



Tecnológico
de Monterrey

Patito

Ana Jimena Gallegos Rongel

A01252605

Maestra Elda Guadalupe Quiroga

Oct 12, 2025

Introducción

El siguiente documento describe el proceso de la construcción del analizador léxico y sintáctico y semántico del mini-lenguaje **Patito**, utilizando la herramienta PLY (Python Lex – Yacc). Este trabajo forma parte del proceso de crear un compilador completo, empezando por el análisis y generación automática de los componentes principales scanner y parser.

Link github

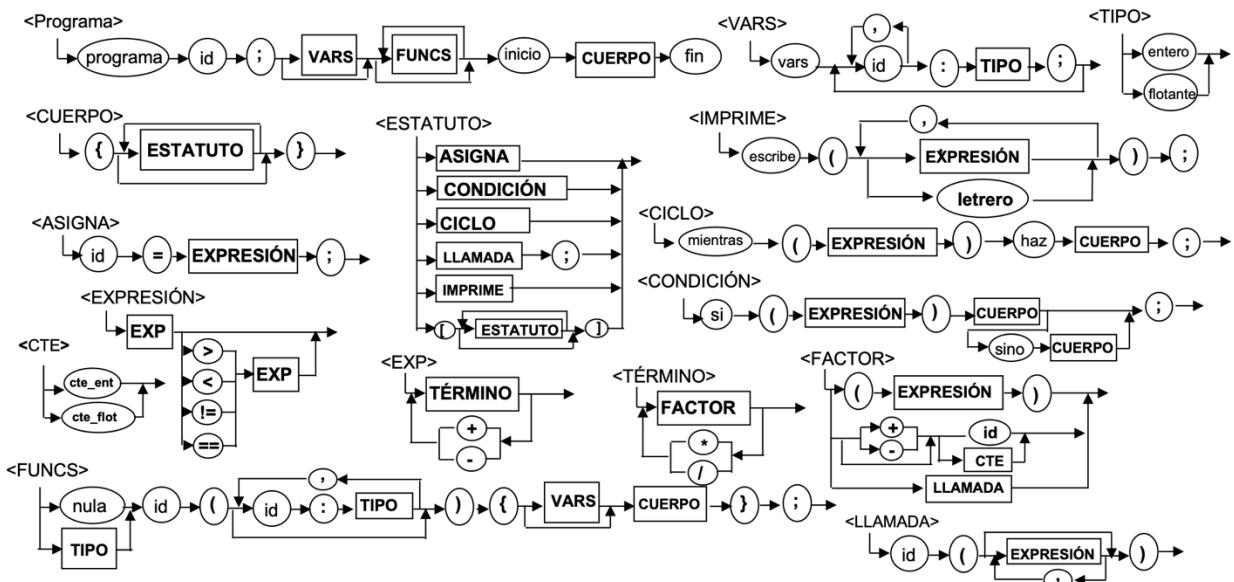
https://github.com/ajimenagallegosr/A01252605_Aplicaciones_Avanzadas/tree/main/Entrega1_A01252605

TC3002B: Desarrollo de aplicaciones avanzadas de Ciencias Computacionales

Módulo: Compiladores

Mini Proyecto INDIVIDUAL : Patito

Octubre 2025



Expresiones regulares

Las expresiones regulares son cadenas de caracteres basadas en reglas sintácticas que permiten describir secuencias de caracteres, en el contexto de Baby_Duck las que utilizaremos son las siguientes:

- Identificadores (ID):** Estas pueden representar nombres de variables, funciones, etc. Deben empezar con una letra o _, y pueden contener letras, números y guion bajo.

`[a-zA-Z_][a-zA-Z0-9_]*`

2. **Números Enteros (INT):** Representan los valores de los números enteros. Para que se considere número entero debe tener por lo menos un número entre 0 y 9 y no puede contener cualquier otro carácter.
`[0-9]+`
3. **Números Flotantes (FLOAT):** Representan los valores de los números flotantes. Para que se considere un número flotante debe tener por lo menos un número entre 0 y 9, después contener un punto, y luego por lo menos un número entre 0 y 9, no puede contener ningún otro carácter.
`[0-9]+.[0-9]+`
4. **Cadenas de texto (cte.STRING):** Representan las cadenas de texto que se encuentran entre “”. Para que se considere un *string* debe empezar con “ y terminar con ”, lo que este adentro no importe siempre y cuando no sean comillas.
`\"*\"`
5. **Operadores y comparadores:** Representan aquellos símbolos utilizados para operaciones aritméticas y para comparaciones.
 - Más: `l+`
 - Menos: `-`
 - Multiplicación: `*`
 - División: `\V`
 - Igual: `=`
 - Mayor: `<`
 - Menor: `>`
 - Diferente que: `!=`
6. **Delimitadores y signos de puntuación:** Representan aquellos símbolos utilizados para marcar el fin de una asignación o para marcar el inicio de una asignación:
 - Paréntesis izquierdo: `\(`
 - Paréntesis derecho: `\)`
 - Llave izquierda: `{`
 - Llave derecha: `}`
 - Corchete izquierdo: `\[`
 - Corchete derecho: `\]`
 - Dos puntos: `:`
 - Punto y colon: `;`
 - Coma: `,`

