

25-06-2018



**ESPOL**

# **ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL**

**LENGUAJES DE PROGRAMACIÓN**

**TERCER AVANCE**

**Integrantes: Gabriel del Pino, Alix Ferrín, Juan  
José Flores**

# Introducción

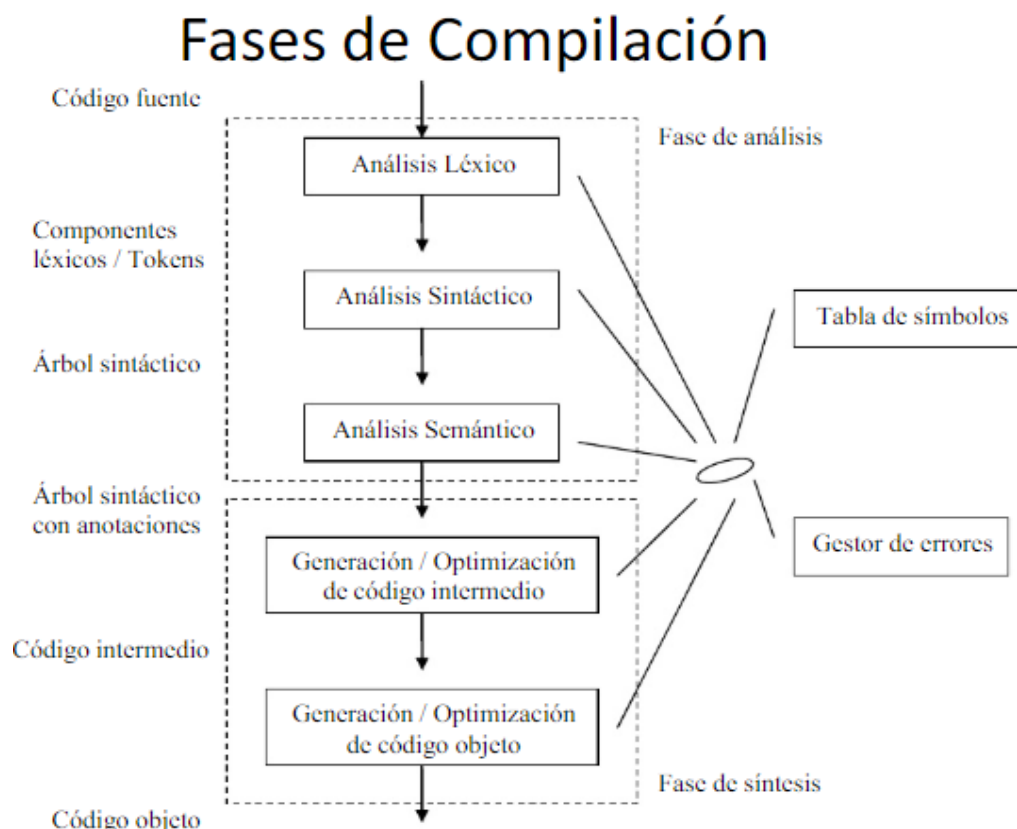
## Compiladores

Los compiladores son programas de computadora que traducen de un lenguaje a otro. Un compilador toma como su entrada un programa escrito en lenguaje fuente, que se lo asocia como un lenguaje de alto nivel y produce un programa equivalente escrito en lenguaje objeto (código de máquina).

### Fases de un compilador

Un compilador se compone internamente de varias etapas que realizan operaciones lógicas.

- **Analizador léxico:** Agrupa las secuencias de caracteres en tokens.
- **Analizador sintáctico:** Determina si la secuencia de componentes léxicos sigue la sintaxis del lenguaje definido y obtiene la estructura jerárquica del programa en forma de árbol.
- **Analizador semántico:** Realiza las comprobaciones necesarias sobre el árbol sintáctico para determinar el correcto significado del programa.
- **Generación y optimización de código intermedio:** Genera un código mejorado, ya no estructurado y más fácil de traducir directamente a código ensamblador o máquina. Cada instrucción tiene un operador y la dirección de dos operandos y un lugar donde guardar el resultado, a eso se lo conoce como código intermedio.
- **Generación código objeto:** Toma como entrada la representación del código intermedio y genera el código objeto



## **Lenguaje Seleccionado**

### **JAVA**

Java es un lenguaje orientado a objetos. La principal característica de Java es que es un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que genera en bytecodes es interpretado por una máquina virtual.

Otra característica de Java es que está preparado para la programación concurrente sin necesidad de utilizar ningún tipo de biblioteca. Finalmente, Java posee un gestor de seguridad con el que poder restringir el acceso a los recursos del sistema. A menudo se argumenta que Java es un lenguaje lento porque debe interpretar los bytecodes a código nativo antes de poder ejecutar un método, pero gracias a la tecnología JIT, este proceso se lleva a cabo una única vez, después el código en código nativo se almacena de tal modo que está disponible para la siguiente vez que se llame.

### **Objetivos**

- Diseño y desarrollo de un compilador diseñado en el lenguaje Python capaz de reconocer expresiones Lambda implementadas en Java 8
- Definir e implementar las fases de compilación.
- Establecer los tokens a ser reconocidos por el compilador durante el análisis sintáctico.

