



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

LABORATORIO DE LENGUAJES FORMALES Y DE PROGRAMACIÓN
SECCIÓN A
CUARTO SEMESTRE

MANUAL TÉCNICO DE 'PARSER-PY'

YENNIFER MARÍA DE LEÓN SAMUC
REGISTRO ACADÉMICO NO. 202231084

QUETZALTENANGO 05 DE SEPTIEMBRE DEL 2023

ÍNDICE

Contenido

IDE utilizado para la creación del programa	1
Requerimientos para el funcionamiento del programa	1
Diagrama de clases	2
Análisis léxico.....	3
Tipos de tokens y características importantes	3
Patrón de gráfica de lexemas.....	4
Algoritmo de funciones importantes	5
Inicializando el analisis	5
Pintar caracteres.....	5
Guardando archivos.....	5
Análisis sintáctico	7
Gramática	7
Terminales y no terminales:.....	7
Simbolo inicial.....	7
Producciones.....	7
Jerarquía de operaciones	10
Afd que ayudan a validar la sintaxis	12
Validación de <elif block>.....	12
Validación de <def statement>	12
Validación de asignaciones.....	12
Validación de <for statement>.....	13
Validación de <conditionals statements>	13
Validación de la declaración de diccionarios.....	13

Algoritmo del análisis sintáctico 14

 Análisis de bloques..... 14

 Análisis de sub bloques 14

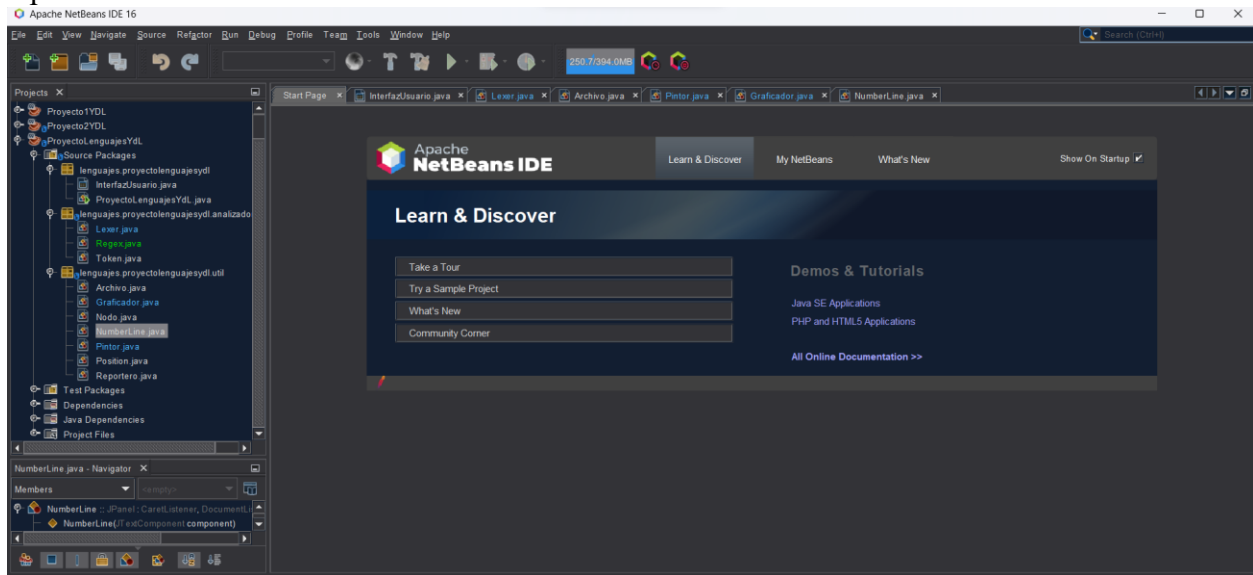
 Para la tabla de símbolos 14

Parser-PY

MANUAL TÉCNICO

IDE UTILIZADO PARA LA CREACIÓN DEL PROGRAMA

Apache NetBeans IDE 16



REQUERIMIENTOS PARA EL FUNCIONAMIENTO DEL PROGRAMA

- Java 19.0.2 2023-01-17

Java(TM) SE Runtime Environment (build 19.0.2+7-44)

Java HotSpot(TM) 64-Bit Server VM (build 19.0.2+7-44, mixed mode, sharing)

