3']  
4. ['gotoF', 't3', '', '7']  
5. ['+', 'B', 'D', 't4']  
6. ['=', 't4', '', 'A']  
7. ['*', 'A', 'C', 't5']  
8. ['=', 't5', '', 'B']  
9. ['=', '0', '', 'x']  
10. ['+', 'x', '1', 't6']  
11. ['=', 't6', '', 'x']  
12. ['=', '2', '', 'A']  
13. ['<', 'x', '6', 't7']  
14. ['gotoV', 't7', '', '10']
```

Prueba #2

```
program HelloWorld;  
  
var A, B, C, D : int;  
  
main  
{  
    if(A+B > C*D) {  
        A = B + D;  
    };  
    B = A * C;  
}  
end
```

```
1. ['+', 'A', 'B', 't1']  
2. ['*', 'C', 'D', 't2']  
3. ['>', 't1', 't2', 't3']  
4. ['gotoF', 't3', '', '7']  
5. ['+', 'B', 'D', 't4']  
6. ['=', 't4', '', 'A']  
7. ['*', 'A', 'C', 't5']  
8. ['=', 't5', '', 'B']
```

Prueba #3

```
program HelloWorld;

var A, B, C, D : int;

main
{
    if(A+B > C*D) {
        A = B + D;
    } else {
        A = D - C;
    };
    B = A * C + D;
}
end|
```

1. ['+', 'A', 'B', 't1']
2. ['*', 'C', 'D', 't2']
3. ['>', 't1', 't2', 't3']
4. ['gotoF', 't3', '', '8']
5. ['+', 'B', 'D', 't4']
6. ['=', 't4', '', 'A']
7. ['goto', '', '', '10']
8. ['-', 'D', 'C', 't5']
9. ['=', 't5', '', 'A']
10. ['*', 'A', 'C', 't6']
11. ['+', 't6', 'D', 't7']
12. ['=', 't7', '', 'B']

Entrega #4: Ejecución - Máquina Virtual

Memoria

Para la representación de la memoria en la máquina virtual se utilizó un diccionario de Python con la dirección de memoria como la llave y el valor de la variable. Se asignaron rangos de direcciones dependiendo del tipo de variable.

- 1000 - 1999: Enteros globales
- 2000 - 2999: Enteros flotantes
- 3000 - 3999: Variables temporales

Ejemplo del mapa:

```
dict(
    1001: 11
    2001: 2.3
    3001: 11.0
    3002: False
    3003: True
    3004: False
    3005: False
)
```

Estas variables posteriormente fueron reemplazadas por las direcciones en los cuádruplos. De esta manera se accesan las variables en la ejecución directamente por la dirección asignada en el diccionario.

```
1. ['=', '0', '', 1000]
2. ['=', '2.3', '', 2000]
3. ['+', 1000, '1', 3000]
4. ['=', 3000, '', 1000]
5. ['<', 1000, '6', 3001]
6. ['gotoF', 3001, '', '12']
7. ['CTE_STRING', '', '', '"Less than 5:"']
8. ['PRINT', '', '', '"Less than 5:"']
9. ['VAR_PRINT', '', '', 1000]
10. ['PRINT', '', '', 1000]
11. ['goto', '', '', '22']
12. ['>', 1000, '9', 3002]
13. ['gotoF', 3002, '', '18']
14. ['<', 1000, '11', 3003]
15. ['gotoF', 3003, '', '18']
16. ['CTE_STRING', '', '', '"EXACTLY 10:"']
17. ['PRINT', '', '', '"EXACTLY 10:"']
18. ['CTE_STRING', '', '', '"More than 5:"']
19. ['PRINT', '', '', '"More than 5:"']
20. ['VAR_PRINT', '', '', 1000]
21. ['PRINT', '', '', 1000]
22. ['<', 1000, '11', 3004]
23. ['gotoV', 3004, '', '3']
```

CPU

Representando el CPU, se elaboró un programa con varios estatutos condicionales. Cada uno de estos representa una operación que realiza el programa. Se ejecuta estatuto por estatuto utilizando un loop, repitiendo en caso de algún DO WHILE.

