CENTRO DE CIENCIAS BÁSICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

AUTÓMATAS II

7° "A"

PROYECTO: DESARROLLO DE UN INTÉRPRETE

Dr. Francisco Javier Ornelas Zapata

Alumnos:

Almeida Ortega Andrea Melissa

Espinoza Sánchez Joel Alejandro

Flores Fernández Óscar Alonso

Gómez Garza Dariana

González Arenas Fernando Francisco

Orocio García Hiram Efraín

Fecha de Entrega: Aguascalientes, Ags.**, 3** de octubre de 2021

Índice

Introducción

Los compiladores y los intérpretes son programas de gran complejidad. El conocimiento que se tiene acerca de cómo estructurarlos es suficiente, así como también lo es la cantidad de herramientas formales para que la complejidad se reduzca a niveles razonables.

En ciencias de la computación, el intérprete es un programa informático capaz de analizar y ejecutar otros programas. Los intérpretes se diferencian de los compiladores o de los ensambladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes solo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

Usando un intérprete, un solo archivo fuente puede producir resultados iguales incluso en sistemas sumamente diferentes si es compilado a distintos ejecutables específicos a cada sistema.

Los programas interpretados suelen ser más lentos que los compilados debido a la necesidad de traducir el programa mientras se ejecuta, pero a cambio son más flexibles como entornos de programación y depuración (lo que se traduce, por ejemplo, en una mayor facilidad para reemplazar partes enteras del programa o añadir módulos completamente nuevos), y permiten ofrecer al programa interpretado un entorno no dependiente de la máquina donde se ejecuta el intérprete, sino del propio intérprete (lo que se conoce comúnmente como máquina virtual).

Para mejorar el desempeño, algunas implementaciones de algunos lenguajes de programación pueden interpretar o compilar el código fuente original en una forma intermedia más compacta, y después traducir eso al código de máquina.

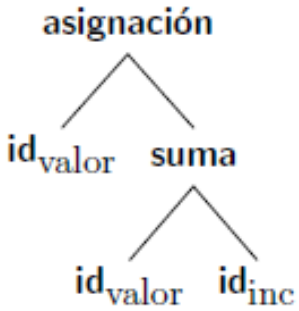
El proceso de traducción entre dos lenguajes se lleva a cabo en dos etapas. La primera etapa se analiza la entrada para averiguar qué es lo que se intenta comunicar. Esto es a lo que se le conoce como análisis. El fruto de esta etapa es una representación de la entrada que permite que la siguiente etapa se desarrolle con facilidad.

La segunda etapa, la síntesis, toma la representación obtenida en el análisis y la transforma en su equivalente, que es en el lenguaje destino cuando se trata de un compilador. Para la interpretación, se utiliza la representación intermedia y así obtener los resultados deseados.

Etapa 1: Análisis

El objetivo de esta etapa es obtener una representación de la entrada que nos permita realizar la síntesis o la interpretación con comodidad. Se usarán los árboles de sintaxis abstracta como representación intermedia. Supóngase el siguiente ejemplo de traducción:

```
valor = valor + inc;
```



El paso de la entrada al árbol de sintaxis abstracta no es trivial. Para facilitarlo, se divide la tarea en varias partes. Supóngase tener que describir un lenguaje de programación. Una manera de hacerlo sería comenzando por describir cuáles son las unidades elementales tales como identificadores, palabras reservadas, operadores, etc. que se encuentran en la entrada. Después se pensaría en como se pueden combinar esas unidades en estructuras mayores tales como expresiones, asignaciones, bucles y demás. Finalmente, una serie de normas que deben cumplirse para que el programa, además de estar "bien escrito", tenga significado. Estas normas se refieren a aspectos tales como que las variables deban declararse o las reglas que se siguen para decidir los tipos de las expresiones.

Cada paso de estos se corresponde con las tres fases en las que frecuentemente se divide el análisis:

- **Análisis léxico:** Divide la entrada en componentes léxicos.
- **Análisis sintáctico:** Encuentra las estructuras presentes en la entrada.
- **Análisis semántico:** Comprueba que se cumplen las restricciones semánticas del lenguaje.

Etapa 2: Síntesis

Terminado el análisis, se debe generar el código objetivo para la máquina. Habitualmente se divide la generación en dos etapas.

- Generación de código intermedio.
- Generación de código objeto.

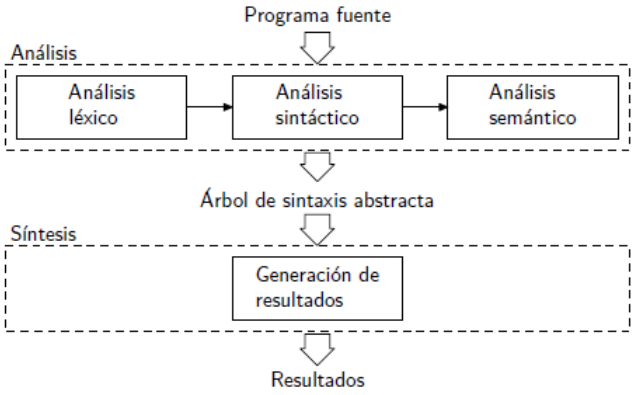
Estos son pasos comunes en los casos de un compilador. Para el caso de un intérprete, el árbol de sintaxis abstracta se divide y se recorre junto con los datos de entrada para obtener los resultados.

