



Desarrollo de aplicaciones avanzadas de ciencias computacionales

Actividad Final Mini Proyecto Parte 1

Presentado por:

Felipe Gabriel Yépez Villacreses

A01658002

Profesores:

Elda Guadalupe Quiroga González

Iván Mauricio Amaya Contreras

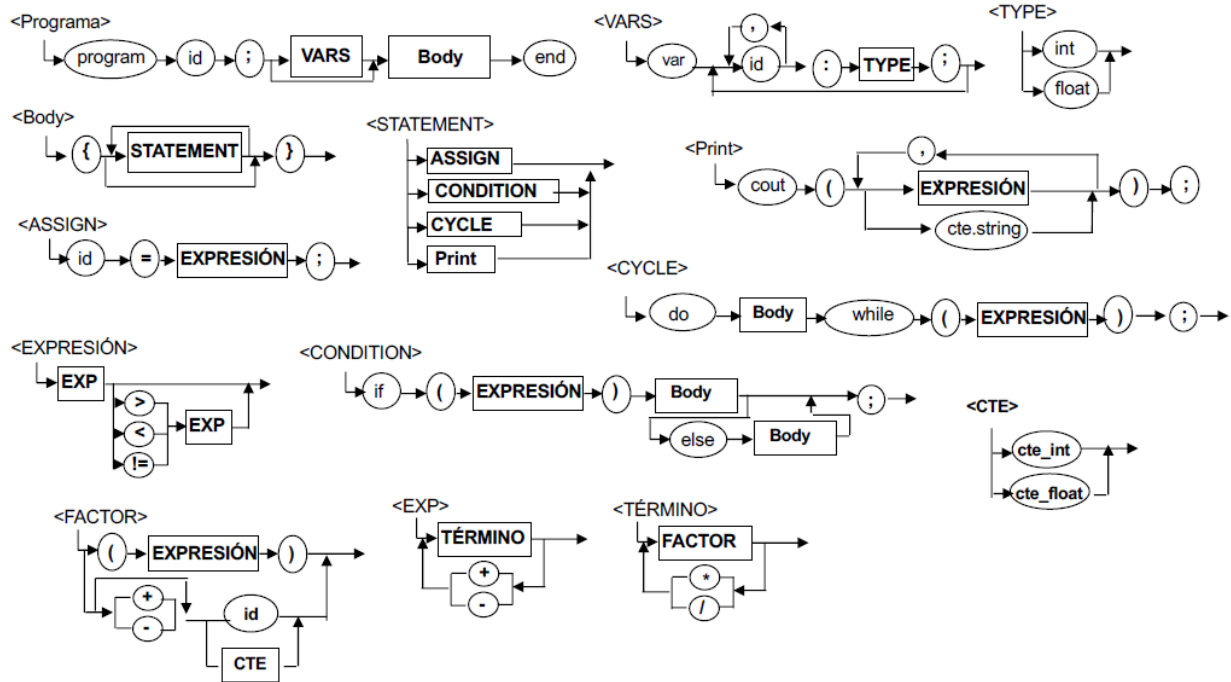
Edgar Covantes Osuna

Alexis Edmundo Gallegos Acosta

Fecha de entrega

Sábado 20 de mayo 2023

Mini Proyecto: Patito



Expresiones Regulares

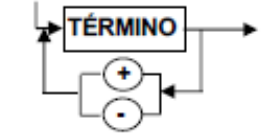
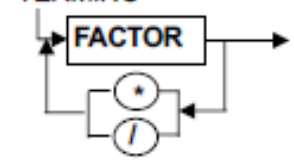
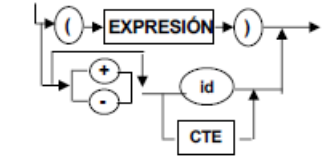
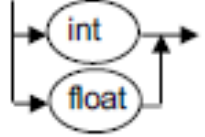
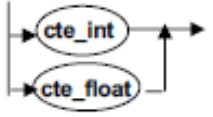
Diseño de expresiones regulares para los diferentes elementos del léxico de lenguaje Patito.

