UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

COLEGIO DE CIENCIA Y TECNOLOGÍA

LICENCIATURA EN INGENIERÍA DE SOFTWARE

Diseño y desarrollo de un software integrado (IDE, compilador y máquina virtual), para la enseñanza de la lógica de programación a partir de edades de 15 años, por medio de la creación propia de un lenguaje de programación de rutas instruccionales

T E S I S

QUE PARA OPTAR POR EL TÍTULO DE

LICENCIADO EN INGENIERÍA DE SOFTWARE

P R E S E N T A :

JONATHAN GARCÍA GIL

D I R E C T O R

MTRO. ADALBERTO ROBLES VALADEZ

Ciudad de México, mayo de 2023.

UNIVERSIDAD AUTÓNOMA DE LA CIUDAD DE MÉXICO

COLEGIO DE CIENCIA Y TECNOLOGÍA

ACADEMIA DE INFORMÁTICA

Licenciatura en Ingeniería de Software

“ APÉNDICE 2: DETALLE DE LA GRAMÁTICA LIBRE DE CONTEXTO”

TESIS

QUE PARA OPTAR POR EL TÍTULO DE LICENCIATURA EN:

INGENIERÍA DE SOFTWARE

PRESENTA:
JONATHAN GARCÍA GIL

DIRECTOR DE TESIS:
Mtro. Adalberto Robles Valadez
Profesor-Investigador de la Academia de Informática, UACM

Ciudad de México, mayo de 2023

Contenido

1 Gramáticas Libres de Contexto 1
1.1 Bloque de Instrucciones 3
1.2 Declaración de variables 3
1.3 Modificar el valor de una variable 6
1.4 Instrucciones con algún parámetro de entrada 7
1.5 Instrucciones simples. 8
1.6 Instrucción Pintar el suelo 9
1.7 Imprimir Variables 10
1.8 Imprimir cadenas 10
1.9 Llamadas a función 11
1.10 Estructuras de Control 11
1.10.1 Estructura If-Else 11
1.10.2 Validación de condiciones lógicas y de comparación 12
1.10.3 Estructura de Negación 14
1.10.4 Estructura Case 14
1.10.5 Estructuras Iterativas: For 15
1.10.6 Estructuras Do While, y While 16
1.11 Funciones 17

1 Gramáticas Libres de Contexto

Lo primero que se tiene que hacer antes de pasar a las herramientas que generarán el Analizador Sintáctico es tener las Gramáticas Libres de Contexto.

En el **Capítulo 6 del trabajo escrito**, cuando se realiza la prueba del Analizador Sintáctico este no valoraba si, por así decirlo, tenía sentido lo que estaba leyendo. Es decir, solo se dedicó a dividir el texto en tokens e Indicar que el programa fuente era o no, válido. Bueno, ahora es turno de crear esas reglas gramaticales para conocer si se está escribiendo de forma correcta un lenguaje de programación.

El primer paso de este proceso es imaginar: ¿cuál será la forma final del lenguaje de programación con mucho más detalle? En este punto se debe detallar con exactitud cada variación que podría tener el lenguaje.

Comenzando con el programa básico, en el **Capítulo 4.4.2 del trabajo escrito** dije que el programa iba a tener una función principal. Es decir, algo que el programador siempre tendrá que escribir. Sucede en lenguajes como Java, C, C#, C++. Siempre hay código tan repetitivo que tarde o temprano algún IDE lo termina escribiendo de forma automática para ahorrar un poco de trabajo.

En el caso del lenguaje, este será el código que siempre se debe escribir:

```
1.  programa(){
2.        « CÓDIGO DEL USUARIO »
3.  }
```

Partiendo de esto se comenzará a construir la gramática para ese lenguaje.

Se tiene que recordar que un Analizador Sintáctico generará un árbol semántico. Si se compara con un árbol de la vida real: las ramas serán los símbolos no terminales, y las hojas serán los símbolos terminales.

Primero hay que definir un Símbolo Inicial, usualmente se ocupa solo la letra “S” de “Start”. En este caso seré más expresivo esperando ser más claro con la gramática. El símbolo inicial será “Inicio”.