Actualmente es habitual encontrar híbridos entre la compilación y la interpretación que consisten en compilar a un lenguaje intermedio para una máquina virtual y después interpretar este lenguaje. Esta aproximación es la que se sigue en Java, Python o .NET como algunos ejemplos.

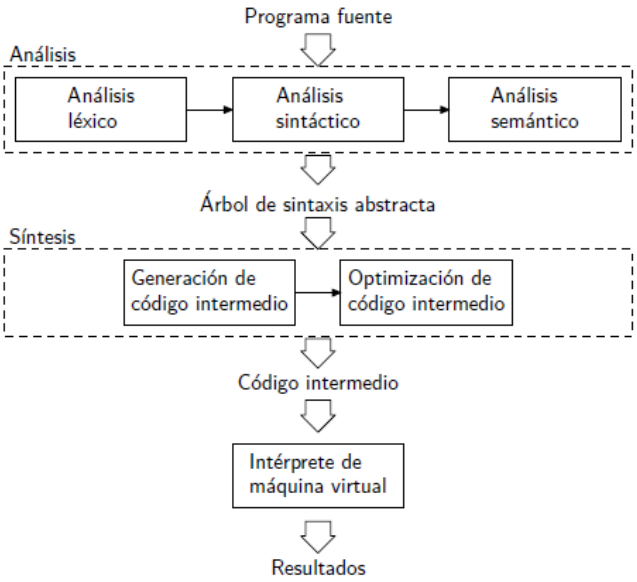
Así también, algunos programas aceptan los archivos fuente guardados en esta representación intermedia como Python, UCSD Pascal y Java.

En la actualidad, uno de los entornos más comunes de uso de los intérpretes es en los navegadores web, debido a la posibilidad que estos tienen de ejecutarse independientemente de la plataforma.

Hay que mencionar que con estos pasos comentados, la estructura de un intérprete es la siguiente:



Así también, si se usara una máquina virtual, se tendría esta estructura:



Bajo estos conceptos y tomando en cuenta algunas especificaciones del propósito de desarrollo del presente documento, se decidió realizar un intérprete, pero no un intérprete en el que se realizaran las mismas actividades de un lenguaje de programación.

El equipo de desarrollo pensó en elaborar un intérprete matemático orientado a esta ciencia a partir de las fuentes bibliográficas. Dado que en la actualidad se están desarrollando muchos avances en lenguajes de programación con orientación a las matemáticas.

A continuación se describe paso a paso la realización del intérprete matemático propuesto por el equipo de desarrollo.

El Intérprete

A continuación se estará describiendo el intérprete matemático en el lenguaje de programación Python que este equipo de programación realizó. Bajo la investigación realizada, se creyó necesario dividir el desarrollo y su descripción en tres etapas que estarán tituladas propiamente a continuación.

El Lexer

Para la realización del leer, fue necesario pensar las posibles entradas del intérprete, propiamente a los tipos de tokens que se insertarían al intérprete, por ejemplo, los números, operadores o paréntesis. Para ello se creó el archivo `tokens.py` donde se escribirían en una variable tipo enum los tipos de tokens que, hasta el momento, tendría nuestro intérprete.

```python
from dataclasses import dataclass
from enum import Enum

class TokenType(Enum):
    NUMBER    = 0
    PLUS      = 1
    MINUS     = 2
    MULTIPLY  = 3
    DIVIDE    = 4
    LPAREN    = 5
    RPAREN    = 6
```

Este tipo de clase sería representado por su propio método de representación de tokens que se encuentra en el mismo archivo:

```python
@dataclass()
class Token:
    type: TokenType
    value: any = None

    def __repr__(self):
        return self.type.name + (f":{self.value}" if self.value != None
        else "")
```

A partir de este conjunto de instrucciones, ahora se comienza con el desarrollo del leer. En él, se leerá el texto introducido. Se realizarán procedimientos de avance

caracter a caracter, creación de tokens según el tipo de cada carácter y la generación de números:

```
class Lexer:
  def __init__(self, text):
    self.text = iter(text)
    self.advance()

  def advance(self):
    try:
      self.current_char = next(self.text)
    except StopIteration:
      self.current_char = None

  def generate_tokens(self):
    while self.current_char != None:
      if self.current_char in WHITESPACE:
        self.advance()
      elif self.current_char == '.' or self.current_char in DIGITS:
        yield self.generate_number()
      elif self.current_char == '+':
        self.advance()
        yield Token(TokenType.PLUS)
      elif self.current_char == '-':
        self.advance()
        yield Token(TokenType.MINUS)
      elif self.current_char == '*':
        self.advance()
        yield Token(TokenType.MULTIPLY)
      elif self.current_char == '/':
        self.advance()
        yield Token(TokenType.DIVIDE)
      elif self.current_char == '(':
        self.advance()
        yield Token(TokenType.LPAREN)
      elif self.current_char == ')':
        self.advance()
        yield Token(TokenType.RPAREN)
      else:
        raise Exception(f"Illegal character '{self.current_char}'")
```

```
def generate_number(self):
  decimal_point_count = 0
  number_str = self.current_char
  self.advance()

  while self.current_char != None and (self.current_char == '.' or self.current_char in
  DIGITS):
    if self.current_char == '.':
      decimal_point_count += 1
      if decimal_point_count > 1:
        break

    number_str += self.current_char
    self.advance()

  if number_str.startswith('.'):
    number_str = '0' + number_str
  if number_str.endswith('.'):
    number_str += '0'

  return Token(TokenType.NUMBER, float(number_str))
```

Donde se separa – sólo por cuestión de denotación – los tokens de los números, pero en términos estrictos de programación, los números también son tokens de tipo numérico, pero en este caso, son tratados de distinta forma.