7. **Espacios en blanco y comentarios:** Representan todo lo que debe ser ignorado por el analizador léxico. Los espacios en blanco, tabulaciones, saltos de línea y comentarios de una línea.

- Espacios en blanco, tabulaciones, saltos de línea: $[\t\r\n]+$
- Comentarios (de una linea): $/*$

Listado de tokens

Reservados

- program
- var
- main
- end
- void
- while
- do
- if
- else
- print
- int
- float

No reservados

- id (Expresión regular)
- cte_int (Enteros, expresión regular)
- cte_float (Flotantes, expresión regular)
- Cte_String (Expresión regular)
- Delimitadores y signos (Expresión regular)
 - =
 - +
 - -
 - *
 - /
 - >
 - <
 - !=

-)
- (
-]
- [
- {
- }
- :
- ;
- ,

Reglas gramaticales

Nota: las reglas entre comillas sencillas (") son solamente reglas para acciones semánticas.

<Programa>

Programa → program ‘create_dirfunc’ id ‘create_id’ ; DECLARACIONES FUNCIONES main BODY end ‘clean_program’

DECLARACIONES → VARS

DECLARACIONES → ϵ

FUNCIONES → ϵ

FUNCIONES → FUNCS FUNCIONES

<VARS>

VARS → var DECLARACION_VAR LISTA_DECLARACIONES

DECLARACION_VAR → LISTA_IDENTIFICADORES : TYPE ;

LISTA_IDENTIFICADORES → id LISTA_IDENTIFICADORES_PRIMA

LISTA_IDENTIFICADORES_PRIMA → , id LISTA_IDENTIFICADORES_PRIMA

LISTA_IDENTIFICADORES_PRIMA → ϵ

LISTA_DECLARACIONES → DECLARACION_VAR LISTA_DECLARACIONES

LISTA_DECLARACIONES → ϵ

<TYPE>

TYPE → INT_TYPE

TYPE → FLOAT_TYPE

<BODY>

BODY → { LISTA_STATEMENTS }

LISTA_STATEMENTS → STATEMENT LISTA_STATEMENTS

LISTA_STATEMENTS → ϵ **<STATEMENT>**

STATEMENT → Print

STATEMENT → CONDITION

STATEMENT → CYCLE

STATEMENT → id OPCION_ID

STATEMENT → { LISTA_STATEMENTS }

OPCION_ID → F_CALL

OPCION_ID → ASSIGN

<PRINT>

PRINT → print (ELEMENTO_IMPRESION LISTA_ELEMENTOS);

ELEMENTO_IMPRESION → EXPRESIÓN

ELEMENTO_IMPRESION → cte.string

LISTA_ELEMENTOS → , ELEMENTO_IMPRESION LISTA_ELEMENTOS

LISTA_ELEMENTOS → ϵ

<ASSIGN>

ASSIGN → = EXPRESIÓN ;

<CYCLE>

CYCLE → while (EXPRESIÓN) do BODY ;

<CONDITION>

CONDITION → if (EXPRESIÓN) BODY PART_ELSE ;

PART_ELSE → else BODY

PART_ELSE → ε

<EXPRESIÓN>

EXPRESIÓN → EXP COMPARACION

COMPARACION → ε

COMPARACION → > EXP

COMPARACION → < EXP

COMPARACION → != EXP

COMPARACION → == EXP

<EXP>

EXP → TÉRMINO SUMA_RESTA

SUMA_RESTA → + TÉRMINO SUMA_RESTA

SUMA_RESTA → - TÉRMINO SUMA_RESTA

SUMA_RESTA → ε

<TÉRMINO>

TÉRMINO → FACTOR MULT_DIV

MULT_DIV → * FACTOR MULT_DIV

MULT_DIV → / FACTOR MULT_DIV

MULT_DIV → ϵ

<FACTOR>

FACTOR → AGRUPACION

FACTOR → SIGNO_UNARIO

FACTOR → CTE

FACTOR → ID ID_OPCION

ID_OPCION → F_CALL

ID_OPCION → ϵ

AGRUPACION → (EXPRESIÓN)