Después del arduo trabajo de definir el símbolo inicial, es momento de abstraer cuáles son los símbolos terminales del lenguaje. Los símbolos terminales, como su nombre lo indica, son aquellos patrones dónde termina el análisis (La hoja de los árboles). En el caso del programa ejemplo anterior hay varios:

- “programa” es un terminal que se ha definido como “PROGRAMA” en el léxico.
- Los paréntesis que abren y cierran son otros terminales.
- Las llaves que abren y cierran son terminales.

La gramática está comenzando a tener forma, observando cómo se visualiza en texto:

```
Inicio ::= PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE LLAVECIERRA
```

Convenientemente se estarán usando algunos símbolos propios de la herramienta CUP como "::=". Además, aquí es cuando tiene sentido haber separado símbolo por símbolo en el Analizador Léxico, ya que ya se sabe cuál símbolo ha leído. Por ejemplo, si se hubiera puesto únicamente "PARÉNTESIS" con una expresión regular " (|) ", en este paso se tendría que estar pensando en cómo reconocer que el paréntesis se está escribiendo de forma correcta. Es decir, validar que hay un paréntesis que abre y otro que cierra. Al haber separado cada símbolo se ha facilitado por mucho, esa tarea de reconocimiento.

Ahora el siguiente paso es pensar en cómo integrar el resto de instrucciones a la gramática. Se podría colocar en la gramática de "Inicio", aunque si se piensa con calma no tiene mucho sentido, porque al final, se tendría un montón de instrucciones y eso provocaría un completo caos. Para esto se tiene los Símbolos No Terminales. Los símbolos no terminales ayudarán a seguir creando más gramáticas muchísimo más completas y complejas.

Continuando con el ejemplo de la primera gramática, ahora se requiere solamente de indicar dónde estará el símbolo no terminal. Para diferenciar los símbolos terminales de los no terminales, a los símbolos no terminales se escribirán en minúsculas, mientras que los símbolos terminales serán escritos en mayúsculas. Este nuevo no terminal irá entre las llaves que abren y cierran, tendrá el nombre de "BloqueDeInstrucciones"

```
Inicio ::= PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones  LLAVECIERRA
```

Pareciera que hasta aquí se ha terminado con la primera gramática, pero no. Aún falta un pequeño análisis más. Se tiene que recordar que se ha puesto que el lenguaje soportará funciones. Si se realiza un análisis de los lenguajes de programación, las funciones usualmente están fuera del programa principal, esperando a ser llamadas. No detallaré exactamente cómo es que se tendría que ver una función (Lo haré más adelante), pero es importante colocarla en este momento para conocer la gramática final del símbolo inicial.

Basándose en el lenguaje C, los bloques de funciones se encontrarán al final de declarar la función principal. No se declaran las funciones. Por lo que el símbolo No Terminal deberá ser colocado al final de la primera gramática. Este no terminal se llamará "Funciones".

```
1.  Inicio ::= PROGRAMA PARENTESISABRE PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones  LLAVECIERRA
    Funciones
2.  ;
```

Con esto se ha terminado la producción de la primera regla gramatical.

1.1 Bloque de Instrucciones

El siguiente paso es enlistar todas las instrucciones que tendrá el lenguaje, esta parte de la gramática solo se compone de No Terminales.