Al finalizar este apartado, el análisis léxico termina su propósito.

El Parser

Ahora en esta sección, creamos un archivo dentro del proyecto que se llama `nodes.py` para definir nuestros diferentes tipos de nodo. En éste usamos el método `__repr__` este es un método de representación que simplemente devuelve el valor como una cadena. A continuación, tenemos un nodo diferente para todas las diferentes operaciones:

nodes.py ×

```
2   from dataclasses import dataclass
3
4   @dataclass
5   class NumberNode:
6     value: any
7
8     def __repr__(self):
9       return f"{self.value}"
10
11  @dataclass
12  class AddNode:
13    node_a: any
14    node_b: any
15
16    def __repr__(self):
17      return f"({self.node_a}+{self.node_b})"
18
19  @dataclass
20  class SubtractNode:
21    node_a: any
22    node_b: any
23
24    def __repr__(self):
25      return f"({self.node_a}-{self.node_b})"
```

nodes.py ×

```
27  @dataclass
28  class MultiplyNode:
29      node_a: any
30      node_b: any
31
32      def __repr__(self):
33          return f"({self.node_a}*{self.node_b})"
34
35  @dataclass
36  class DivideNode:
37      node_a: any
38      node_b: any
39
40      def __repr__(self):
41          return f"({self.node_a}/{self.node_b})"
42
43  @dataclass
44  class PlusNode:
45      node: any
46
47      def __repr__(self):
48          return f"(+{self.node})"
49
50  @dataclass
51  class MinusNode:
52      node: any
```

Creamos otro archivo dentro de la misma carpeta y lo llamaremos `parser.py`, este analizador realizará el trabajo de construir un árbol para definir las operaciones del programa. Así el analizador construirá un árbol con el orden correcto de operaciones.

El analizador va a transformar nuestros tokens en nodos, por lo que necesitaremos acceso a los tipos de tokens para verificar los tipos de tokens y, por supuesto, también necesitaremos importar nuestros tipos de nodos para que ahora podamos crear nuestra clase de analizador. En el método de inicialización vamos a tomar los tokens que analizaremos:

```
2   from tokens import TokenType
3   from nodes import *
4
5   class Parser:
6     def __init__(self, tokens):
7       self.tokens = iter(tokens)
8       self.advance()
9
10    def raise_error(self):
11      raise Exception("Invalid sintax")
12
13    def advance(self):
14      try:
15        self.current_token = next(self.tokens)
16      except StopIteration:
17        self.current_token = None
18
19    def parse (self):
20      if self.current_token == None:
21        return None
22
23      result = self.expr()
24
25      if self.current_token != None:
26        self.raise_error()
27
28      return result
29
```

En cada rol que hemos descrito anteriormente solo agregará una verificación de seguridad y verificará si el token actual no es ninguno, esto significa que no hay tokens y el usuario no ha ingresado nada y, en este caso, podemos devolver ninguno, así que no vamos a devolver la nota después de la regla de expresión; solo se verifica si el token actual no es igual a ninguno, en cuyo caso esto significa que hay más tokens que deben procesarse y, por alguna razón, no se han procesado.

Ahora comenzamos a definir la regla de expresión, así que crearemos este método y, para las operaciones mayor y menor de dos términos llamaremos el término de punto propio a busca de un término.

Ahora comenzamos a definir la regla de expresión, por lo que crearemos ese método, queremos buscar las operaciones de más y menos de dos términos para poder llamar al término de punto propio a que busque un término. Podemos asignar el resultado solo a una variable de resultado y buscar un cero o más operadores o menos.

```
30    def expr(self):
31      result = self.term()
32
33      while self.current_token != None and self.current_token.type in (TokenType.PLUS,
        TokenType.MINUS):
34        if self.current_token.type == TokenType.PLUS:
35          self.advance
36          result = AddNode(result, self.term())
37        elif self.current_token.type == TokenType.MINUS:
38          self.advance()
39          result = SubtractNode(result, self.term())
40
41      return result
42
43    def term(self):
44      result = self.factor()
45
46      while self.current_token != None and self.current_token.type in (TokenType.MULTIPLY,
        TokenType.DIVIDE):
47        if self.current_token.type == TokenType.MULTIPLY:
48          self.advance
49          result = MultiplyNode(result, self.factor())
50        elif self.current_token.type == TokenType.DIVIDE:
51          self.advance()
52          result = DivideNode(result, self.factor())
53
54      return result
```

```
56      def factor(self):
57          token = self.current_token
58
59          if token.type == token.type.LPAREN:
60              self.advance
61              result = self.expr()
62
63              if self.current_token.type != TokenType.RPAREN:
64                  self.raise_error()
65
66              self.advance()
67              return result
68
69          if token.type == TokenType.NUMBER:
70              self.advance
71              return NumberNode(token.value)
72
73          elif token.type == TokenType.PLUS:
74              self.advance
75              return PlusNode(self.factor())
76
77          elif token.type == TokenType.MINUS:
78              self.advance
79              return MinusNode(self.factor())
80
81          self.raise_error()
```

El Intérprete

Para esta fase se crearon dos archivos más: `interpreter.py` y `values.py`.