SIGNO_UNARIO → + VALOR

SIGNO_UNARIO → - VALOR

VALOR → id

VALOR → CTE

<FUNCS>

FUNCS → ‘prepare_new_func’ FUNCS_TYPE ‘add_current_type’ id ‘add_function’ (‘start_func_vars’ PARAMETROS){ BLOQUE_FUNCION }‘end_func’;

FUNCS_TYPE → void

FUNCS_TYPE → TYPE

PARAMETROS → PARAMETRO LISTA_PARAMETROS

PARAMETROS → ϵ

PARAMETRO → id : TYPE

LISTA_PARAMETROS → , PARAMETRO LISTA_PARAMETROS

LISTA_PARAMETROS → ε

BLOQUE_FUNCION → VARS BODY

BLOQUE_FUNCION → BODY

<F_CALL>

F_CALL → (ARGUMENTOS)

ARGUMENTOS → EXPRESIÓN LISTA_ARGUMENTOS

ARGUMENTOS → ε

LISTA_ARGUMENTOS → , EXPRESIÓN LISTA_ARGUMENTOS

LISTA_ARGUMENTOS → ε

Revisión de herramientas de generación de compiladores

Antes de seleccionar la herramienta, se investigaron varias alternativas de generación automática de analizadores léxicos y sintácticos. Algunas de las herramientas no elegidas fueron Lex & Yacc, CUP & JFLEX, y COCO/R.

Lex & Yac

- Son herramientas clásicas en la construcción de compiladores.
- Su enfoque en C y Unix/Linux les da alto rendimiento pero las hace menos accesible para entornos modernos.
- Requiere de dos archivos (.l y .y) y un proceso de compilación adicional en C.
- Tiene una muy buena base teórica, pero su sintaxis es rígida y menos intuitiva.

CUP & JFLEX

- Son alternativas modernas desarrolladas en Java.
- Permiten una integración fluida y generan código estructurado orientado a objetos.

- Dependen completamente del ecosistema Java, lo que implica un entorno más complejo.

COCO/R

- Es una herramienta muy completa y versátil, capaz de generar automáticamente el scanner y el parser desde un solo archivo .gtg
- Soporta multiples lenguajes
- La documentación es más técnica, con poco ejemplos actualizados.

Finalmente, se eligió PLY como la herramienta para el desarrollo del compilador Patito. Esta biblioteca implementa el mismo principio de Lex y Yacc, pero completamente en Python, permitiendo escribir reglas gramaticales como funciones dentro del mismo código fuente. Fue seleccionada porque integra de manera sencilla el análisis léxico y sintáctico, manteniendo la estructura clásica de estas herramientas, pero con la flexibilidad y legibilidad que ofrece Python.

Razones principales por el cual se eligio:

- Integración total con Python (no se necesitan generar archivos externos)
- Sintaxis simple y legible
- Facilidad para depurar y extender
- Documentación clara y comunidad activa
- Compatibilidad multiplataforma (se puede utilizar en mac, Linux, Windows usando solamente Python y el paquete ply instalado)

Reglas de construcción de patito

Lexer

El scanner fue implementado en el archivo LexerPatito.py.

El formato para las palabras reservadas, es todo entre {} llaves, poniendo la palabra que el programa reconoceria entre " (comillas simples) seguido por : (dos puntos) y como se guardaria el token en mayusculas.

Palabras reservadas:

```
# palabras reservadas
reserved = {
    'program' : 'PROGRAM',
    'var' : 'VAR',
    'main' : 'MAIN',
    'end' : 'END',
    'void' : 'VOID',
    'while' : 'WHILE',
```

```
'do' : 'DO',
'if' : 'IF',
'else' : 'ELSE',
'print' : 'PRINT',
'int': 'INT_TYPE',
'float': 'FLOAT_TYPE'
}
```

Después de definir las palabras reservadas se listan todos los token que el compilador puede reconocer. Los nombre se escriben en mayúsculas.