Enlace para la descarga: <https://www.oracle.com/java/technologies/downloads/>

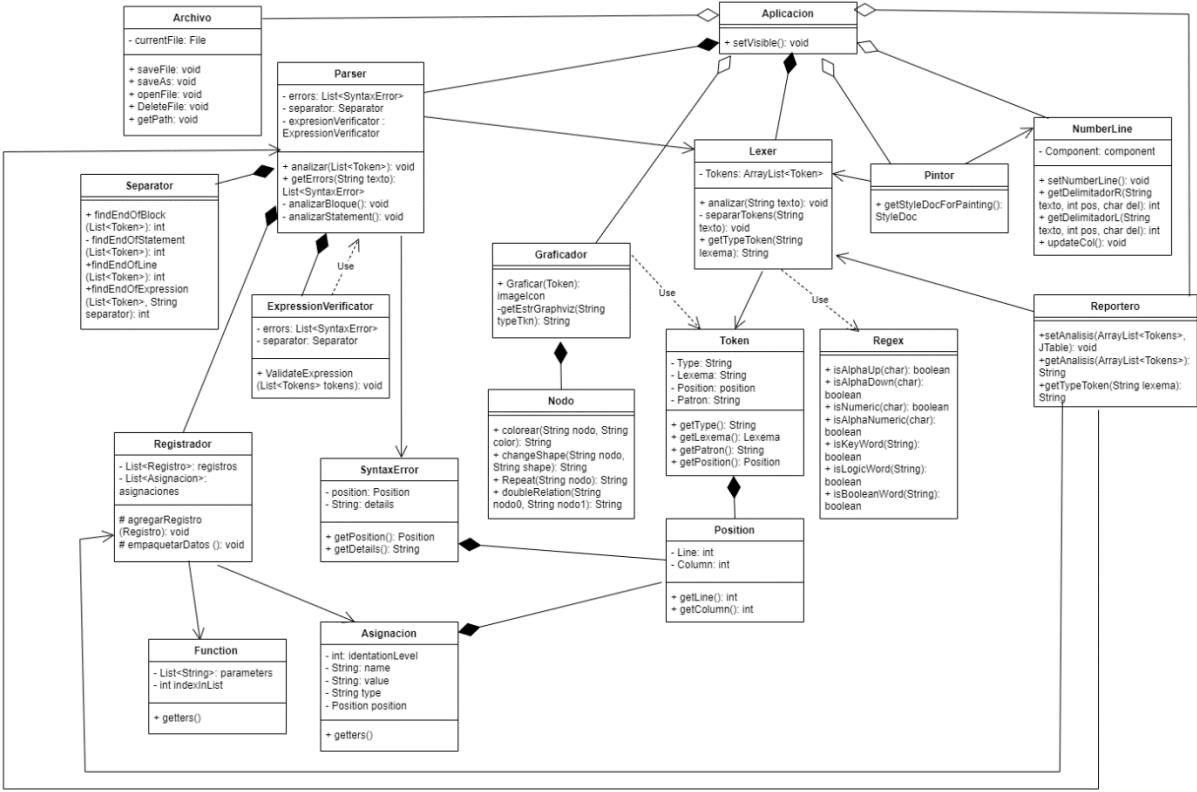


- Graphviz version 8.1.0 (20230707.0739)

Enlace para la descarga: <https://graphviz.org/download/>



DIAGRAMA DE CLASES



Enlace para una mejor visualización del diagrama:

https://drive.google.com/file/d/1sn_3oU6Cr7oAqS0k2b5GRYHnSFSpfmjQ/view?usp=sharing