- reserved -> [program end var int float cout if else do while]
- id -> [a-z A-Z] [a-z A-Z 0-9]*
- cte_string -> ".*?"
- cte_int -> [0-9]+
- cte_float -> [0-9]+ . [0-9]+
- leftParenthesis -> (
- rightParenthesis ->)
- leftBrace -> {
- rightBrace -> }
- colon -> :
- coma -> ,
- semicolon -> ;
- equal -> =
- add -> +
- minus -> -
- multiply -> *
- divide -> /
- greaterThan -> >
- lessThan -> <

- not -> !=

Gramáticas libres de contexto

<p><Programa></p>	<p>$\langle \text{program} \rangle \rightarrow \text{program id ; R } \langle \text{body} \rangle \text{ end}$ $R \rightarrow \langle \text{vars} \rangle$ $R \rightarrow \epsilon$</p>
<p><VARS></p>	<p>$\langle \text{vars} \rangle \rightarrow \text{var } O$ $O \rightarrow S P$ $S \rightarrow \text{id}$ $P \rightarrow , O$ $P \rightarrow : \langle \text{type} \rangle ; Q$ $Q \rightarrow \epsilon$ $Q \rightarrow O$</p>
<p><Body></p>	<p>$\langle \text{body} \rangle \rightarrow \{ M \}$ $M \rightarrow \langle \text{statement} \rangle M$ $M \rightarrow \epsilon$</p>
<p><STATEMENT></p>	<p>$\langle \text{statement} \rangle \rightarrow \langle \text{assign} \rangle$ $\langle \text{statement} \rangle \rightarrow \langle \text{condition} \rangle$ $\langle \text{statement} \rangle \rightarrow \langle \text{cycle} \rangle$ $\langle \text{statement} \rangle \rightarrow \langle \text{print} \rangle$</p>
<p><ASSIGN></p>	<p>$\langle \text{assign} \rangle \rightarrow \text{id} = \langle \text{expression} \rangle ;$</p>
<p><CYCLE></p>	<p>$\langle \text{cycle} \rangle \rightarrow \text{do } \langle \text{body} \rangle \text{ while } (\langle \text{expression} \rangle) ;$</p>
<p><CONDITION></p>	<p>$\langle \text{condition} \rangle \rightarrow \text{if } (\langle \text{expression} \rangle) \langle \text{body} \rangle L ;$ $L \rightarrow \epsilon$ $L \rightarrow \text{else } \langle \text{body} \rangle$</p>
<p><EXPRESIÓN></p>	<p>$\langle \text{expression} \rangle \rightarrow \langle \text{exp} \rangle J$ $J \rightarrow \epsilon$ $J \rightarrow K \langle \text{exp} \rangle$ $K \rightarrow >$ $K \rightarrow <$ $K \rightarrow !=$</p>
<p><Print></p>	<p>$\langle \text{print} \rangle \rightarrow \text{cout } (G) ;$ $G \rightarrow H I$ $H \rightarrow \langle \text{expression} \rangle$ $H \rightarrow \text{cte_string}$</p>

	$I \rightarrow , G$
<p><EXP></p> 	$\langle exp \rangle \rightarrow \langle term \rangle E$ $E \rightarrow \epsilon$ $E \rightarrow F \langle term \rangle$ $F \rightarrow +$ $F \rightarrow -$
<p><TÉRMINO></p> 	$\langle term \rangle \rightarrow \langle factor \rangle C$ $C \rightarrow \epsilon$ $C \rightarrow D \langle term \rangle$ $D \rightarrow *$ $D \rightarrow /$
<p><FACTOR></p> 	$\langle factor \rangle \rightarrow (\langle expression \rangle)$ $\langle factor \rangle \rightarrow AB$ $A \rightarrow \epsilon$ $A \rightarrow +$ $A \rightarrow -$ $B \rightarrow id$ $B \rightarrow \langle cte \rangle$
<p><TYPE></p> 	$\langle type \rangle \rightarrow int$ $\langle type \rangle \rightarrow float$
<p><CTE></p> 	$\langle cte \rangle \rightarrow cte_int$ $\langle cte \rangle \rightarrow cte_float$

Funcionamiento Alcanzado

Para realizar el código utilicé la librería PLY(Python Lex-Yacc) en Python. PLY permite crear parsers y compiladores utilizando el algoritmo LALR. Mediante el uso de PLY logré plasmar lo anteriormente descrito en este documento para incluir las expresiones regulares al crear el analizador léxico y la gramática libre de contexto al crear el analizador sintáctico.