En el archivo `values.py` se escribió una clase Number que definiría la representación a modo de cadena de un número:

```python
from dataclasses import dataclass

@dataclass
class Number:
  value: float

  def __repr__(self):
    return f"{self.value}"
```

Este archivo se limita a sólo esta clase para esta representación por falta de tiempo, sin embargo, si se desea expandir este trabajo, en este archivo sería idóneo agregar más clases para más tipos de representaciones, no sólo números. Entre algunas ideas, fracciones, notación científica, números complejos, expresiones algebraicas, entre muchas otras posibilidades de expansión al código.

Ahora, para el archivo `interpreter.py` éste deberá conocer los nodos de previas implementaciones y los tipos de números recién implementados en esta sección que estará produciendo.

El primer paso que se realizará será crear una clase Interpreter que tendrá como método la revisión de tipo de método en cada fase para simular el árbol:

```python
class Interpreter:
  def visit(self, node):
    method_name = f'visit_{type(node).__name__}'
    method = getattr(self, method_name)
    return method(node)
```

Los tipos de nodos se declaran próximamente como métodos para la búsqueda de cada lado del nodo, inferior y superior:

```
def visit_NumberNode(self, node):
  return Number(node.value)

def visit_AddNode(self, node):
  return Number(self.visit(node.node_a).value + self.visit(node.node_b).value)

def visit_SubtractNode(self, node):
  return Number(self.visit(node.node_a).value - self.visit(node.node_b).value)

def visit_MultiplyNode(self, node):
  return Number(self.visit(node.node_a).value * self.visit(node.node_b).value)

def visit_DivideNode(self, node):
  try:
    return Number(self.visit(node.node_a).value / self.visit(node.node_b).value)
  except:
    raise Exception("RuntimeError")

def visit_PlusNode(self, node):
  return self.visit(node.node)

def visit_MinusNode(self, node):
  return Number(-self.visit(node.node).value)
```

Finalmente se edita en el archivo `main.py` el agregado del llamado a un objeto Interpreter. Primeramente se elimina la posibilidad de teclear una línea nula, omitiendo este análisis con la línea:

```
if not tree: continue
```

Lo siguiente será crear el objeto interpreter y recorrer su árbol generado:

```
interpreter = Interpreter()
value = interpreter.visit(tree)
print(value)
```

Pruebas Unitarias

Para el apartado de pruebas unitarias, que es la última sección, se agregará para dar una opción en código de evaluación sobre el proyecto que ya hasta el apartado anterior está explicado en su totalidad.

En primer lugar se debió implementar un evaluador para el Lexer, el cual evaluara casos como un Lexer con datos vacíos, para la entrada de números, operadores, paréntesis y su evaluación general:

```python
class TestLexer(unittest.TestCase):
    def test_empty(self):
        tokens=list(Lexer("").generate_tokens())
        self.assertEqual(tokens, [])


    def test_empty(self):
        tokens=list(Lexer(" \t\n  \t\t\n\n").generate_tokens())
        self.assertEqual(tokens, [])

    def test_numbers(self):
        tokens=list(Lexer("123 123.456 123. .456").generate_tokens())
        self.assertEqual(tokens, [
            Token(TokenType.NUMBER, 123.000),
            Token(TokenType.NUMBER, 123.456),
            Token(TokenType.NUMBER, 123.000),
            Token(TokenType.NUMBER, 000.456),
            Token(TokenType.NUMBER, 000.000),
            ])

    def test_operators(self):
        tokens=list(Lexer("+-*/").generate_tokens())
        self.assertEqual(tokens, [
            Token(TokenType.PLUS),
            Token(TokenType.MINUS),
            Token(TokenType.MULTIPLY),
            Token(TokenType.DIVIDE),
            ])

    def test_parens(self):
        tokens= list(Lexer("()").generate_tokens())
        self.assertEqual(tokens, [
            Token(TokenType.LPAREN),
            Token(TokenType.RPAREN),
            ])
```

A continuación se realiza la evaluación para el Parser la cual realiza evaluaciones cada vez con mayor alcance matemático. En la primera función realiza una evaluación a los números involucrados dentro de la expresión:

```
class TestParser(unittest.TestCase):
    def test_empty(self):
        tokens=[]
        node= Parser(tokens).parse()
        self.assertEqual(node, None)

    def test_numbers(self):
        tokens= [Token(TokenType.NUMBER, 51.2)]
        node= Parser(tokens).parse()
        self.assertEqual(node, NUmberNode(51.20))
```

Posteriormente a toda una expresión matemática:

```
def test_individual_operations(self):
    token= [
        Token(TokenType.NUMBER, 27),
        Token(TokenType.PLUS),
        Token(TokenType.NUMBER, 14),
        ]



    node= Parser(tokens).parse()
    self.assertEqual(node, AddNode(NumberNode(27), NumberNode(14)))

    token= [
        Token(TokenType.NUMBER, 27),
        Token(TokenType.MINUS),
        Token(TokenType.NUMBER, 14),
        ]

    node= Parser(tokens).parse()
    self.assertEqual(node, SubtractNode(NumberNode(27), NumberNode(14)))

    token= [
        Token(TokenType.NUMBER, 27),
        Token(TokenType.MULTIPLY),
        Token(TokenType.NUMBER, 14),
        ]

    node= Parser(tokens).parse()
    self.assertEqual(node, MultiplyNode(NumberNode(27), NumberNode(14)))

    token= [
        Token(TokenType.NUMBER, 27),
        Token(TokenType.DIVIDE),
        Token(TokenType.NUMBER, 14),
        ]

    node= Parser(tokens).parse()
    self.assertEqual(node, DivideNode(NumberNode(27), NumberNode(14)))
```

