CENTRO DE ENSEÑANZA TÉCNICA Y SUPERIOR

Escuela de Ingeniería

Teoría de Autómatas

Diseño del Analizador Léxico-Sintáctico

Luis Alejandro Naranjo Osuna – 28301

Mexicali, Baja California, 16 de mayo de 2018

Introducción:

A continuación se presenta la gramática de un lenguaje de programación de alto nivel titulado “tinyPOI”, está basada en la actividad de equipos realizada en clase y tiene el objetivo de sentar bases para desarrollar un analizador léxico.

La gramática es de tipo Libre de Contexto e incluye todas las definiciones necesarias para verificar la validez de instrucciones ingresadas en un archivo de texto.

El lenguaje incluye definiciones para dígitos, números de tipo entero y real, caracteres, cadenas de caracteres, arreglos de datos, operaciones numéricas y booleanas, operadores numéricos, booleanos y comparativos, bloques de control como condiciones y ciclos, tipos de datos, nombres válidos de variables, declaración de variables, asignación de valores, declaración e invocación de funciones, funciones nativas como lectura e impresión y expresiones programáticas.

Diseño:

Diccionario de las instrucciones:

Digito_positivo
Cualquier caracter numérico diferente de 0
Digito
Cualquier caracter numérico
Signo
Símbolos +, -
Numero
La unión de Numero_entero con Numero_real
Numero_entero
Cualquier combinación de dígitos que no comience con 0
Numero_entero_siguientes
Utilizado por Numero_entero para apicar una restricción en su primer dígito
Numero_signo
La concatenación opcional de un Signo con Numero_entero
Numero_real
La concatenación de Numero_signo con el símbolo . (punto) con Numero_decimal
Numero_decimal
Cualquier combinación de dígitos

Caracter
Colección de caracteres alfanuméricos y especiales
Caracter_alfa
Todos los caracteres del abecedario en minúsculas y mayúsculas
Caracteres
Concatenación de *n* Caracter
Cadena
Caracteres entre “” (comillas)

Arreglo
Arreglo_siguientes entre [ ] (corchetes)
Arreglo_siguientes
Regresa la concatenación de *n* variables o nombres de variables

Variable

Regresa Numero, Cadena o Arreglo, con la opción de encerrarlo entre ( ) (paréntesis)

Variable_nombre

Define una combinación de Caracteres que no comience en un Caracter numérico

Variable_nombre_siguientes

Utilizado por Variable_nombre para aplicar una restricción en el primer Caracter

Operando

Regresa una Variable o Variable_nombre

Operador

Regresa un Operador_nuerico u Operador_booleano

Operador_numerico

Colección de símbolos (+, -, *, /, ^, %)

Operador_booleano

Colección de símbolos (&, |)

Operador_comparador

Colección de símbolos (!, ==, <, >)

Operacion_numerica

Regresa un Operando u Operacion, luego un Operador_numerico, y después otro Operando u Operacion, también puede regresar otra Operacion_numerica entre ( ) (paréntesis)

Comparacion

Regresa un Variable o Comparacion, luego un Operador_comparador u Operador_booleano, y después otra Variable o Comparacion, también puede regresar otra Comparacion entre ( ) (paréntesis)

Condicion

Regresa el bloque de control condicional, donde se toma una la palabra inicial es “si”, a este le sigue una Comparacion entre ( ) (paréntesis) y una Expresion entre { } (llaves). Adicionalmente, se le puede concatenar la palabra “sino”, que servirá como bloque de control alternativo en caso de que no se cumplan las condiciones anteriores, pero sí la que este defina, le sigue la misma sintáxis que a la palabra “si”. También se puede agragar la palabra “no”, seguido únicamente de una Expresion entre { } (llaves), usado en caso de que no se cumpla ninguna de las condiciones anteriores.

Ciclo

Regresa un Ciclo_while o un Ciclo_for

Ciclo_while

Comienza con la palabra “mientras”, luego una Comparacion entre ( ) (paréntesis), y finalmente una Expresión entre { } (llaves)

Ciclo_for

Comienza con la palabra “para”, luego una Declaracion o Asignacion opcional, seguida de un ; (punto y coma), una Comparación obligatoria, otro ; (punto y coma) y una Asignacion opcional, todo lo anterior excepto la palabra “for” debe ser encerrado entre ( ) (paréntesis), y finalmente una Expresión entre { } (llaves)

Tipo_dato

“Num”, “Cad”, o “[ ]” (sin las comillas)

Declaracion

Regresa Tipo_dato concatenado con Variable_nombre o con Asignacion

Asignacion

Variable_nombre concatenaco con el símbolo “=” (igual), concatenado con Variable_nombre o con Asignacion

Funcion

Regresa “funcion” (sin las comillas) concatenado con Variable_nombre, concatenado con paréntesis balanceados, ya sean vacíos o con Argumentos dentro, los paréntesis mencionados van seguidos de una Expresion entre { } (llaves)

Funcion_invocacion

Variable_nombre, concatenado con paréntesis balanceados, ya sean vacíos o con Argumentos dentro

Lectura

Regresa el símbolo “?” (sin las comillas), seguido de el nombre de una variable

Impresion

Regresa el símbolo “?” (sin las comillas), seguido de una variable o el nombre de una variable

Expresion

Regresa uno de los siguientes: Condicion, Declaracion, Asignacion, Funcion, Funcion_invocacion, Ciclo, “regresa” seguido de Varaible, “regresa” seguido de Variable_nombre. A esto le sigue un ; (punto y coma).