Se observa que los mismos nombres indican todas las posibles combinaciones de instrucciones que se pueden realizar dentro de una función principal o de cualquier otra función.

```
1.  BloqueDeInstrucciones::=
2.        InstruccionDeclaracionDeVariables BloqueDeInstrucciones
3.        | InstruccionModificacionDeValorDeVariables BloqueDeInstrucciones
4.        | InstruccionAvanzar BloqueDeInstrucciones
5.        | InstruccionEspera BloqueDeInstrucciones
6.        | InstruccionGirarALaIzquierda BloqueDeInstrucciones
7.        | InstruccionGirarALaDerecha BloqueDeInstrucciones
8.        | InstruccionTomarObjeto BloqueDeInstrucciones
9.        | InstruccionSoltarObjeto BloqueDeInstrucciones
10.       | InstruccionEliminarObjeto BloqueDeInstrucciones
11.       | InstruccionDesactivarKaboom BloqueDeInstrucciones
12.       | InstruccionPintarSuelo BloqueDeInstrucciones
13.       | InstruccionDejarDePintarSuelo BloqueDeInstrucciones
14.       | InstruccionImprimirVariables BloqueDeInstrucciones
15.       | InstruccionImprimirCadenas BloqueDeInstrucciones
16.       | InstruccionTengoObjetoDelante BloqueDeInstrucciones
17.       | InstruccionTengoBombaDelante BloqueDeInstrucciones
18.       | InstruccionQueTengoDelanteDeMi BloqueDeInstrucciones
19.       | TengoMuroDelanteDeMi BloqueDeInstrucciones
20.       | InstruccionTerminarBloque BloqueDeInstrucciones
21.       | EstrucutraDeControlIf BloqueDeInstrucciones
22.       | EstrucutraDeControlCase BloqueDeInstrucciones
23.       | EstrucutraDeControlFor BloqueDeInstrucciones
24.       | EstrucutraDeControlDoWhile BloqueDeInstrucciones
25.       | EstrucutraDeControlWhile BloqueDeInstrucciones
26.       | LlamadaAFuncion BloqueDeInstrucciones
27.       | /* Epsilon: Sin Bloque de Instrucciones */
28. ;
```

1.2 Declaración de variables

A partir de aquí se debe detenerse a analizar distintas cadenas válidas de las instrucciones que se está diseñando con el objetivo de que tomar en cuenta todos los casos posibles.

Para declarar variables podrán tener los siguientes escenarios:

```
1.  var nuevaVariable;
2.  variable nuevaVariable2 <- 88;
3.  var nuevaVariable, nuevaVariable2 <- (88+otraVariable), nuevaVariable3;
```

Comenzando con lo simple, todas las instrucciones comienzan con el token “VARIABLE”, y al final termina con un punto y coma. Tomando en cuenta lo anterior se construirá la nueva gramática.

```
1.  InstruccionDeclaracionDeVariables ::=
2.        VARIABLE DeclaracionDeVariables PUNTOYCOMA
3.  ;
4.
```

Ahora es turno de analizar lo que contendrá el no terminal “DeclaracionDeVariables” tomando en cuenta los distintos escenarios:

- Se puede recibir únicamente un identificador.
- Se puede recibir una múltiple declaración de identificadores.
- Se puede recibir únicamente el identificador y a su vez asignarle un valor.

La gramática se tendrá que ver de la siguiente manera:

```
1.  DeclaracionDeVariables::=
2.        IDENTIFICADOR
3.        | IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
4.        | IDENTIFICADOR MultiplesDeclaraciones
5.  ;
6.
```

Comenzando a definir qué sucede cuando ocurre una múltiple asignación de valores. se debe de auto referir esta nueva producción hasta llegar al caso base dónde ya no se auto refiere.

```
1.  MultiplesDeclaraciones::=
2.        SEPARADOR IDENTIFICADOR MultiplesDeclaraciones
3.        | SEPARADOR IDENTIFICADOR AsignacionDeValoresAlDeclararVariable MultiplesDeclaraciones
4.        | SEPARADOR IDENTIFICADOR AsignacionDeValoresAlDeclararVariable
5.        | SEPARADOR IDENTIFICADOR
6.  ;
7.
```

Al final, queda analizar cuáles son los elementos válidos que podría contener una asignación de valores. Cada que se declara una variable se puede asignarle un valor, estos elementos válidos serán:

- Asignarle un número
- Asignarle el valor de otra variable (Identificador)
- Asignarle un número aleatorio
- Asignarle una operación aritmética.