Finalmente realiza la evaluación a todas las expresiones dentro de la instrucción especificada:

```
def test_full_expression(self):
    tokens=[
        Token(TokenType.NUMBER, 27),
        Token(TokenType.PLUS),
        Token(TokenType.LPAREN),
        Token(TokenType.NUMBER, 43),
        Token(TokenType.DIVIDE),
        Token(TokenType.NUMBER, 36),
        Token(TokenType.MINUS),
        Token(TokenType.NUMBER, 48),
        Token(TokenType.RPAREN),
        Token(TokenType.MULTIPLY),
        Token(TokenType.NUMBER, 51),
        ]
    node= Parser(tokens).parse()
    self.assertEqual(node, AddNode(
        NumberNode(27),
        MultiplyNode(
            SubtractNode(
                DivideNode(
                    NUmberNode(43),
                    NUmberNode(36),
                    ),
                NUmberNode(48),
                ),
            NumberNode(51),
            )
        ))
```

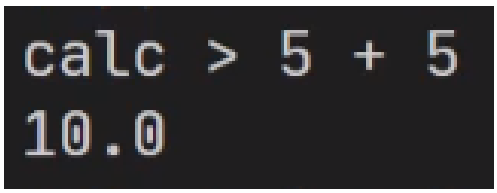
Finalmente se redacta el evaluador para el intérprete, que se encarga de revisar la coherencia entre los elementos de una expresión y entre todas las expresiones de la instrucción:

```
class TestInterpreter(unittest.TestCase):
    def test_number(self):
        Interpreter().visit(NumberNode(51.2))
        self.assertEqual(value, Number(51.2))

    def test_individual_operations(self):
        value= Interpreter().visit(AddNode(NumberNOde(27), NumberNode(14)))
        self.assertEqual(value, Number(41))

        value= Interpreter().visit(SubtractNode(NumberNOde(27), NumberNode(14)))
        self.assertEqual(value, Number(13))

        value= Interpreter().visit(MultoplyNode(NumberNOde(27), NumberNode(14)))
        self.assertEqual(value, Number(378))

        value= Interpreter().visit(DivideNode(NumberNOde(27), NumberNode(14)))
        self.assertAlmostEqual(value.value, 1.92857, 5)

        with self.assertRaises(Exception):
            Interpreter().visit(DivideNode(NumberNode(27), NUmberNode(0)))

    def test_full_expression(self):
        tree= AddNode(
            NumberNode(27),
            MultiplyNode(
                SubtractNode(
                    DivideNode(
                        NUmberNode(43),
                        NUmberNode(36),
                        ),
                    NUmberNode(48),
                    ),
                NumberNode(51),
                )
            )

        result= Interpreter().visit(tree)
        self.assertAlmostEqual(result.value, -2360.08, 2)
```

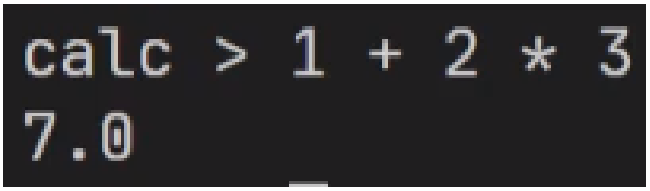
Todos los archivos se encuentran anexos al documento, así como al entregable del presente archivo.

Análisis y Procesamiento de Resultados

El intérprete lee las operaciones aritméticas insertadas. Un ejemplo del funcionamiento correcto básico del intérprete es el siguiente:

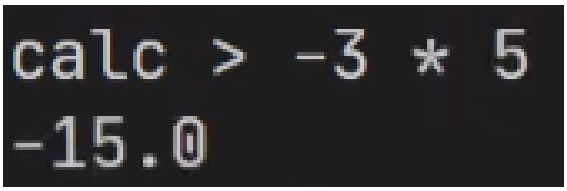


Sin embargo, ejecuciones más complejas pueden llevarse a cabo:

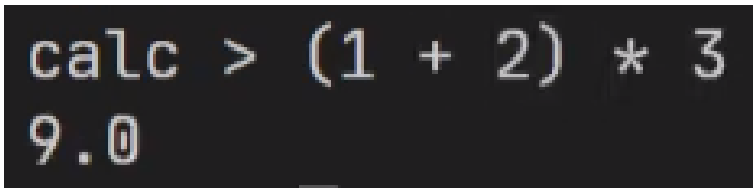


Nótese que la jerarquía de operaciones aritméticas se mantiene por cómo se construyó la lectura del árbol del intérprete.

Otras ejecuciones son las siguientes, como el uso de números negativos:



También se prueba el uso de paréntesis como parte del análisis del intérprete:



Conclusiones

Andrea Melissa Almeida Ortega: Tras realizar esta práctica nos queda más claro lo que es in intérprete ya que no hay mejor manera de conocer algo que desarrollándolo paso a paso, para así comprender mejor su funcionamiento y dejándonos bien establecida la diferencia entre compiladores e intérpretes.

En este trabajo me di cuenta de la gran importanciaque es combinar la teoría con la práctica para llegar al conocimiento deseado.

Joel Alejandro Espinoza Sánchez: A lo largo de estar trabajo cómo se realiza paso a paso un intérprete, en el que, debido a que previamente se realizó un compilador con características muy similares, se optó por desarrollar algo novedoso como sería un intérprete orientado a aplicaciones matemáticas.