Tokens y expresiones regulares:

```
tokens = [
    'ID',
    'CTE_INT',
    'CTE_FLOAT',
    'CTE_STRING',
    'EQUALS',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'GREATER',
    'LESS',
    'DIFFERENT',
    'RPARENTESIS',
    'LPARENTESIS',
    'RBRACKETS',
    'LBRACKETS',
    'RBRACES',
    'LBRACES',
    'COLON',
    'SEMICOLON',
    'COMMA'
] + list(reserved.values())
```

Cada token se define mediante una expression regular. Cuando un token no necesita de un procesamiento extra, simplemente se define con empezando con “t_”.

```
#Regex for simple tokens

t_EQUALS = r'\='
t_PLUS = r'\+'
t_MINUS = r'-'
```

```
t_TIMES = r'\*' 
```

Sin embargo, cuando se necesita una acción adicional, como distinguir entre una palabra reservada y un identificador normal, se define mediante una función que también debe comenzar con “t_” y contiene la expresión regular dentro. Por ejemplo, si el lexer encuentra “if”, “while”, “print”, etc. Los reconoce como palabras reservadas, y si no los clasifica como id.

```
# Regex rule with some action code

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value, 'ID')
    return t

def t_CTE_FLOAT(t):
    r'[0-9]+\.[0-9]+'
    t.value = float(t.value)
    return t
```

Parser

Una vez que el scanner esta listo, se debe pasar con el parser. Este se encarga de interpretar los tokens y validar que sigan las reglas gramaticales del lenguaje. En mi caso esta implementado en ParserPatito.py. La manera de definir las reglas gramaticales es muy sencilla.

Cada regla se implementa como una función en Python, donde el nombre comienza con “p_” seguido del nombre de la regla gramatical.

Dentro de cada función, se escribe la regla en una cadena entre comillas simples. Con el siguiente formato:

‘nombre_regla : componente1 regla2 componente3’

Si una regla tiene diferentes varaciones, se pueden definir funciones adicionales con un extra despues del nombre gramatical de la regla.

Finalmente, despues de la regla se pueden agregar acciones opcionales como print() para pruebas o pass si no se requiere acción en ese punto.

Ejemplos:

```
def p_program(p):
    'program : PROGRAM ID SEMICOLON declaraciones funciones MAIN body END'
```

```

print("Programa válido")
def p_declaraciones_vars(p):
    'declaraciones : vars'
    pass

def p_declaraciones_vacias(p):
    'declaraciones : '
    pass
def p_funcs(p):
    'funcs : VOID ID LPARENTESIS parametros RPARENTESIS LBRACKETS
bloque_funcion RBRACKETS SEMICOLON'
    print(f"Nombre de función detectada: {p[2]}")
    pass

def p_parametros_recursivo(p):
    'parametros : parametro lista_parametros'
    pass

def p_parametros_vacio(p):
    'parametros : '
    pass

```

Principales Test-cases

Para mis casos de prueba define varios para cada tipo de statement del lenguaje, asegurandome de cumplir todas las reglas. Adicionalmente, puse algunos donde habia errores, para buscar que el parser diera “error”. Me asegure que hubiera pruebas para: programa, type, asignación, print, if, else, while, func, condition, error, y sus varaciones (multiples declaraciones, etc).

Ejemplos:

```

# Programa
(" 1. Programa mínimo",
"""

program test;
main {
}
end
""),

# Variables
("2. Declaración simple de variables",
"""

```

```

program var_simples;
var x : int;
main {
}
end
""),

("Declaración múltiple de variables",
"""

program var_multiples;
var x, y, z : float;
main {
}
end
"""),

```

Adicionalmente, antes de los casos de prueba del parser, hice casos de prueba para el Lexer para asegurarme que leyera los tokens correctamente. Incluyendo todas las palabras reservadas y tokens que el Lexer pudiera identificar, y en caso de error hacer las correcciones necesarias.

Semantica de Variables, estructuras seleccionadas

Para esta parte del proyecto habia dos estructuras que se deben implementar el directorio de funciones y la tabla de variables. El directorio de funciones debe contener las funciones que tiene nuestro codigo (incluyendo el programa completo) y debe estar “linkeado” a la tabla de variables. Por su parte la tabla de variables tiene que tener las variables declaradas y su tipo.

Para el Directorio de Funciones se opto por un estrutura “Diccionario” que contiene todas funciones declaradas (program, void, int, float).

```

directory = {
    'func_name' : {
        'type': 'int' | 'float' | 'void' | 'program',
        'vars': VarTable()
    }
}

```

Se opto por usar un diccionario porque se hacen las busquedas con O(1), evita duplicados automaticamente, facil de extender a futuro, ademas, es una estructura que ya conozco y con la que me siento comoda trabajando.