El analizador léxico utiliza las expresiones regulares para identificar los tokens del lenguaje. Para cada elemento se revisa si existe una coincidencia en el diccionario del lenguaje y lo mapea a su respectivo token en caso de existir.

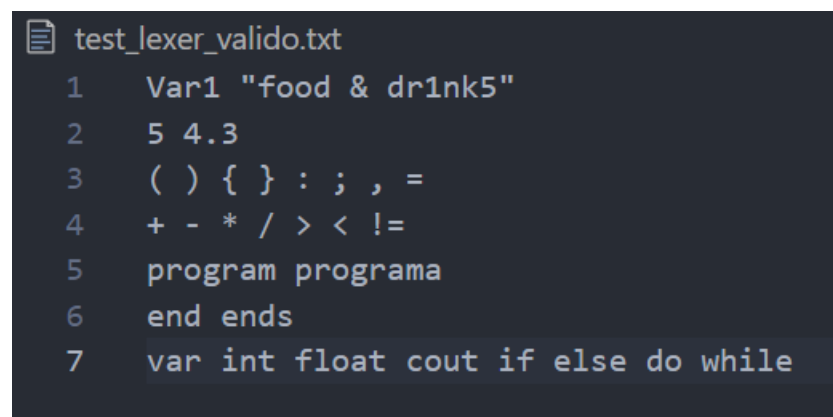
Hasta este punto de desarrollo se puede detectar tokens que no se hayan reconocido en el lenguaje Patito. De igual forma se puede mostrar qué tipo de token se ha detectado correctamente con su respectivo identificador.

El analizador sintáctico puede determinar si un archivo completo del lenguaje Patito sigue las reglas gramaticales, caso contrario determina que hubo un error de sintaxis. El analizador sintáctico o parser permite identificar si la secuencia de elementos encontrados sigue una estructura aceptada por el lenguaje Patito. Utiliza el analizador léxico para reconocer cada token y poder así determinar la secuencia que se sigue de los mismos en un archivo del lenguaje.

Pruebas Analizador Léxico

Para probar el analizador léxico se probó con lo siguiente.

Prueba correcta:



```
test_lexer_valido.txt
1  Var1 "food & dr1nk5"
2  5 4.3
3  ( ) { } : ; , =
4  + - * / > < !=
5  program programa
6  end ends
7  var int float cout if else do while
```

Esta prueba utiliza cada uno de los tokens existentes en el lenguaje Patito con la finalidad de detectar si alguno no está siendo reconocido correctamente. En caso de fallar debería mostrar que el token no fue reconocido correctamente.

De igual forma, adicional a esto, se ejecutará la misma prueba con la finalidad de obtener los tokens mapeados a su respectivo identificador con la finalidad de conocer que cada uno está siendo correctamente identificado.

Prueba incorrecta:

```
test_lexer_invalid.txt
1  Var1 "food & dr1nk5"
2  5 4.3
3  ( ) { } : ; , =
4  + - * / > < !=
5  program programa
6  end ends
7  var int float cout if else do while
8  ^ @ 4a ! a4
```

Al final de este archivo se incluyen algunos tokens que no existen en el lenguaje Patito por lo que al ejecutar esta prueba se espera que muestre los tokens que no fueron reconocidos