Alternativamente, puede regresar Expresion Expresion (dos expresiones seguidas)

El símbolo inicial es Expresion.

Tipos de variables y operaciones:

Las variables pueden ser Num y Cad, donde Num es un número entero o punto flotante y Cad es una cadena de caracteres, además.

También se aceptan estructuras Arreglo, que contiene una *n* cantidad de variables de cualquier tipo.

Las operaciones pueden ser numéricas o booleanas.

Los operandos pueden ser la definición de una variable, el nombre de una variable ya existente, o bien, otra operación.

De ser numéricas, solo se pueden utilizar operadores numéricos (+, -, *, /).

De ser booleanas, se pueden utilizar Operadores tanto numéricos como booleanos.

Gramática completa

Digito_positivo:
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

Digito:
- 0
- Digito_positivo

Signo
- +
- -

Numero:
- Numero_signo
- Numero_real

Numero_entero:
- Digito

Digito_positivo Numero_entero_siguientes
Numero_entero_siguientes:
Digito
Numero_entero_siguientes Digito
Numero_signo:
Numero_entero
Numero_signo Numero_entero
Numero_signo Numero_real
Numero_real:
Numero_signo . Numero_decimal
Numero_decimal:
Digito
Numero_decimal Digito

Caracter:
Caracter_alfa
Digito
.
,

:
;
/
?
!
@
#
$
%
^
&
*
(
)
[
]
{
}
<

>
-
_
=
+
`
~

Caracter_alfa:

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
t
u
v
w
x
y
z
A
B
C
D
E
F

G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

Caracteres:
Caracter
Caracter Caracteres

Cadena:
" Caracteres "

Arreglo:
[Arreglo_siguientes]

Arreglo_siguientes:
Variable
Variable_variable
Variable , Arreglo_siguientes
Variable_nombre , Arreglo_siguientes

Variable:
(Variable)
Numero
Cadena
Arreglo

Variable_nombre:
(Variable_nombre)
Caracter_alfa
Caracter_alfa Variable_nombre_siguientes
Variable_nombre_siguientes:
Caracter_alfa Variable_nombre_siguientes
Digito Variable_nombre_siguientes

Operando:
Variable
Variable_nombre
Operador
Operador_numerico
Operador_booleano
Operador_numerico:
+
-
*
/
^
%
Operador_booleano:
&
|
Operacion_numerica:
(Operacion_numerica)
Operando Operador_numerico Operando
Operando Operador_numerico Operacion
Operacion Operador_numerico Operando

Operador_comparador:
!
==
<
>
Comparacion:
(Comparacion)
Variable Operador_comparador Variable
Variable Operador_comparador Comparacion

Comparacion Operador_comparador Variable
Comparacion Operador_booleano Comparacion

Expresion:
Condicion ;
Declaracion ;
Asignacion ;
Funcion ;
Funcion_invocacion ;
Ciclo ;
regresa Variable ;
regresa Variable_nombre ;
Expresion Expresion

Condicion:
si (Comparacion) { Expresion }
si (Comparacion) { Expresion } no { Expresion }
si (Comparacion) { Expresion } sino { Expresion }
si (Comparacion) { Expresion } sino { Expresion } no { Expresion }
Ciclo:
Ciclo_while
Ciclo_for
Ciclo_while:
mientras (Comparacion) { Expresion }
Ciclo_for:
para (Declaracion ; Comparacion ; Asignacion) { Expresion }
para (Asignacion ; Comparacion ; Asignacion) { Expresion }
para (; Comparacion ; Asignacion) { Expresion }
para (Declaracion ; Comparacion ;) { Expresion }
para (Asignacion ; Comparacion ;) { Expresion }
para (; Comparacion ;) { Expresion }

Tipo_dato:
Num
Cad
[]
Declaracion:
Tipo_dato Variable_nombre
Tipo_dato Asignacion

Asignacion:
   Variable_nombre = Variable
   Variable_nombre = Funcion

Funcion:
   funcion Variable_nombre ( ) { Expresion }
   funcion Variable_nombre(Argumentos) { Expresion }
Funcion_invocacion:
   Variable_nombre ( )
   Variable_nombre (Argumentos)

Argumentos:
   Tipo_dato Variable_nombre
   Tipo_dato Variable_nombre, Argumentos

Lectura:
   ? Variable_nombre
Impresion:
   > Variable
   > Variable_nombre

START:
   Expresion

Conclusiones:

Siento que el lenguaje que hice está bastante limitado en cuestión de sus capacidades, creo que esto es porque no me he puesto a pensar en las implicaciones de cómputo de lo que estoy definiendo.

Para que este lenguaje fuera considerado “poderoso” considero que debería ser capaz de manipular bytes directamente, con tipos de dato como Byte y operadores de bits como recorrimiento y considerando los operadores booleanos para funcionar a nivel bit.