UNIVERSIDAD SAN CARLOS DE GUATEMALA
CENTRO UNIVERSITARIO DE OCCIDENTE
DIVISIÓN DE CIENCIAS DE LA INGENIERÍA

ESTUDIANTE:
JORGE ANIBAL BRAVO RODRÍGUEZ

CARNÉ:
202131782

DOCENTE:
CHRISTIAN LÓPEZ

CURSO:
LENGUAJES FORMALES Y DE PROGRAMACIÓN

PRÁCTICA 2

NOTACIÓN DE LA SINTAXIS: Extended Backus-Naur Form -EBNF-

La sintaxis se refiere a la estructura de los elementos de un lenguaje en función de su tipo. Las notaciones Backus-Naur Form y su versión extendida son ampliamente usadas para describir la gramática de un lenguaje, siendo esta un conjunto de reglas (también llamadas **reglas de producción**), haciendo referencia únicamente al conjunto de cadenas válidas de acuerdo a las reglas de su gramática.

- **Una descripción BNF/EBNF es una lista de reglas.**
- **Las reglas se utilizan para definir símbolos con ayuda de otros símbolos.**

TIPOS DE SÍMBOLOS EBNF

- **TERMINALES:** cadenas escritas entre comillas, destinados a usarse tal como son.
- **NO TERMINALES:** nombres que hacen referencia a otras cosas.

NOTACIÓN DE LAS REGLAS GRAMATICALES

- **Lado izquierdo:** describe el nombre de un no terminal para definirlo
- **=:** separa el lado izquierdo del derecho. Se lee “se define como”.
- **Lado derecho:** es la definición del no terminal.

COMPOSICIÓN DE NO TERMINALES

- Secuenciación: combinación de uno o más terminales y/o no terminales.
- Elección: el operador | indica que las partes separadas por él son opciones.
- Opcional: el operador [ ] indica que lo encerrado en él es opcional.
- Repetición: el operador { } indica que puede repetirse cero o más veces.
- Agrupación: el operador ( ) agrupa elementos y establece precedencia.

SINTAXIS PARA PARSER-PY

La sintaxis está inspirada en el documento de definición de la práctica 2 para el curso de Lenguajes Formales y de Programación ss 2023.

Expresiones regulares: usadas en el analizador léxico, abarcan parte de la definición gramatical para EBNF. SON EXPRESIONES REGULARES, NO USAN NOTACIÓN EBNF.

BLOQUES DE CÓDIGO: En la fase inicial de planteamiento, representan una lista dentro de un token de tipo " sintáctico "
En el segundo planteamiento considero que un bloque de código debe tratarse como un **contenedor** que guarda dentro de sí una lista de tokens sintácticos. Este contenedor también puede ser un token especializado con la lista. El contenedor de código es usado por elementos específicos de la sintaxis como **if, else, funciones, etc.**
Las sentencias dentro de los bloques de código deben evaluarse individualmente con autómatas finitos deterministas.

- Expresiones regulares para constantes:
  dígito = 0-9 +
  decimal = dígito + "." dígito +
  cad doblecomilla = " .* "
  cad simplecomilla = ' .* '
  true = "true"
  false = "false"

1. EXPRESIONES: Se considera una expresión todo aquello que retorne un valor.
   expresión = variable # variable declarada?
   expresión = función # void es una función
   expresión = arreglo { arreglo } # arreglo n- dimensional
   expresión = diccionario
   expresión = constante

   # variables declaradas: definir su tipo (puede ser uno de los siguientes)
   constante = dígito | decimal | cad doblecomilla | cad simplecomilla | true | false
   arreglo = "[" constante | variable "]" # variable declarada?
   diccionario = "{" [par clave valor] "}" # puede estar vacío
   par clave valor = identificador ":" constante # constante es terminal
   tupla = "(" { constante } ")" # son inmutables

   op log = " and " | " or "
   op neg = " not " # uso dif de "and" y "or"
   op comp = " == " | " != " | " >= " | " <= " | " < " | " > "
   expresión condicional = [op neg] expresión condicional
   expresión condicional = " true " | " false "
   expresión condicional = expresión condicional op log expresión condicional
   expresión condicional = (expresión) op comp (expresión)
   operador ternario = expresión "if" expresión condicional "else" expresión