Con base en lo anterior se tendrá la siguiente producción:

```
1.  AsignacionDeValoresAlDeclararVariable::=
2.        OPERADORASIGNACION NUMERO
3.        | OPERADORASIGNACION IDENTIFICADOR
4.        | OPERADORASIGNACION NumeroAleatorio
5.        | OPERADORASIGNACION OperacionAritmetica
6.  ;
7.
```

Para una producción de número aleatorio se requiere definir si será completamente aleatorio o el número aleatorio a generar estará entre algún rango de valores. Por lo cual quedará de la siguiente manera:

```
1.  NumeroAleatorio::=
2.      RANDOM PARENTESISABRE NUMERO SEPARADOR NUMERO PARENTESISCIERRA
3.      | RANDOM PARENTESISABRE PARENTESISCIERRA
4.  ;
5.
```

Continuando produciendo lo que será una operación aritmética. Una operación aritmética puede abrir con paréntesis o no. Por lo que comenzar a separar si una operación comienza con paréntesis o no.

```
1.  OperacionAritmetica::=
2.        OperacionSinParentesis
3.        | OperacionConParentesis
4.  ;
5.
```

Continuando con las operaciones que inician con paréntesis. Dentro del paréntesis se puede abrir otro más, por lo que referenciar esta producción con la producción “OperacionAritmetica” de nuevo, así mismo puede existir el caso de que la operación aritmética continue aún cerrado el paréntesis, por ejemplo, (2+2)+9, para esto, igualmente referenciar esta producción con otra nueva llamada “continuaOperacion”. Estas 2 nuevas producciones quedarán así:

```
1.  OperacionConParentesis::=
2.      PARENTESISABRE OperacionAritmetica PARENTESISCIERRA continuaOperacion
3.  ;
4.
```

```
1.  continuaOperacion::=
2.      ContinuacionOperacionSinParentesis
3.      | /* No continua nada después de un paréntesis que cierra */
4.  ;
5.
```

Continuando con el caso en el que ya no existan más paréntesis por tomar en cuenta. Ahora se comenzará a producir la operación aritmética como tal. Se comienza identificando los posibles valores que pueden generar una suma aritmética. Es decir, se puede obtener el valor de una variable, se podrá comenzar con un número o incluso, ¿por qué no? Generar un número aleatorio. Después de esta verificación, se requiere otra producción para continuar generando la operación. La gramática quedará así:

```
1.  OperacionSinParentesis::=
2.        IDENTIFICADOR ContinuacionOperacionSinParentesis
3.        | NUMERO ContinuacionOperacionSinParentesis
4.        | NumeroAleatorio ContinuacionOperacionSinParentesis
5.  ;
6.
```

Ahora toca el turno de continuar auto referenciando la misma producción hasta el caso final, es decir, no queden más elementos que validar.

```
1.  ContinuacionOperacionSinParentesis::=
2.        OperadoresAritmeticos IDENTIFICADOR ContinuacionOperacionSinParentesis
3.        | OperadoresAritmeticos NUMERO ContinuacionOperacionSinParentesis
4.        | OperadoresAritmeticos NumeroAleatorio ContinuacionOperacionSinParentesis
5.        | OperadoresAritmeticos OperacionConParentesis
6.        | OperadoresAritmeticos IDENTIFICADOR
7.        | OperadoresAritmeticos NUMERO
8.        | OperadoresAritmeticos NumeroAleatorio
9.  ;
10.
```

Es necesario generar una nueva producción llamada “OperadoresAritmeticos” debido a que en el Analizador Léxico se tienen 3 distintas formas de identificar los operadores aritméticos. La nueva producción quedará de la siguiente manera:

```
1.  OperadoresAritmeticos::=
2.        OPERADORARITMETICO
3.        | OPERADORARITMETICOSUMA
4.        | OPERADORARITMETICORESTA
5.  ;
6.
```

Así se ha concluido solo con la parte para declarar variables en el lenguaje.

1.3 Modificar el valor de una variable

Ahora es el turno de verificar el caso dónde se tenga que modificar el valor de una variable, este paso ya es sencillo, pues se pueden referenciar lo anteriormente realizado. En este caso se puede referenciar esta producción a la que lleva el nombre de “AsignacionDeValoresAlDeclararVariable”.

```
1.  InstruccionModificacionDeValorDeVariables::=
2.        IDENTIFICADOR AsignacionDeValoresAlDeclararVariable PUNTOYCOMA
3.  ;
4.
```

1.4 Instrucciones con algún parámetro de entrada

Las instrucciones con algún parámetro de entrada serán las que podrá realizar el personaje, por ejemplo, si se quiere mover 5 veces en el mapa no se va a escribir 5 veces la instrucción. Por lo que esta instrucción podrá llevar un parámetro de entrada que le indicará a la instrucción cuántas veces debe repetirse.