El funcionamiento es como el de un intérprete tradicional y fue interesante aplicar conceptos de análisis léxico, sintáctico y semántico a las necesidades de un lector matemático para su procesamiento aritmético.

Óscar Alonso Flores Fernández: A lo largo de la realización de este proyecto pudimos experimetar de primera mano las complicaciones y el trabajo que requiere la realización de un intérprete

Dariana Gómez Garza: En la realización de esta práctica nos queda aún más claro la diferencia entre un intérprete y un compilador, aunque el intérprete se componga de "compiladores" es sencillo de entender gracias a la teoría vista en clase y que fue investigada. Para esta actividad tratamos de dividirnos el trabajo en partes iguales para hacer el código y la documentación y así todos hacer aportes equitativos.

Fernando Francisco González Arenas: Los intérpretes son muy similares a los compiladores, con la diferencia que los intérpretes van traduciendo el código línea por línea, aumentando el tiempo de ejecución pero dando mayor flexibilidad al momento de hacer cambios al programa o de ejecutarlo en diferentes maquinas convirtiéndose en un tipo de máquina virtual.

En este proyecto aprendí la teoría, funcionamiento y composición de los intérpretes más a fondo, conociendo la forma como funcionan interna y externamente, lo cual me servirá en futuros trabajos ya sea académicos o en el campo laboral.

Hiram Efraín Orocio García: a

Fuentes de Consulta

- Anónimo. (2013). *Intérprete (Informática)*. Septiembre 30, 2021, de Wikipedia Sitio web: https://es.wikipedia.org/wiki/Int%C3%A9rprete_(inform%C3%A1tica)
- Anónimo. (2014). *Procesadores de Lenguaje: Estructura de los Compiladores e Intérpretes*. Castellón: Editorial Universitat Jaume I.
- CodePulse. (2020). *Simple Math Interpreter in Python (1/4) - Lexer*. Septiembre 30, 2021, de YouTube Sitio web: https://www.youtube.com/watch?v=88lmIMHhYNs.
- CodePulse. (2020). *Simple Math Interpreter in Python (2/4) - Parser*. Septiembre 30, 2021, de YouTube Sitio web: https://www.youtube.com/watch?v=TwKWUj033vY.
- CodePulse. (2020). *Simple Math Interpreter in Python (3/4) – Interpreter*. Septiembre 30, 2021, de YouTube Sitio web: https://www.youtube.com/watch?v=45epnUwFALo.
- CodePulse. (2020). *Simple Math Interpreter in Python (4/4) – Unit testing*. Septiembre 30, 2021, de YouTube Sitio web: https://www.youtube.com/watch?v=tczjDCbykyM.

Anexos

Anexo 1: Archivo `main.py`

```
#%% Portada
'''
                Centro de Ciencias Básicas
         Departamento de Ciencias de la Computación
                       Autómatas II
                          7° "A"

                         Intérprete

         Profesor: Francisco Javier Ornelas Zapata

         Alumnos:
                  Almeida Ortega Andrea Melissa
                  Espinoza Sánchez Joel Alejandro
                  Flores Fernández Óscar Alonso
                  Gómez Garza Dariana
                  González Arenas Fernando Francisco
                  Orocio García Hiram Efraín

Fecha de Entrega: Aguascalientes, Ags., 3 de octubre de 2021
'''


from lexer import Lexer
from parser_ import Parser
from interpreter import Interpreter

while True:
  try:
    text = input("calc > ")
    lexer = Lexer(text)
    tokens = lexer.generate_tokens()
    parser = Parser(tokens)
    tree = parser.parse
  #print(list(tokens))
  #print(tree)
    if not tree: continue
    interpreter = Interpreter()
    value = interpreter.visit(tree)
    print(value)
  except Exception as e:
    print(e)
```

Anexo 2: Archivo `lexer.py`

```
from tokens import Token, TokenType

WHITESPACE = ' \n\t'
DIGITS = '0123456789'

class Lexer:
     def __init__(self, text):
          self.text = iter(text)
          self.advance()

     def advance(self):
          try:
               self.current_char = next(self.text)
          except StopIteration:
               self.current_char = None

     def generate_tokens(self):
          while self.current_char != None:
               if self.current_char in WHITESPACE:
                    self.advance()
               elif self.current_char == '.' or self.current_char
in DIGITS:
                    yield self.generate_number()
               elif self.current_char == '+':
                    self.advance()
                    yield Token(TokenType.PLUS)
               elif self.current_char == '-':
                    self.advance()
                    yield Token(TokenType.MINUS)
               elif self.current_char == '*':
                    self.advance()
                    yield Token(TokenType.MULTIPLY)
               elif self.current_char == '/':
                    self.advance()
                    yield Token(TokenType.DIVIDE)
               elif self.current_char == '(':
                    self.advance()
                    yield Token(TokenType.LPAREN)
               elif self.current_char == ')':
                    self.advance()
                    yield Token(TokenType.RPAREN)
               else:
                    raise      Exception(f"Illegal      character
'{self.current_char}'")
```

```
     def generate_number(self):
          decimal_point_count = 0
          number_str = self.current_char
          self.advance()

          while self.current_char != None and (self.current_char ==
'.' or self.current_char in DIGITS):
               if self.current_char == '.':
                    decimal_point_count += 1
                    if decimal_point_count > 1:
                         break

               number_str += self.current_char
               self.advance()

          if number_str.startswith('.'):
               number_str = '0' + number_str
          if number_str.endswith('.'):
               number_str += '0'

          return Token(TokenType.NUMBER, float(number_str))
```

Anexo 3: Archivo nodes.py

```
from dataclasses import dataclass

@dataclass
class NumberNode:
      value: any

      def __repr__(self):
            return f"{self.value}"

@dataclass
class AddNode:
      node_a: any
      node_b: any

      def __repr__(self):
            return f"({self.node_a}+{self.node_b})"

@dataclass
class SubtractNode:
      node_a: any
      node_b: any

      def __repr__(self):
            return f"({self.node_a}-{self.node_b})"

@dataclass
class MultiplyNode:
      node_a: any
      node_b: any

      def __repr__(self):
            return f"({self.node_a}*{self.node_b})"

@dataclass
class DivideNode:
      node_a: any
      node_b: any

      def __repr__(self):
            return f"({self.node_a}/{self.node_b})"

@dataclass
class PlusNode:
      node: any
```

```
     def __repr__(self):
          return f"(+{self.node})"

@dataclass
class MinusNode:
     node: any

     def __repr__(self):
          return f"(-{self.node})"
```

Anexo 4: Archivo parser_.py

```
from tokens import TokenType
from nodes import *

class Parser:
  def __init__(self, tokens):
    self.tokens = iter(tokens)
    self.advance()

  def raise_error(self):
    raise Exception("Invalid sintax")

  def advance(self):
    try:
      self.current_token = next(self.tokens)
    except StopIteration:
      self.current_token = None

  def parse (self):
    if self.current_token == None:
      return None

    result = self.expr()

    if self.current_token != None:
      self.raise_error()

    return result

  def expr(self):
    result = self.term()

    while self.current_token != None and self.current_token.type in
(TokenType.PLUS, TokenType.MINUS):
      if self.current_token.type == TokenType.PLUS:
        self.advance
        result = AddNode(result, self.term())
      elif self.current_token.type == TokenType.MINUS:
        self.advance()
        result = SubtractNode(result, self.term())

    return result

  def term(self):
    result = self.factor()
```

```
    while self.current_token != None and self.current_token.type in
(TokenType.MULTIPLY, TokenType.DIVIDE):
      if self.current_token.type == TokenType.MULTIPLY:
        self.advance
        result = MultiplyNode(result, self.factor())
      elif self.current_token.type == TokenType.DIVIDE:
        self.advance()
        result = DivideNode(result, self.factor())

    return result

  def factor(self):
    token = self.current_token

    if token.type == token.type.LPAREN:
      self.advance
      result = self.expr()

      if self.current_token.type != TokenType.RPAREN:
        self.raise_error()

      self.advance()
      return result

    if token.type == TokenType.NUMBER:
      self.advance
      return NumberNode(token.value)

    elif token.type == TokenType.PLUS:
      self.advance
      return PlusNode(self.factor())

    elif token.type == TokenType.MINUS:
      self.advance
      return MinusNode(self.factor())

    self.raise_error()
```

Anexo 5: Archivo `tokens.py`

```
from dataclasses import dataclass
from enum import Enum

class TokenType(Enum):
      NUMBER    = 0
      PLUS      = 1
      MINUS     = 2
      MULTIPLY  = 3
      DIVIDE    = 4
      LPAREN    = 5
      RPAREN    = 6

@dataclass()
class Token:
      type: TokenType
      value: any = None

      def __repr__(self):
            return self.type.name + (f":{self.value}" if self.value
!= None else "")
```

Anexo 6: Archivo `interpreter.py`

```
from nodes import *
from values import Number

class Interpreter:
  def visit(self, node):
    method_name = f'visit_{type(node).__name__}'
    method = getattr(self, method_name)
    return method(node)

  def visit_NumberNode(self, node):
    return Number(node.value)

  def visit_AddNode(self, node):
    return          Number(self.visit(node.node_a).value          +
self.visit(node.node_b).value)

  def visit_SubtractNode(self, node):
    return          Number(self.visit(node.node_a).value          -
self.visit(node.node_b).value)

  def visit_MultiplyNode(self, node):
    return          Number(self.visit(node.node_a).value          *
self.visit(node.node_b).value)

  def visit_DivideNode(self, node):
    try:
      return         Number(self.visit(node.node_a).value         /
self.visit(node.node_b).value)
    except:
      raise Exception("RuntimeError")

  def visit_PlusNode(self, node):
    return self.visit(node.node)

  def visit_MinusNode(self, node):
    return Number(-self.visit(node.node).value)
```

Anexo 7: Archivo `values.py`

```
from dataclasses import dataclass

@dataclass
class Number:
  value: float

  def __repr__(self):
    return f"{self.value}"
```

Anexo 8: Archivo `lexer_test.py`

```
import unittest
from tokens import Token, TokenType
from lexer import Lexer

class TestLexer(unittest.TestCase):
    def test_empty(self):
        tokens=list(Lexer("").generate_tokens())
        self.assertEqual(tokens, [])


    def test_empty(self):
        tokens=list(Lexer(" \t\n  \t\t\n\n").generate_tokens())
        self.assertEqual(tokens, [])

    def test_numbers(self):
        tokens=list(Lexer("123             123.456             123.
.456").generate_tokens())
        self.assertEqual(tokens, [
            Token(TokenType.NUMBER, 123.000),
            Token(TokenType.NUMBER, 123.456),
            Token(TokenType.NUMBER, 123.000),
            Token(TokenType.NUMBER, 000.456),
            Token(TokenType.NUMBER, 000.000),
            ])

    def test_operators(self):
        tokens=list(Lexer("+-*/").generate_tokens())
        self.assertEqual(tokens, [
            Token(TokenType.PLUS),
            Token(TokenType.MINUS),
            Token(TokenType.MULTIPLY),
            Token(TokenType.DIVIDE),
            ])

    def test_parens(self):
        tokens= list(Lexer("()").generate_tokens())
        self.assertEqual(tokens, [
            Token(TokenType.LPAREN),
            Token(TokenType.RPAREN),
            ])

    def test_all(self):
        tokens=   list(Lexer("27   +   (43   /   36   -   48)   *
51").generate_tokens())
```

```
        self.assertEqual(tokens, [
            Token(TokenType.NUMBER, 27),
            Token(TokenType.PLUS),
            Token(TokenType.LPAREN),
            Token(TokenType.NUMBER, 43),
            Token(TokenType.DIVIDE),
            Token(TokenType.NUMBER, 36),
            Token(TokenType.MINUS),
            Token(TokenType.NUMBER, 48),
            Token(TokenType.RPAREN),
            Token(TokenType.MULTIPLY),
            Token(TokenType.NUMBER, 51),
            ])
```

Anexo 9: Archivo parser_test.py

```
import unittest
from tokens import Token, TokenType
from parser_ import Parser
from nodes import *

class TestParser(unittest.TestCase):
    def test_empty(self):
        tokens=[]
        node= Parser(tokens).parse()
        self.assertEqual(node, None)

    def test_numbers(self):
        tokens= [Token(TokenType.NUMBER, 51.2)]
        node= Parser(tokens).parse()
        self.assertEqual(node, NUmberNode(51.20))


    def test_individual_operations(self):
        token= [
            Token(TokenType.NUMBER, 27),
            Token(TokenType.PLUS),
            Token(TokenType.NUMBER, 14),
            ]



        node= Parser(tokens).parse()
        self.assertEqual(node,              AddNode(NumberNode(27),
NumberNode(14)))

        token= [
            Token(TokenType.NUMBER, 27),
            Token(TokenType.MINUS),
            Token(TokenType.NUMBER, 14),
            ]

        node= Parser(tokens).parse()
        self.assertEqual(node,         SubtractNode(NumberNode(27),
NumberNode(14)))

        token= [
            Token(TokenType.NUMBER, 27),
            Token(TokenType.MULTIPLY),
            Token(TokenType.NUMBER, 14),
```

```
            ]

        node= Parser(tokens).parse()
        self.assertEqual(node,         MultiplyNode(NumberNode(27),
NumberNode(14)))

        token= [
            Token(TokenType.NUMBER, 27),
            Token(TokenType.DIVIDE),
            Token(TokenType.NUMBER, 14),
            ]

        node= Parser(tokens).parse()
        self.assertEqual(node,           DivideNode(NumberNode(27),
NumberNode(14)))

    def test_full_expression(self):
        tokens=[
            Token(TokenType.NUMBER, 27),
            Token(TokenType.PLUS),
            Token(TokenType.LPAREN),
            Token(TokenType.NUMBER, 43),
            Token(TokenType.DIVIDE),
            Token(TokenType.NUMBER, 36),
            Token(TokenType.MINUS),
            Token(TokenType.NUMBER, 48),
            Token(TokenType.RPAREN),
            Token(TokenType.MULTIPLY),
            Token(TokenType.NUMBER, 51),
            ]
        node= Parser(tokens).parse()
        self.assertEqual(node, AddNode(
            NumberNode(27),
            MultiplyNode(
                SubtractNode(
                    DivideNode(
                        NUmberNode(43),
                        NUmberNode(36),
                        ),
                    NUmberNode(48),
                    ),
                NumberNode(51),
                )
            ))
```

Anexo 10: Archivo `interpreter_test.py`

```
import unittest
from nodes import *
from interpreter import interpreter
from values import Number

class TestInterpreter(unittest.TestCase):
    def test_number(self):
        Interpreter().visit(NumberNode(51.2))
        self.assertEqual(value, Number(51.2))

    def test_individual_operations(self):
        value=          Interpreter().visit(AddNode(NumberNOde(27),
NumberNode(14)))
        self.assertEqual(value, Number(41))

        value=     Interpreter().visit(SubtractNode(NumberNOde(27),
NumberNode(14)))
        self.assertEqual(value, Number(13))

        value=     Interpreter().visit(MultoplyNode(NumberNOde(27),
NumberNode(14)))
        self.assertEqual(value, Number(378))

        value=       Interpreter().visit(DivideNode(NumberNOde(27),
NumberNode(14)))
        self.assertAlmostEqual(value.value, 1.92857, 5)

        with self.assertRaises(Exception):
            Interpreter().visit(DivideNode(NumberNode(27),
NUmberNode(0)))

    def test_full_expression(self):
        tree= AddNode(
            NumberNode(27),
            MultiplyNode(
                SubtractNode(
                    DivideNode(
                        NUmberNode(43),
                        NUmberNode(36),
                        ),
                    NUmberNode(48),
                    ),
                NumberNode(51),
                )
```

```
    )

result= Interpreter().visit(tree)
self.assertAlmostEqual(result.value, -2360.08, 2)
```