2. DECLARACIÓN DE VARIABLES Y ASIGNACIÓN:
   Estructura: identificador operador_asignacion expresión

   variable = identificador
   op asign = " = " | " += " | " -= " | " **= " | " /= " | " //= " | " %= " | " *= "

   asignación = identificador { " , " identificador } op asign expresión { " , " expresión }
   # comprobar -> al menos la misma cantidad de expresiones que identificadores

3. CONDICIONALES:
   if = " if " expresión condicional " : " bloque de código [elif { elif }] [else]
   elif = " elif " expresión condicional " : " bloque de código
   else = " else " bloque de código

   # Un operador ternario es una expresión puesto que devuelve un valor
   # Una expresión condicional es una expresión puesto que devuelve un valor (v/f)

4. CICLOS:

   FOR: Un bucle for en Python es una estructura de control que se utiliza para iterar sobre una secuencia.

   for **variable** in **secuencia** :

   las secuencias pueden ser:
   - **arrays/listas** -> frutas = ['manzana','pera']
   - **rangos** -> range(10)
   - **cadenas de caracteres**
   - **diccionarios**
   - **tuplas** -> tupla = (x, 1, "texto") -> son inmutables

   for = " for " identificador " in " secuencia bloque de código [else]

   # La sentencia **else** es opcional y se ejecuta si la ejecución del for no se interrumpe con un break

   secuencia = arreglo | rango | caddoblecomilla | cadsimplecomilla | diccionario | tupla
   rango = " range " " (" dígito [" , " digito] [" ," digito] ") "

   # El rango se puede crear con 3 argumentos: inicio, fin y tamaño del paso
   # **También validar variable numérica, aquí solo se usó dígito por comodidad**

   WHILE

   while = " while" expresión condicional " : " bloque de código

5. FUNCIONES / MÉTODOS

Bloques de código reutilizables que realizan una tarea específica. Las sentencias pertenecientes a ellos deben tener la misma identación.

función = “def” identificador “ (” [identificador { “ , ” identificador }] “) ” bloque de código [return expresión]
# verificar el tipo de expresiones que se pueden retornar

(REFERENCIA: Bloques de código - documento de definición de la práctica 2 para el curso de Lenguajes Formales y de Programación ss 2023)

Dentro de cada bloque existirán las siguientes sentencias de manera opcional:

- expresiones
- instrucciones
    - condicionales
        - if
        - operador ternario
    - ciclos
        - for
        - for-else
        - while

En la fase inicial de planteamiento, representan una lista dentro de un token de tipo “ sintáctico ”
En el segundo planteamiento considero que un bloque de código debe tratarse como un **contenedor** que guarda dentro de sí una lista de tokens sintácticos. Este contenedor también puede ser un token especializado con la lista. El contenedor de código es usado por elementos específicos de la sintaxis como **if, else, funciones, etc. (REFERENCIA A LA PARTE INICIAL - SINTAXIS DE PARSER PY).**

TOKENS SINTÁCTICOS

La clase **Parser.java** tiene por objetivo analizar una lista enlazada de tokens provenientes del análisis léxico y así crear tokens sintácticos, los cuales son las estructuras reconocidas por el programa apoyado en una serie de autómatas finitos deterministas especializados en cada tipo de estructura sintáctica.

Para ello, se utiliza un tipo de token llamado SyntaxToken.

SyntaxToken
- sentencia (String) - fila, columna (int) - tipo (String)
+ SyntaxToken(sentencia, fila, columna, tipo) + getters + setters

BLOQUES DE CÓDIGO

Los bloques de código son usados por un pequeño número de sentencias.
Estas sentencias son las siguientes:

- if
- elif
- else
- for
- while
- función

A su vez, estos bloques de código son también tokens sintácticos, por lo que me es lógico extender de esa clase, además de especializar el token función, ya que de este se espera que se reciba cero o más parámetros.