De la misma manera, para la tabla de variables se opto por una estructura “Diccionario”, esta debe tener el nombre de las variables y su tipo. Recordemos que se debe hacer una “nueva” para cada función del programa. Las razones para implementar un diccionario son las mismas que en el directorio de funciones.

```
table = {
    'var_name': { 'type': 'int' | 'float' }
}
```

Operaciones Principales

- 1) Agregar función: Se llama con el nombre de la función y el tipo. Verifica duplicados, registra tipo y nombre, y crea su VarTable.

```
def add_function(self, name, type):
    if name in self.directory:
        raise Exception(f"ERROR: La función '{name}' ya está declarada.")
    self.directory[name] = {
        'type': type,
        'vars': VarTable()
    }
```

- 2) Agregar variable: Se llama con el nombre de la función, el nombre de la variable y el tipo. Además, debemos recordar que nuestro diccionario de funciones y tabla de variables están linkeados. Esta operación se encarga de que no esté repetida y de insertarlo en la VarTable correspondiente de la función actual.

Diccionario de funciones:

```
def add_var(self, func_name, var_name, var_type):
    self.directory[func_name]['vars'].add(var_name, var_type)
```

Tabla de Variables:

```
def add(self, name, type):
    if name in self.table:
        raise Exception(f"ERROR: Variable '{name}' ya está declarada.")
    self.table[name] = { 'type': type }
```

- 3) Checar tipo de variable: Esto será útil más adelante cuando se enlace el cubo semántico, se utilizará para validar expresiones y validar asignaciones.

Diccionario de funciones:

```
def get_var_type(self, func_name, var_name):
```

```
    return self.directory[func_name]['vars'].get_type(var_name)
```

Tabla de Variables:

```
def get_type(self, name):
    if name not in self.table:
        raise Exception(f"ERROR: Variable '{name}' no declarada.")
    return self.table[name]['type']
```

Cuadruplos, PILAS y Filas

Para implementar los cuadruplos se opto por hacer pilas y filas, con el fin de manejar operadores, operando, tipos, y cuadruplos.

PilaOper - Pila de Operadores

Contiene los operadores aritmeticos, relacionales, lógicos encontrados durante el análisis sintáctico. Su proposito es controlar el orden de evaluación usando precedencias.

Ejemplos: + , - , * , & , < , > , !=, ==

Se usa junto con el cubo semántico para validar operaciones.

Como se usa:

```
semantic.PilaOper.append('+')
```

PilaO - Pila de Operandos

Guarda los identificadores, constantes o resultados temporales que serán utilizados dentro de un operación.

Ejemplos: a, b, 1, 2, etc.

Como se usa:

```
Semantic.PilaO.append(p[1])
```

PilaT - Pila de Tipos

Guarda los tipos asociados a los operandos en PilaO, de forma paralela. Despues trabaja con el cubo semantico, para verificar que si se puedan hacer las operaciones o como guardar el nuevo tipo creado en por el cuadruplo.

Como se usa:

```
var_type = semantic.func_dir.get_var_type(...)  
semantic.PilaT.append(var_type)  
semantic.PilaT.append("int")
```

QuadList – Lista de Cuadruplos

Se usa para guardar los cuadruplos generados. Su sintaxis es: operador, elemento izquierdo, elemento derecho, resultado.

Como se usa:

```
Semantic.genereta_quad(op, l, r, res)
```

Temporalres – new_temp()

Genera los nombres para guardar los resultados intermedios. T0, t1, t2, etc.

Funcion reduce_oper

Esta función es de las mas importantes para la generación de cuadruplos, ya que se encarga de hacer los appends() o pops() y la generación de cuadruplos, cuando entra un operador nuevo, toma en cuenta la importancia del operador y si hay parentesis pendientes. Gracias a esta logica se realizan los cuadruplos de manera correcta, esta función se suele llamar cuando entra un operador en cualquier punto.

Puntos neuralgicos

Ayuda:

Para la generación del proyecto se utilizo la ayuda de chatgpt 5. La verdad se usaron muchas prompts, pero la herramienta llegaba a alucinar mucho. Por lo que opte por irlo implementando yo y pedirle consejos de lo que estaba haciendo, o correcciones cuando algo dejaba de funcionar.

Extra, evidencia de trabajo:

