06-07-2018

ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

LENGUAJES DE PROGRAMACIÓN

PROYECTO PRIMER PARCIAL

AVANCE FINAL

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

Profesor: MSc. Rodrigo Saraguro

Introducción

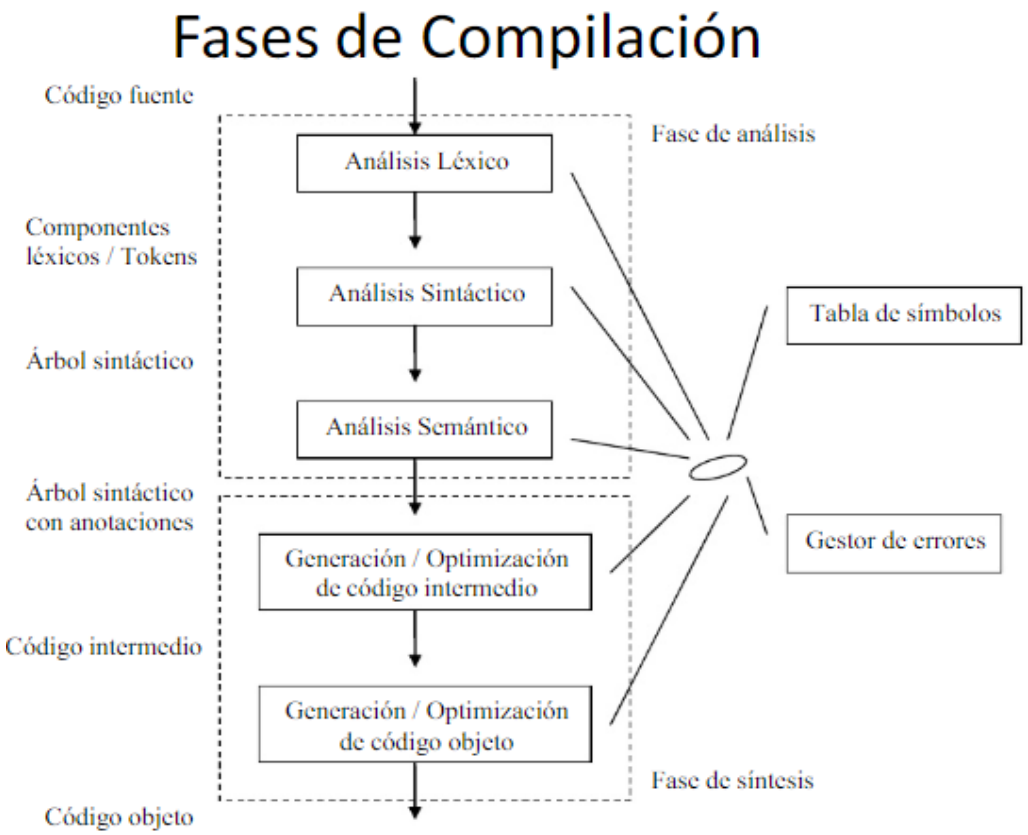
Compiladores

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

Proceso de implementación de 10 ejemplos de expresiones lambda.

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

yacc.py | mlexer.py

```
import ply.lex as lex

tokens=['IDENTIDAD','MAS','MENOS','DIVIDE','POR','MODULO','POTENCIA','Y','OINCL','OEXCL','NEGADO','IGUAL','IDENTICO','DIFERENTE','MAYOR', 'MENOR',
        'PARENTH_IZQ', 'PARENTH_DER', 'LLAVE_IZQ','LLAVE_DER', 'DO', 'POR', 'FOREACH', 'BREAK',
        'CONTINUE', 'REQUIRE', 'GOTO','VAR','OR','AND',
        'NUMBER', 'COMA', 'TIPO', 'LAMBDA','FIN', 'INT', 'STRING', 'DOUBLE', 'FLOAT']

t_ignore = ' \t'
t_MAS=r'\+'
t_MENOS=r'-'
```

Definición de reglas simples (ejemplos)

```
t_ignore = ' \t'
t_MAS=r'\+'
t_MENOS=r'-'
t_POR=r'\*'
t_DIVIDE=r'/'
t_MODULO=r'%'
t_POTENCIA=r'\*\*'
t_Y=r'&'
t_OINCL=r'\|'
t_OEXCL=r'\^'
t_NEGADO=r'/~'
t_IGUAL=r'='
t_IDENTICO=r'=='
t_DIFERENTE=r'!='
t_COMA = r','
t_MAYOR=r'>'
t_MENOR=r'<'
t_PARENTH_IZQ = r'\('
```

Definición de palabras reservadas

```
reserved = {
    'if': 'IF',
    'then':'THEN',
    'while':'WHILE',
    'else if': 'ELSEIF',
    'int': 'INT',
    'String': 'STRING',
    'double': 'DOUBLE',
    'float': 'FLOAT'
}
```

Definición de reglas complejas (ejemplos)

yacc.py | mlexer.py

```
51
52  def t_VAR(p):
53      r'\w+'
54      p.type = reserved.get(p.value, 'VAR')
55      return p
56
57
58  def t_NUMBER(t):
59      r'\d+'
60      t.value = int(t.value)
61      return t
62
63
64  def t_INT(t):
65      r'int'
66      t.value = t.value
67      return t
68
```

Definición de error

yacc.py | mlexer.py

```
73      return t
74
75
76  def t_FLOAT(t):
77      r'float'
78      t.value = t.value
79      return t
80
81
82  def t_DOUBLE(t):
83      r'double'
84      t.value = t.value
85      return t
86
87
88  def t_error(t):
89      print(t)
90
```

YACC

Definición reglas de parsing

yacc.py | mlexer.py

```
1   import mlexer
2   import ply.yacc as yacc
3
4
5   tokens = mlexer.tokens
6
7   ##PROYECTO LENGUAJES, DEFINICION DE EXPRESIONES LAMBDA JAVA 8##
8
9
10  def p_expr_general(p):
11      '''expr_general : type VAR IGUAL expresion_lambda
12      | VAR IGUAL expresion_lambda
13      | expresion_lambda'''
14
15
16  def p_expresion_lamda(p):
17      '''expresion_lambda : expresion_interna LAMBDA expresion FIN
18      | PARENTH_IZQ expresion_interna PARENTH_DER LAMBDA expresion FIN'''
19
```

Definición errores, definición de parser y modelo de ejecución

yacc.py | mlexer.py

```
59
60      def p_empty(p):
61          '''empty : '''
62          p[0] = None
63
64
65      def p_error(p):
66          print(p)
67
68
69      parser = yacc.yacc(debug = False, write_tables = False)
70
71      while True:
72          try:
73              s = input("Ingresa algo;")
74          except EOFError:
75              break
76          parser.parse(s)
77          print("Cool")
78
```