### **Alcance**

El tema a desarrollar será la función lambda y sus distintas aplicaciones en la programación funcional implementada en la versión 8 de Java. Nuestra aplicación será capaz de reconocer determinadas implementaciones establecidas anteriormente, correspondientes a esta función y confirmarle al usuario de la aplicación si la expresión fue escrita correctamente o si tuvo algún error en la parte léxica o en la sintaxis.

### **Metodología**

Para empezar se definen las palabras reservadas y símbolos para la verificación de lambda, por medio de expresiones regulares. Es en esta etapa donde se utiliza el lex de la librería PLY, en el cual se pueden definir la cantidad de tokens que sean necesarios para nuestro compilador.

Por medio del yacc, se realizará el análisis sintáctico, que son las reglas gramaticales definidas según el lenguaje. Dentro de la definición de estas reglas, se pueden establecer patrones de constantes, reglas compuestas o una combinación de ambas.

Una vez que el usuario haya ingresado una función lambda por medio de la interfaz se le dará a conocer si su expresión estuvo correcta o tuvo algún error de léxico o algún error en la sintaxis.

## Herramientas a utilizar

Dentro de Python, una librería muy fuerte para la definición de tokens y parsers que componen un compilador es ply. Mediante esta librería, nos valdremos de herramientas de definición y validación para nuestro compilador, así también del lenguaje BNF para la definición de ciertas reglas para definir patrones de entrada en nuestro sistema.

## Desarrollo

10 ejemplos de expresiones lambda

- Comparación.
- Definición de tipo de dato en una colección o estructura de dato.
- Atajo para manejo de eventos.

Mediante la parte del lexer se pueden definir los tokens necesarios para un posible uso de lambda expression.

Luego, en la parte del yacc se implementa el desempeño de esos tokens previamente definidos.

## LEXER

### Definición de tokens

```
mlexer.py x yacc.py x
1 import ply.lex as lex
2
3 tokens=['IDENTIDAD','MAS','MENOS','DIVIDE','POR','MODULO','POTENCIA','Y','OINCL','OEXCL','NEGADO','IGUAL','IDENTICO','DIFERENTE','MAYOR','MENOR',
4         'IF','ELSE','ELSEIF','WHILE','PARENTH_IZQ','PARENTH_DER','LLAVE_IZQ','LLAVE_DER','DO','FOR','FOREACH','BREAK',
5         'CONTINUE','SWITCH','CASE','DECLARE','RETURN','REQUIRE','INCLUDE','GOTO','VAR','OR','AND',
6         'NUMBER','COMA','STRING','TIPO','LAMBDA','FIN']
7
8 t_ignore = ' \t'
9 t_MAS=r'\+'
10 t_MENOS=r'\-'
```

### Definición de reglas (ejemplos)

```
mlexer.py x yacc.py x
1 import ply.lex as lex
2
3 tokens=['IDENTIDAD','MAS','MENOS','DIV
4         'IF','ELSE','ELSEIF','WHILE
5         'CONTINUE','SWITCH','CASE',
6         'NUMBER','COMA','STRING','I
7
8 t_ignore = ' \t'
9 t_MAS=r'\+'
10 t_MENOS=r'\- '
11 t_POR=r'\*'
12 t_DIVIDE=r'/'
13 t_MODULO=r'%'
14 t_POTENCIA=r'\*\*'
15 t_Y=r'&'
16 t_OINCL=r'\|'
17 t_OEXCL=r'\^'
18 t_NEGADO=r'/'
19 t_IGUAL=r'='
20 t_IDENTICO=r'=='
21 t_DIFERENTE=r'!='
22
```

### Definición de palabras reservadas

```
mlexer.py x yacc.py x
52 t_FIN = r';'
53
54 reserved = {
55     'if': 'IF',
56     'then': 'THEN',
57     'while': 'WHILE',
58     'else if': 'ELSEIF',
59     'include': 'INCLUDE',
60     'return': 'RETURN',
61     'case': 'CASE',
62     'switch': 'SWITCH'
63 }
64
```

## YACC

### Definición de precedencia y reglas de parsing

```
mlexer.py x yacc.py x
6
7 precedence = (
8     ('left', 'MAS', 'MENOS'),
9     ('left', 'POR', 'DIVIDE'),
10 )
11
12
13
14 def p_expr_general(p):
15     '''expr_general : TIPO VAR IGUAL expresion_lambda
16     | VAR IGUAL expresion_lambda
17     | expresion_lambda'''
18
19
20 def p_expresion_lambda(p):
21     '''expresion_lambda : expresion_interna LAMBDA expresion FIN
22     | PARENTH_IZQ expresion_interna PARENTH_DER LAMBDA expresion FIN'''
23
24
25 def p_expresion_interna(p):
26     '''expresion_interna : TIPO VAR COMA expresion_interna
27     | TIPO VAR
28     | VAR'''
29
```

### Definición errores, definición de parser y modelo de ejecución

```
mlexer.py x yacc.py x
66
67 ##PROYECTO LENGUAJES, DEFINICION DE EXPRESIONES LAMBDA JAVA 8##
68
69 def p_empty(p):
70     '''empty : '''
71     p[0] = None
72
73
74 def p_error(p):
75     print(p)
76
77
78 parser = yacc.yacc(debug = False, write_tables = False)
79
80 while True:
81     try:
82         s = input("Ingresa algo;")
83     except EOFError:
84         break
85     parser.parse(s)
86     print("Cool")
```