CodeBlock (extends SyntaxToken)
- codeBlock (LinkedList < SyntaxToken >)
+ getCodeBlock(): LinkedList < SyntaxToken >

Funcion (extends CodeBlock)
- parameters (LinkedList < SyntaxToken >)
+ getParameters(): LinkedList < SyntaxToken >

LISTAS DE TOKENS

Para el manejo de los tokens obtenidos por el análisis sintáctico se hace uso de dos listas de tipo **token sintáctico.**

- **sintaxTokens:** es una lista enlazada que contiene los tokens de manera general, de manera secuencial.
- **codeBlockTokens:** esta lista es únicamente para almacenar objetos de tipo **bloque de código.** Dentro suyo se encuentran de manera secuencial los bloques de código, por lo que es más fácil recorrerla si únicamente se desea consultar esta información.

FUNCIONAMIENTO DEL RECONOCIMIENTO

El procedimiento de reconocimiento de código es el siguiente:

1. Obtención del código
    - El código a analizar se obtiene de su fuente, se almacena en un String que luego será analizado por el analizador léxico.
2. Análisis léxico
    - Para el análisis léxico se creó un objeto especializado llamado **Lexer (**clase Lexer.java**)** el cual contiene otro objeto de tipo Tokenizer generado mediante la librería JFlex.
        - El objeto generado por JFlex solamente puede hacer uso de un **Reader** por lo que podría parecer lógico acceder a él mediante el uso de archivos. Sin embargo, yo utilicé un StringReader para hacerlo más sencillo.
        - El objeto **Lexer** contiene un método llamado **analize()** el cual recibe el String con el código a analizar léxicamente.
        - En este método existe un ciclo sencillo while que se detiene hasta que el **Tokenizer** devuelva un token nulo.
        - En cada iteración se agrega a la lista de tokens el token obtenido por Tokenizer.
3. Análisis sintáctico
    - La clase responsable del análisis sintáctico es **Parser.java** el cual contiene un método llamado **parse**. Este método recibe la lista de Tokens obtenida desde el Lexer en el paso anterior.
    - Parse **identifica el token inicial** y en base a eso **dispara el análisis usando el o los autómatas finitos deterministas que acepten ese token inicial.**
    - Cuando se reconoce la estructura sintáctica como correcta, el autómata llama a otro método dentro de la clase Parser llamado **agregarTokenSintactico**, el cual recibe el token correspondiente y procede a hacer tres cosas:
        - verificar que la lista de tokens sintácticos no esté vacía, en cuyo caso no lo esté, obtiene el último token sintáctico agregado a la lista, luego verifica si es un token de tipo **bloque de código**. Si lo es, procede a verificar que la indentación del nuevo token a agregar sea mayor al token de tipo bloque, lo que significa que va contenido dentro suyo. De lo contrario, omite este paso.
        - Agregar el nuevo token a la lista general de tokens.

- Verificar si el nuevo token es de tipo **bloque de código**, en cuyo caso, lo añade a la lista de tokens de bloques de código.

Luego de esto, se ejecuta nuevamente el método Parse hasta terminar con la lista de tokens obtenida. Este método se ejecuta de manera recursiva.

IDENTIFICACIÓN DEL TIPO DE AUTÓMATA A USAR

Al inicio de algunas sentencias en el código no se espera ningún tipo de símbolo en específico, por lo que pasar todos los autómatas uno a uno hasta identificar el tipo que debe ser usado no es muy eficiente. A pesar de que este analizador sintáctico no pretende ser súper eficiente si que se puede mejorar un poco la detección de código. Yo propongo el uso de **diccionarios** los cuales puedan disparar el uso de un autómata detectando el tipo de token inicial a procesar.

AUTÓMATAS RECURRENTES

Autómatas que pueden ser usados en distintos casos:

- rango
- arreglo
- diccionario
- par clave / valor
- tupla
- else
- expresión condicional -> todas las expresiones en realidad

DIAGRAMA DE CLASES