Por ejemplo, la instrucción avanzar quedará de la siguiente manera:

```
1.  InstruccionAvanzar::=
2.        AVANZAR PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

La producción de parámetros de entrada, podrán recibir los siguientes elementos:

- Puede no recibir parámetros (Querrá decir que solo se ejecuta 1 vez)
- El valor de una variable
- Un número
- Una operación aritmética
- Incluso, un valor aleatorio.

Por lo que la producción deberá tener la siguiente manera:

```
1.  ParametrosDeEntradaDeUnaInstruccion::=
2.        IDENTIFICADOR
3.        | NUMERO
4.        | NumeroAleatorio
5.        | OperacionAritmetica
6.        | /* Los parámetros de entrada pueden ir vacíos */
7.  ;
8.
```

Lo mismo se debe hacer con cada una de las instrucciones que podrán recibir un parámetro, y referenciar a la producción “ParametrosDeEntradaDeUnaInstruccion”, quedando estas instrucciones de la siguiente manera:

```
1.  InstruccionEspera::=
2.        ESPERA PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionGirarALaIzquierda::=
2.        IZQUIERDA PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionGirarALaDerecha::=
2.        DERECHA PARENTESISABRE ParametrosDeEntradaDeUnaInstruccion PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

1.5 Instrucciones simples.

Las instrucciones simples serán las que el personaje podrá ejecutar, pero no tendrán parámetros de entrada porque no lo necesitan. Las instrucciones son las siguientes:

```
1.  InstruccionTomarObjeto::=
2.        TOMAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionSoltarObjeto::=
2.        SOLTAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionEliminarObjeto::=
2.        ELIMINAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionDesactivarKaboom::=
2.        DESACTIVARKABOOM PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionDejarDePintarSuelo::=
2.        DEJAPINTAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionTengoObjetoDelante::=
2.        OBJETODELANTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionTengoBombaDelante::=
2.        KABOOMDEFRENTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionQueTengoDelanteDeMi::=
2.        QUETENGODELANTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  TengoMuroDelanteDeMi::=
2.        MURODELANTE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionTerminarBloque::=
2.        TERMINARBLOQUE PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

1.6 Instrucción Pintar el suelo

La instrucción pintar el suelo llevará parámetros de entrada, pero estos no serán como los pasados. Sino que podrá recibir la palabra de una paleta de colores predeterminada. Por ejemplo, pintar_suelo(verde). Así mismo se tiene de crear otra producción de colores. Debido a que se puede dejar el parámetro de entrada vacío, se debe tomar en cuenta un color predeterminado.

```
1.  InstruccionDejarDePintarSuelo::=
2.        DEJAPINTAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  InstruccionImprimirVariables::=
2.        IMPRIMIRVARIABLE PARENTESISABRE IDENTIFICADOR PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

1.7 Imprimir Variables

Esta instrucción simplemente imprimirá el valor de una variable. Tiene el objetivo de facilitar al usuario final una instrucción control que informe de los valores de las variables durante la ejecución de un programa. Es bastante útil en el depuramiento.

```
1.  InstruccionImprimirVariables::=
2.        IMPRIMIRVARIABLE PARENTESISABRE IDENTIFICADOR PARENTESISCIERRA PUNTOYCOMA
3.  ;
```

1.8 Imprimir cadenas

A diferencia de la instrucción anterior, ahora se puede construir cadenas concatenando cadenas de texto y valores de variables. Esta producción está fuertemente basada en Java y no en C. Personalmente me gusta más la forma en la que se producen cadenas en Java que en C. Este será el único elemento inspirado en Java.

```
1.  InstruccionImprimirCadenas::=
2.        IMPRIMIRCADENA PARENTESISABRE Cadenas PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

La producción “Cadenas” será la encargada de generar una cadena compuesta. Esta primera producción podrá aceptar una cadena de texto o algún valor de variable y al final, continuar en otra producción llamada “ContinuacionDeCadenas”.

```
1.  Cadenas::=
2.        IDENTIFICADOR ContinuacionDeCadenas
3.        | CADENA ContinuacionDeCadenas
4.  ;
```

En esta producción se volverá a ocupar un operador aritmético, pero para este caso no será para realizar una operación aritmética, sino para unir una cadena de texto con los valores de algún identificador.

```
1.  ContinuacionDeCadenas::=
2.        OPERADORARITMETICOSUMA IDENTIFICADOR ContinuacionDeCadenas
3.        | OPERADORARITMETICOSUMA CADENA ContinuacionDeCadenas
4.        | /* Ningúna cadena más */
5.  ;
```

1.9 Llamadas a función

La llamada a función es otra instrucción importante en el lenguaje, los parámetros de entrada de una función son dinámicos, pueden tener ningún parámetro o pueden tener varios.

Para esto se necesitan las siguientes producciones, la primera es cómo se define la llamada a función y las siguientes 2 definen los parámetros de entrada.

```
1.  LlamadaAFuncion::=
2.        IDENTIFICADOR PARENTESISABRE Parametros PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  Parametros::=
2.        NUMERO ParametrosContinuacion
3.        | IDENTIFICADOR ParametrosContinuacion
4.        | NumeroAleatorio ParametrosContinuacion
5.        | OperacionAritmetica ParametrosContinuacion
6.        | /* Puede no llevar identificadores */
7.  ;
8.
```

```
1.  ParametrosContinuacion::=
2.        SEPARADOR NUMERO ParametrosContinuacion
3.        | SEPARADOR IDENTIFICADOR ParametrosContinuacion
4.        | SEPARADOR NumeroAleatorio ParametrosContinuacion
5.        | SEPARADOR OperacionAritmetica ParametrosContinuacion
6.      | /* No envía más parámetros */
7.  ;
8.
```

1.10 Estructuras de Control

Por último, se revisan las estructuras de control que pueden generar el lenguaje.

1.10.1 Estructura If-Else

Comenzando con la estructura if else. Para lograr esto se requiere separar cada bloque de la estructura de control, es decir, se debe tomar en cuenta que solo puede ir la instrucción “if(){}” sola (sin algún else), que puede existir un if else anidado: “if else(){}if else(){}if else(){}”, y que puede aparecer 1 o 0 veces la instrucción “else{}” (sin condición) al final de la instrucción.

Esto es fácil de lograr, las producciones quedarán de la siguiente manera. Hay que notar que en este punto aparece una producción interesante, es la redirección a “BloqueDeInstrucciones”, es decir, que dentro de estas estructuras se encontrará otro subprograma.

```
1.  EstrucutraDeControlIf::=
2.        SI PARENTESISABRE Condicion PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
    EstrucutraDeControlIfElse
3.  ;
4.
```

```
1.  EstrucutraDeControlIfElse::=
2.        SINOSI   PARENTESISABRE   Condicion   PARENTESISCIERRA   LLAVEABRE   BloqueDeInstrucciones
    LLAVECIERRA EstrucutraDeControlIfElse
3.        | EstrucutraDeControlElse
4.  ;
5.
```

```
1.  EstrucutraDeControlElse::=
2.        SINO LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
3.        | /* NO lleva nada*/
4.  ;
5.
```

1.10.2 Validación de condiciones lógicas y de comparación

Ahora, en este punto del proyecto se tiene que realizar la gramática que cubre las validaciones de una condición lógica o de comparación.

Comenzando con lo fácil, las condiciones de comparación. Las condiciones de comparación pueden llegar simples, y otras encerradas en paréntesis, es decir, como en los siguientes ejemplos:

A < 9 , (A<9)

por lo que la gramática se verá de la siguiente manera, aquí tomo en cuenta las comparaciones lógicas que se analizarán en breve.

```
1.  Condicion::=
2.        CondicionDeComparacionSimple
3.        | CondicionLogica
4.        | CondicionDeComparacionConParentesis
5.  ;
6.
```

```
1.  CondicionDeComparacionConParentesis::=
2.        OperadorNegacion PARENTESISABRE CondicionDeComparacionSimple PARENTESISCIERRA
3.  ;
4.
```

Ahora analizando las comparaciones simples, éstas solo pueden comparar valores aritméticos. Por lo que los siguientes elementos que se pueden comparar podrán ser los siguientes:

- Un número.
- El valor de una variable
- Aunque rara vez, un número aleatorio

Otro aspecto a tomar en cuenta es que puede estar escrita únicamente la palabra verdadero o falso. Sin necesidad de evaluar alguna expresión.