ANÁLISIS LÉXICO

TIPOS DE TOKENS Y CARACTERÍSTICAS IMPORTANTES

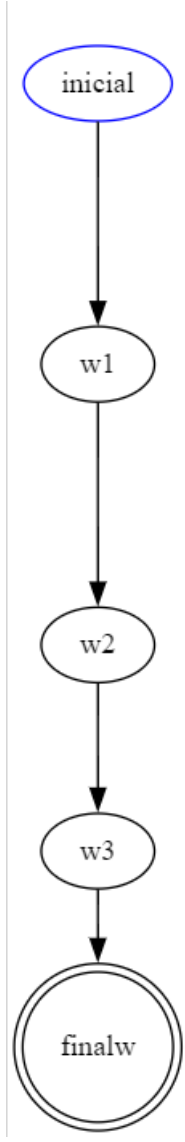
Tipo de token		Patrón	Color	Observaciones
Identificador		[a-zA-Z_][a-zA-Z0-9_]*	Blanco	
Aritméticos		[+ - * / // % *]	Celeste	
Comparación		[== != > < >= <=]	Celeste	
Lógicos		[and or not]	Celeste	
Asignación		= [aritmético][=]	Celeste	
Palabras Clave/ Reservada		[and as assert break class continue def del elif else except False finally for from global if import in is lambda None nonlocal not or pass raise return True try while with yield]	Morado	Deben coincidir mayúsculas y minúsculas
Constantes	Enteros	[0-9]+	Anaranjado	
	Decimales	[0-9]+['.'][0-9]+		
	Cadena	["']*["]'		
	Booleanas	[True False]		
Comentario		[#]+[.]*	Gris	No se toman en cuenta comentarios de más de una línea
Otros		[() { } [] ; :]	Verde	

Nota: la cerradura + significa que al menos se debe tener al menos un carácter requerido de ese tipo; mientras que la cerradura * indica que se puede tener uno de ese tipo o no (dejar en su lugar una cadena vacía “ε”).

Ambas cerraduras indican que se puede tener más de un tipo de carácter indicado.

PATRÓN DE GRÁFICA DE LEXEMAS

Aplica para todos los tokens

Codigo Graphviz	Imagen
<p><i>#nota: cada letra de la palabra será un nodo</i></p> <pre>inicial[color=blue] finalw[shape=doublecircle]</pre> <p><i>#se comienza un ciclo que tiene que abarcar cada letra del lexema a graficar.</i> inicial -> w1 -> w2 -> w3 w3 -> finalw</p>	 <pre>graph TD; inicial((inicial)) --> w1((w1)); w1 --> w2((w2)); w2 --> w3((w3)); w3 --> finalw(((finalw)))</pre>

Nota: el círculo azul denota el comienzo, el círculo con un margen doble denota un estado de aceptación

ALGORITMO DE FUNCIONES IMPORTANTES

INICIALIZANDO EL ANALISIS

// verificar los limites del archivo

Obtener el texto total

Dividir el texto en líneas

// hay que llevar un registro del número de líneas.

Ir recorriendo cada carácter de una línea con la variable columna

Cuando se cambia de línea la variable columna se reinicia

// crear funciones que identifiquen si el carácter es minúscula, mayúscula o numérico

PINTAR CARACTERES

Encontrar un delimitador a la izquierda (el salto de línea más cercano a la izquierda o el inicio del documento)

Encontrar un delimitador a la derecha (el salto de línea más cercano o el fin del documento)

Cortar todo el texto contenido en el editor desde el delimitadorL hasta el delimitadorR

Realizar un análisis léxico del texto cortado

Ir pintando los tokens según la tabla, apoyándonos de su tipo y índice relativo de inicio/fin a la cadena cortada.

GUARDANDO ARCHIVOS

Si se ha indicado un directorio valido

Preguntar si se quiere guardar

Si es verdadero

Obtener el texto del editor y guardarlo en el directorio actual

Si es falso

Cerrar la ventana emergente y no hacer nada

De lo contrario

Preguntar si se quiere guardar el documento

Si es verdadero

Asegurarse de que ingreso un nombre valido, no vacio

Si es un nombre valido

Obtener el texto del editor y crear un nuevo archivo

Si no es un nombre valido

Mostrar una ventana emergente que indique que no se ha podido guardar

Si es falso

Cerrar la ventana emergente y no hacer nada

ANÁLISIS SINTÁCTICO

GRAMÁTICA

TERMINALES Y NO TERMINALES:

Se usa la notación BFN para describir la gramática, utilizando la siguiente clave:

- <No terminal>
 - También conocidos como variables, son todos aquellos que pueden ser descompuestos en otras reglas de la gramática
- ‘terminal’
 - Son aquellos que no pueden ser descompuestos en otras reglas de la gramáticas, en este caso, se toman todos los token primitivos, es decir, todos aquellos que el analizador léxico nos da.
- $\epsilon \rightarrow$ vacío

SIMBOLO INICIAL

\rightarrow ‘block’

PRODUCCIONES

//Bloques de código, son aquellos que abarcan sub bloques más pequeños

$\langle \text{block} \rangle ::= \langle \text{sub block} \rangle \langle \text{block} \rangle$

$\mid \epsilon$

$\langle \text{block r} \rangle ::= \langle \text{sub block r} \rangle \langle \text{block r} \rangle$

$\mid \epsilon$

$\langle \text{block b} \rangle ::= \langle \text{sub block b} \rangle \langle \text{block b} \rangle$

$\mid \epsilon$

//Sub bloques de código, son aquellos que abarcan bloques de código más pequeños que definen sentencias específicas

$\langle \text{sub block} \rangle ::= \langle \text{if block} \rangle$

$\mid \langle \text{while block} \rangle$

$\mid \langle \text{for block} \rangle$

$\mid \langle \text{def block} \rangle$

| <assignment>
 | <use function>
 <sub block r> ::= <sub block>
 | <return statement>
 <sub block b> ::= <sub block>
 | <break statement>

//Definición de bloques de código que contienen sentencias

<if block> ::= <if statement> <block>
 | <if statement> <block> <elif block>
 | <if statement> <block> <else block>
 <elif block> ::= [<elif statement> <block>]+
 | [<elif statement> <block>]+ <else block>
 <else block> ::= <else statement> <block>
 <while block> ::= <while statement> <block b>
 <for block> ::= <for statement><block b>
 | <for statement><block b><else block>
 <def block> ::= <def statement> <block r>
 assignment> ::= 'identifier' [',' 'identifier']* <assignment operator> <expression>[','
 <expression>]* ')'

<use function> ::= 'identifier' '(' '
 | 'identifier' '(' 'identifier' [',' 'identifier']* ')'

//Definición de sentencias particulares

<if statement> ::= 'if' <expression> ':'
 <elif statement> ::= 'elif' <expression> ':'
 <else statement> ::= 'else' ':'
 <while statement> ::= 'while' <expression> ':'
 <for statement> ::= 'for' 'identifier' 'in' <expression> ':'

$\langle \text{def statement} \rangle ::= \text{'def' 'identifier' '(' ' ' ')' ':'}$
 $\quad | \text{'def' 'identifier' '(' ' ' 'identifier' '[' ',' 'identifier' ']' * ')' ':'}$

//Definición de expresiones, todo aquello que retorne un valor

$\langle \text{expression} \rangle ::= \langle \text{number} \rangle \langle \text{derivate expression} \rangle$
 $\quad | \langle \text{single expression} \rangle \langle \text{derivate expression} \rangle$
 $\quad | \text{'not' } \langle \text{expression} \rangle$
 $\quad | \text{'(' } \langle \text{expression} \rangle \text{'})'}$
 $\quad | \langle \text{use function} \rangle$
 $\quad | \langle \text{list} \rangle$
 $\quad | \langle \text{use list/dictionary} \rangle$
 $\quad | \langle \text{dictionary} \rangle$

$\langle \text{derivate expression} \rangle ::= \langle \text{operator} \rangle \langle \text{expression} \rangle$
 $\quad | \langle \text{ternary expression} \rangle$
 $\quad | \varepsilon$

$\langle \text{number} \rangle ::= \text{'+' } \langle \text{single number} \rangle | \text{'-' } \langle \text{single number} \rangle$

$\langle \text{single expression} \rangle ::= \text{'String' } | \langle \text{boolean} \rangle | \langle \text{single number} \rangle | \text{'identifier'}$

$\langle \text{boolean} \rangle ::= \text{'True' } | \text{'False'}$

$\langle \text{single number} \rangle ::= \text{'int' } | \text{'float'}$

$\langle \text{ternary expression} \rangle ::= \text{'if' } \langle \text{expression} \rangle \text{'else' } \langle \text{expression} \rangle$

$\langle \text{operator} \rangle ::= \text{'and' } | \text{'or' } | \text{'sum' } | \text{'rest' } | \text{'power' } | \text{'division' } | \text{'remainder'}$

$\langle \text{list} \rangle ::= \text{'[' ' '}$

$\quad | \text{'[' } \langle \text{expression} \rangle \text{' , ' } \langle \text{expression} \rangle \text{' * ' '}'$

$\langle \text{dictionary} \rangle ::= \text{'{' ' '}$

$\quad | \text{'{' } \langle \text{expression} \rangle \text{' : ' } \langle \text{expression} \rangle \text{' [' , ' } \langle \text{expression} \rangle \text{' : ' } \langle \text{expression} \rangle \text{' * ' '}'$

$\langle \text{use list/dictionary} \rangle ::= \text{'identifier' '[' } \langle \text{expression} \rangle \text{' [' , ' } \langle \text{expression} \rangle \text{' * ' '}'$

JERARQUÍA DE OPERACIONES

Resumen:

- 1- Operaciones entre paréntesis, listas y diccionarios
- 2- Operador unario, como un negativo o positivo que acompañe a un número o identificador.
- 3- Potencias y módulo
- 4- Multiplicación y división
- 5- Sumas y restas
- 6- Operaciones comparativas (mayor que, menor que, igual, diferente, entre otros)
- 7- Negación (not)
- 8- Operaciones lógicas (and or)

Nota: Si existieran operaciones al mismo nivel de prioridad, se leerá de izquierda a derecha.

Gramática de la jerarquía de operaciones:

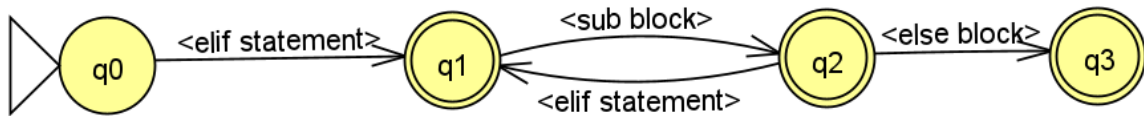
- $\langle \text{logic operation} \rangle ::= \langle \text{not operation} \rangle \langle \text{logic operation} \rangle'$
- $\langle \text{logic operation} \rangle' ::= \text{"and"} \langle \text{not operation} \rangle \langle \text{logic operation} \rangle'$
 $\quad \mid \text{"or"} \langle \text{not operation} \rangle \langle \text{logic operation} \rangle'$
 $\quad \mid \epsilon$
- $\langle \text{not operation} \rangle ::= \text{"not"} \langle \text{not operation} \rangle \mid$
 $\quad \langle \text{comparative operation} \rangle$
- $\langle \text{comparative operation} \rangle ::= \langle \text{basic operation} \rangle \langle \text{comparative operation} \rangle'$
- $\langle \text{comparative operation} \rangle' ::=$
 $\quad \langle \text{comparative symbol} \rangle \langle \text{basic operation} \rangle \langle \text{comparative operation} \rangle'$
 $\quad \mid \epsilon$
- $\langle \text{comparative symbol} \rangle ::= \text{"<"} \mid \text{">"} \mid \text{">="} \mid \text{"<="} \mid \text{"!="} \mid \text{"=="}$
- $\langle \text{basic operation} \rangle ::= \langle \text{medium operation} \rangle \langle \text{basic operation} \rangle'$
- $\langle \text{basic operation} \rangle' ::= \text{"+"} \langle \text{medium operation} \rangle \langle \text{basic operation} \rangle'$
 $\quad \mid \text{"-"} \langle \text{medium operation} \rangle \langle \text{basic operation} \rangle'$
 $\quad \mid \epsilon$

- $\langle \text{medium operation} \rangle ::= \langle \text{extended operation} \rangle \langle \text{medium operation} \rangle'$
- $\langle \text{medium operation} \rangle' ::= "*" \langle \text{extended operation} \rangle \langle \text{medium operation} \rangle'$
 $| "/" \langle \text{extended operation} \rangle \langle \text{medium operation} \rangle'$
 $| "//" \langle \text{extended operation} \rangle \langle \text{medium operation} \rangle'$
 $|\epsilon$
- $\langle \text{extended operation} \rangle ::= \langle \text{unary operation} \rangle \langle \text{extended operation} \rangle'$
- $\langle \text{extended operation} \rangle' ::= "***" \langle \text{unary operation} \rangle \langle \text{extended operation} \rangle'$
 $| "% " \langle \text{unary operation} \rangle \langle \text{extended operation} \rangle'$
 $|\epsilon$
- $\langle \text{unary operation} \rangle ::= "-" \langle \text{number} \rangle$
 $| "+" \langle \text{number} \rangle$
 $|\langle \text{primary expression} \rangle$
- $\langle \text{number} \rangle ::= "int" | "float"$
- $\langle \text{primary expression} \rangle ::= \text{identifier}$
 $|\langle \text{number} \rangle$
 $|\text{String}$
 $|\text{Boolean}$
 $|"(" \langle \text{logic operation} \rangle ")"$
 $|"{" \langle \text{logic operation} \rangle ":" \langle \text{logic operation} \rangle ("," \langle \text{logic operation} \rangle ":" \langle \text{logic operation} \rangle)* "}"$
 $|"{" "}"$
 $|"[" "]"$
 $|"[" \langle \text{logic operation} \rangle ("," \langle \text{logic operation} \rangle)* "]"$

AFD QUE AYUDAN A VALIDAR LA SINTAXIS

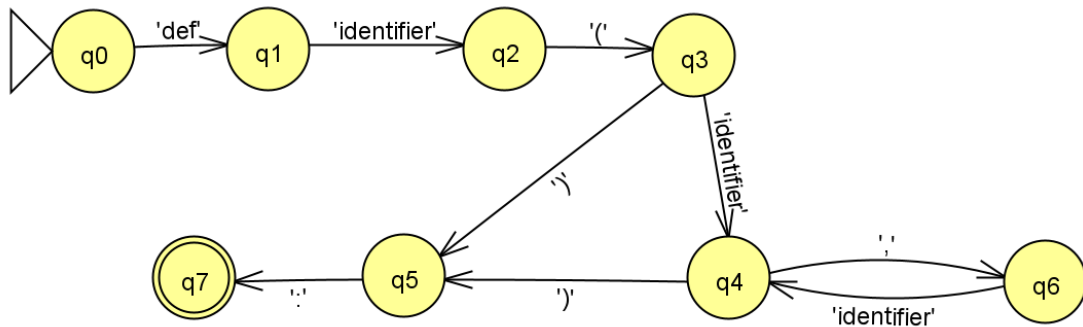
Estos están pensados para validar la estructura de diferentes sentencias

Validación de <elif block>

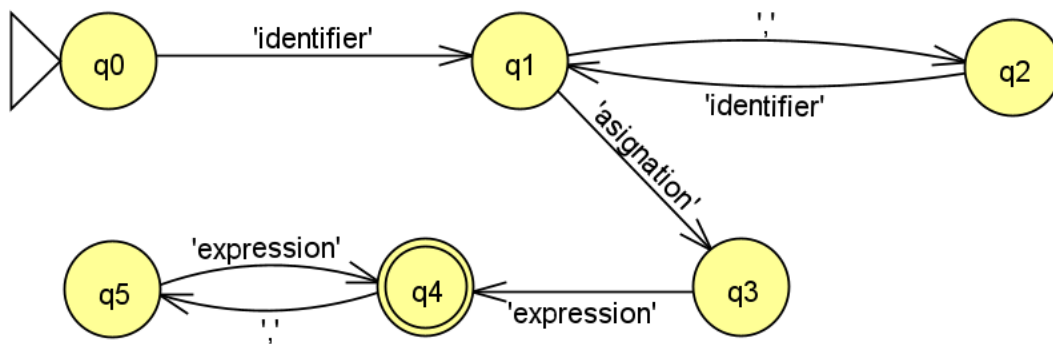


Nota: muchas otras sentencias de bloques siguen el mismo patrón e incluso uno más simple (necesitando nada más, por ejemplo, su respectivo statement y opcionalmente su sub bloque), por lo que no se incluirá su autómata.

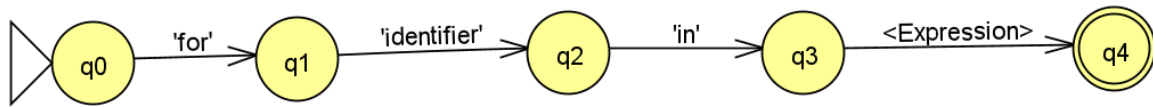
Validación de <def statement>



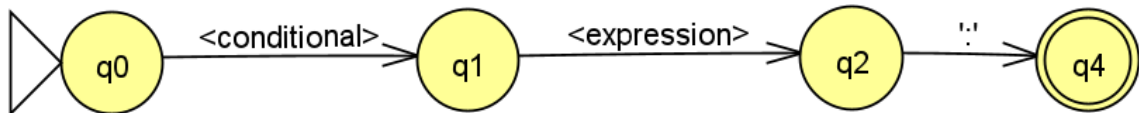
Validación de asignaciones



Validación de <for statement>



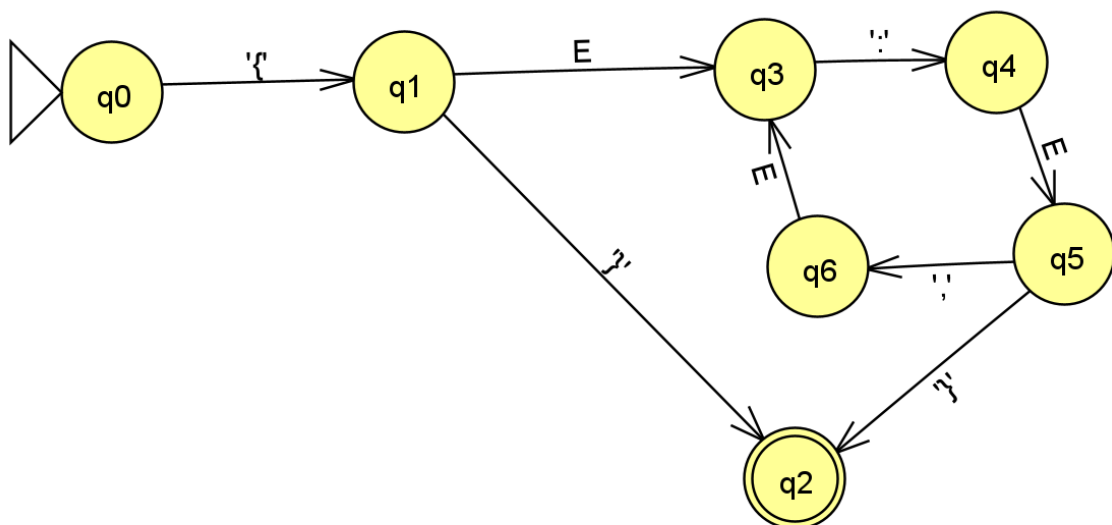
Validación de <conditionals statements>



Notas:

- <conditional> puede ser remplazado con 'if', 'elif' o 'while', pues los tres obedecen a la misma estructura.
- La estructura de <else statement> es muy simple, así que no se ha incluido, para darse una idea, se puede suprimir la transición de "<expression>" y así sería el autómata.

Validación de la declaración de diccionarios



ALGORITMO DEL ANÁLISIS SINTÁCTICO

ANÁLISIS DE BLOQUES

1. De una estructura completa se cortará por “Bloques” (un bloque es un pedazo de código que tiene la misma indentación)
2. VERIFICAR IDENTACIÓN
3. Se vuelve a cortar la estructura sub bloques
4. VERIFICAR IDENTACIÓN
5. Se analiza cada sub bloque dependiendo de su tipo (if, while, for, otros).

ANÁLISIS DE SUB BLOQUES

1. Se corta el sub bloque en el fin de la primera línea
2. Se analiza la línea según el tipo específico de statement (definición de función, if, elif y otros) de la primera línea
3. Se analiza un bloque (cuya indentación es más a la derecha)

PARA LA TABLA DE SÍMBOLOS

➔ Asignación simple

Encontrar el fin de expresión

Si es un solo token

Asignar el token y asignar el tipo si no es igual a identificador, de lo contrario como indefinido

Si hay más de un token

Asignar toda la expresión y setear el tipo como ‘no definido’

Si el fin de expresión, concuerda con el fin de la asignación dejarlo así

De lo contrario agregar una advertencia, de que no se pudo asignar bien algo

➔ Asignación compleja

Encontrar el fin de expresión

Asignar no conocido aun el valor, así como también el tipo

Si el fin de expresión, concuerda con el fin de la asignación dejarlo así

De lo contrario agregar una advertencia, de que no se pudo asignar bien algo