Pruebas Analizador Sintáctico

Prueba correcta:

```
test_parser_valido.txt
1  program Felipe;
2
3  var num, i, j: int;
4  mean: float;
5
6  {
7      num = 0;
8      i = 5 * 4 + 3.1;
9      j = 0;
10     if(i != 5){
11         i = 5;
12     }
13     else{
14         mean = i;
15     };
16     if(5 > 4){
17
18     };
19     do{
20         j = j + 1;
21         cout ("Print");
22     } while(j < 3);
23     cout (+3.2 * mean + 4);
24     cout ((-num * 5.3 + 5) + 6 > -4 * 5, "operacion");
25     cout(5, 6);
26     cout(5.7);
27 }
28 end
```

Este caso de prueba sigue la estructura definida por el lenguaje Patito y contiene cada una de sus variables sintácticas y tokens. Esta prueba permite determinar que el parser o analizador sintáctico tiene correctamente estipuladas las reglas definidas por el lenguaje para determinar que las gramáticas libres de contexto fueron correctamente utilizadas y programadas. Al correr el código no debería mostrarse ningún error ya que la estructura seguida es correcta. Se intentó con esta prueba incluir la mayor cantidad de casos aceptados por el lenguaje.

Prueba incorrecta:

```
test_parser_invalid.txt
1  programa Felipe;
2
3  var num, i, j: int;
4  mean: float;
5
6  {
7      num = 0;
8      i = 5 * 4 + 3.1;
9      j = 0;
10     if(i != 5){
11         i = 5;
12     }
13     else{
14         mean = i;
15     };
16     if(5 > 4){
17
18     }
19     do{
20         j = j + 1;
21         cout ("Print");
22     } while(j < 3);
23     cout (+3.2 * mean + 4);
24     cout ((-num * 5.3 + 5) + 6 > -4 * 5, "operacion");
25     cout(5, 6);
26     cout(5.7);
27 }
28 end
```

Este caso de prueba contiene un error ya que al final del segundo bloque condicional, no se delimita su final ya que no cuenta con el token ; un error común que sucede a los programadores. Al ejecutar esta prueba se espera que el parser muestre que existió un error de sintaxis y de un poco de información al respecto de lo que sucedió.

Salidas Obtenidas

```
Testing invalid lexer file...
  Invalid character: { ^ } in line 8   at position 120
  Invalid character: { @ } in line 8   at position 122
  Invalid character: { ! } in line 8   at position 127

Testing valid lexer file...

Testing invalid parser file...
  Syntax error in input
    Expected token before { programa } in line 14   at position 0

Testing valid parser file...
```

Al ejecutar cada uno de los casos de prueba anteriores se obtuvo las salidas esperadas. Al leer los archivos de prueba correctos para el leer y parser no se mostró mensaje de error indicando que estuvieron correctos. Para los casos de prueba inválidos, se logró detectar los tokens incorrectos y la ubicación del error de sintaxis donde se esperaba algo y no se incluyó la secuencia correcta mostrando que el archivo no logra compilar.

De igual forma, se desplegó cada token con su respectivo identificador para conocer que están siendo correctamente detectados.

```
Testing valid lexer file...
ID | Var1
CTE_STRING | food & drink5
CTE_INT | 5
CTE_FLOAT | 4.3
LEFTPARENTHESIS | (
RIGHTPARENTHESIS | )
LEFTBRACE | {
RIGHTBRACE | }
COLON | :
SEMICOLON | ;
COMA | ,
EQUAL | =
ADD | +
MINUS | -
MULTIPLY | *
DIVIDE | /
GREATERTHAN | >
LESSTHAN | <
NOT | !=
PROGRAM | program
ID | programa
END | end
ID | ends
VAR | var
INT | int
FLOAT | float
COUT | cout
IF | if
ELSE | else
DO | do
WHILE | while
```

Se muestra la lista completa de tokens con su identificador.

De esta forma se muestra que las pruebas realizadas fueron ejecutadas con éxito obteniendo las salidas esperadas. Con estas pruebas será posible ejecutarlas en el futuro para que en caso de que se cambie el código se pueda seguir verificando de forma automatizada que el leer y parser funcionan de la forma que se espera.