```
1.  CondicionDeComparacionSimple::=
2.        IDENTIFICADOR CondicionDeComparacionSimpleContinuacion
3.        | NUMERO CondicionDeComparacionSimpleContinuacion
4.        | NumeroAleatorio CondicionDeComparacionSimpleContinuacion
5.        | BOOLEANO
6.  ;
7.
```

```
1.  CondicionDeComparacionSimpleContinuacion::=
2.        OPERADORDECOMPARACION IDENTIFICADOR
3.        | OPERADORDECOMPARACION NUMERO
4.        | OPERADORDECOMPARACION NumeroAleatorio
5.  ;
6.
```

Continuando con las comparaciones lógicas, en las comparaciones lógicas se pueden comparar 2 valores que pueden ser comparaciones simples (o booleanos). Se tiene que resolver la comparación simple para poder realizar una comparación lógica. Es decir, lo siguiente:

«Comparación Simple» COMPARACIÓN LÓGICA «Comparación Simple»

«Valor booleano» COMPARACIÓN LÓGICA «Comparación Simple»

«Comparación Simple» COMPARACIÓN LÓGICA «Valor booleano»

«Valor booleano» COMPARACIÓN LÓGICA «Valor booleano»

Para evitar errores, el lenguaje obligará al usuario a encerrar en un paréntesis estas comparaciones. La generación de esta gramática es sencilla, pues las validaciones de comparaciones simples ya se han realizado.

```
1.  CondicionLogica::=
2.        CondicionDeComparacionConParentesis OPERADORLOGICO CondicionDeComparacionConParentesis
3.  ;
4.
```

1.10.3 Estructura de Negación

Por último, se declara la producción del operador de negación que puede ir o no. Este operador convertirá un falso en verdadero y viceversa.

```
1.  OperadorNegacion::=
2.        OPERADORNEGACION
3.        | /* No hay Operador*/
4.  ;
5.
```

1.10.4 Estructura Case

La siguiente estructura por realizar, será la estructura Case. Esta estructura compara una variable con distintos casos hasta que se cumple cierta condición. Por cuestiones de diseño y con el objetivo de generar buenas prácticas en la programación, la instrucción "*default*" será obligada para el programador. Y se puede detener su análisis con la instrucción "*break*;" como en el lenguaje C.

Basándose en lo anterior la gramática quedará de la siguiente manera:

```
1.  EstrucutraDeControlCase::=
2.        COMPARAR  PARENTESISABRE  IDENTIFICADOR  PARENTESISCIERRA  LLAVEABRE  Casos  CasoDefault
    LLAVECIERRA
3.  ;
4.
```

```
1.  Casos::=
2.        CASO CondicionParaCasos DOSPUNTOS InstruccionesCasos
3.        | /* Sin más casos */
4.  ;
5.
```

```
1.  InstruccionesCasos::=
2.      BloqueDeInstrucciones FIN PUNTOYCOMA Casos
3.      | BloqueDeInstrucciones Casos
4.  ;
5.
```

```
1.  CondicionParaCasos::=
2.      OperadorDeComparacionOpcional NUMERO
3.      | OperadorDeComparacionOpcional NumeroAleatorio
4.  ;
5.
```

```
1.  OperadorDeComparacionOpcional::=
2.      OPERADORDECOMPARACION
3.      | /* No hay Operador de Comparación */
4.  ;
5.
```

```
1.  CasoDefault::=
2.        DEFAULT DOSPUNTOS BloqueDeInstrucciones FIN PUNTOYCOMA
3.        | DEFAULT DOSPUNTOS BloqueDeInstrucciones
4.  ;
5.
```

1.10.5 Estructuras Iterativas: For

La estructura for se compone de inicialización de valores, condición de paro e instrucciones de decremento o aumento de valores. Están separados por un punto y coma. Aunque en lenguajes como C, todas las instrucciones dentro de los parámetros de un ciclo "*for*" son opcionales, en este lenguaje de programación no lo será. Es decir, en C se puede dejar el espacio de la comparación en blanco, pero para este lenguaje ese caso no será posible.

Para realizar un "*for*" se requieren de las siguientes producciones:

```
1.  EstrucutraDeControlFor::=
2.        PARA     PARENTESISABRE     DeclaracionesParaFor     PUNTOYCOMA     Condicion     PUNTOYCOMA
    InstruccionesParaFor PARENTESISCIERRA LLAVEABRE BloqueDeInstrucciones LLAVECIERRA
3.  ;
4.
```

```
1.  DeclaracionesParaFor::=
2.        IDENTIFICADOR AsignacionDeValoresAlDeclararVariable DeclaracionesParaForContinuacion
3.        |           VARIABLE           IDENTIFICADOR           AsignacionDeValoresAlDeclararVariable
    DeclaracionesParaForContinuacion
4.        | /* Puede ir nada */
5.  ;
6.
```

```
1.  DeclaracionesParaForContinuacion::=
2.        SEPARADOR               IDENTIFICADOR               AsignacionDeValoresAlDeclararVariable
    DeclaracionesParaForContinuacion
3.        |      SEPARADOR      VARIABLE      IDENTIFICADOR      AsignacionDeValoresAlDeclararVariable
    DeclaracionesParaForContinuacion
4.        | /* Ninguna otra declaración  */
5.  ;
6.
```

```
1.  InstruccionesParaFor::=
2.        IDENTIFICADOR                OPERADORARITMETICOSUMA                OPERADORARITMETICOSUMA
    InstruccionesParaForContinuacion
3.        |          IDENTIFICADOR          OPERADORARITMETICORESTA          OPERADORARITMETICORESTA
    InstruccionesParaForContinuacion
4.        | /* Puede Ir nada */
5.  ;
6.
```

```
1.  InstruccionesParaForContinuacion::=
2.        SEPARADOR        IDENTIFICADOR        OPERADORARITMETICOSUMA        OPERADORARITMETICOSUMA
    InstruccionesParaForContinuacion
3.        |     SEPARADOR     IDENTIFICADOR     OPERADORARITMETICORESTA     OPERADORARITMETICORESTA
    InstruccionesParaForContinuacion
4.        | /* Ninguna Otra Operacion */
5.  ;
6.
```

1.10.6 Estructuras Do While, y While

Para este tipo de estructuras ha quedado fácil porque prácticamente se tienen todas las producciones necesarias realizadas. Por lo cual, solo se requiere definir su estructura que es la siguiente:

```
1.  EstrucutraDeControlDoWhile::=
2.        HACER   LLAVEABRE   BloqueDeInstrucciones   LLAVECIERRA   REPITE   PARENTESISABRE   Condicion
    PARENTESISCIERRA PUNTOYCOMA
3.  ;
4.
```

```
1.  EstrucutraDeControlWhile::=
2.        REPITEHASTA   PARENTESISABRE   Condicion   PARENTESISCIERRA   LLAVEABRE   BloqueDeInstrucciones
    LLAVECIERRA
3.  ;
4.
```

1.11 Funciones

Por último, se define lo que será la estructura de las funciones o subprocesos.

Para esta estructura, solo hay que enfocarse en los parámetros de entrada, que pueden ser ninguno o varios parámetros.

```
1.  Funciones::=
2.        IDENTIFICADOR     PARENTESISABRE     ParametrosDeEntrada     PARENTESISCIERRA     LLAVEABRE
    BloqueDeInstrucciones LLAVECIERRA Funciones
3.        | /* No hay funciones */
4.  ;
5.
```

```
1.  ParametrosDeEntrada::=
2.        VARIABLE IDENTIFICADOR MasParametrosDeEntrada
3.        | /* Sin parametros de entrada */
4.  ;
5.
```

```
1.  MasParametrosDeEntrada::=
2.        SEPARADOR VARIABLE IDENTIFICADOR MasParametrosDeEntrada
3.        | /* No más parámetros */
4.  ;
5.
```