```
while i < len(self.cuadрупlos):
    op, v1, v2, r = self.cuadрупlos[i]
    if op == '=':
        if type(v1) == int:
            self.memoria[r] = self.memoria[v1]
        else:
            self.memoria[r] = float(v1)
        i += 1
        self.fix_type(r)
    elif op == '+':
        if type(v1) == int and type(v2) == str:
            self.memoria[r] = self.memoria[v1] + float(v2)
        elif type(v1) == str and type(v2) == int:
            self.memoria[r] = float(v1) + self.memoria[v2]
        elif type(v1) == int and type(v2) == int:
            self.memoria[r] = self.memoria[v1] + self.memoria[v2]
        elif type(v1) == str and type(v2) == str:
            self.memoria[r] = float(v1) + float(v2)
        i += 1
    elif op == '-':
        if type(v1) == int and type(v2) == str:
            self.memoria[r] = self.memoria[v1] - float(v2)
        elif type(v1) == str and type(v2) == int:
            self.memoria[r] = float(v1) - self.memoria[v2]
        elif type(v1) == int and type(v2) == int:
            self.memoria[r] = self.memoria[v1] - self.memoria[v2]
        elif type(v1) == str and type(v2) == str:
            self.memoria[r] = float(v1) - float(v2)
        i += 1
    elif op == '*':
        if type(v1) == int and type(v2) == str:
            self.memoria[r] = self.memoria[v1] * float(v2)
        elif type(v1) == str and type(v2) == int:
            self.memoria[r] = float(v1) * self.memoria[v2]
        elif type(v1) == int and type(v2) == int:
            self.memoria[r] = self.memoria[v1] * self.memoria[v2]
        elif type(v1) == str and type(v2) == str:
            self.memoria[r] = float(v1) * float(v2)
        i += 1
    elif op == '/':
        if type(v1) == int and type(v2) == str:
            self.memoria[r] = self.memoria[v1] / float(v2)
        elif type(v1) == str and type(v2) == int:
            self.memoria[r] = float(v1) / self.memoria[v2]
        elif type(v1) == int and type(v2) == int:
            self.memoria[r] = self.memoria[v1] / self.memoria[v2]
        elif type(v1) == str and type(v2) == str:
            self.memoria[r] = float(v1) / float(v2)
        i += 1
```

Interfaz

Se desarrolló una app en Flask que ejecuta un servidor local, sirviendo una página en HTML con la interfaz gráfica donde se pueden ejecutar programas de LittleDuck.

Para la ejecución del programa se tiene que instalar Flask y PLY.

- pip install flask
- pip install ply

Ejecutar en el directorio “python3 app.py” para correr la app y abrir navegador en localhost:5000.

Screenshots de aplicación

The image displays two screenshots of the LittleDuck web application interface. Each screenshot consists of a code editor on the left and a console output area on the right.

Top Screenshot:

- Code Editor:** Contains a program named 'HelloWorld'. It declares variables `x` (int) and `y` (float). In the `main` block, it sets `x = 0` and enters a `do` loop. Inside the loop, `x` is incremented by 1. An `if` statement checks `x < 6`; if true, it prints 'Less than 5:'. If false, it checks `x > 9`; if true, it prints 'EXACTLY 10'. The loop continues until `x < 11`, at which point it prints 'More than 5:' and exits the loop.
- Run Code Button:** A button labeled 'Run Code' is located above the console.
- Console Output:** Displays the results of the program execution: 'Less than 5:' followed by numbers 1 through 5, 'More than 5:' followed by numbers 6 through 9, 'EXACTLY 10' at line 10, and 'More than 5:' followed by numbers 10 and 11.

Bottom Screenshot:

- Code Editor:** Contains a program named 'HelloWorld'. It declares variables `x` (int) and `y` (float). In the `main` block, it sets `x = 4` and enters an `if` statement checking `x != 5`. If true, it prints 'NOT 5'. The program ends.
- Run Code Button:** A button labeled 'Run Code' is located above the console.
- Console Output:** Displays the result of the program execution: 'NOT 5'.

```
SI
2.0
Hello World!
3.0
Hello World!
4.0
Hello World!
5.0
Hello World!
6.0
Hello World!
7.0
Hello World!
6
```

[illegible]

littleduck_demo.mov