**Asignatura:** Procesadores de lenguajes

**Grupo de prácticas:** A3

**Equipo:** 7 (Lenguaje BBAAC)

**Integrantes del grupo:** Marta Benítez Hernández, Mohammed Lahssaini Nouijah, Marina Ruiz Palomino, y Santiago Salazar Cano

**PRÁCTICA 1: DISEÑO DEL LENGUAJE**

**1. Introducción, breve descripción del lenguaje asignado.**

El lenguaje asignado es BBAAC, cuyas siglas hacen referencia a un lenguaje cuya sintaxis está inspirada en el lenguaje de programación C, en el que sus palabras reservadas están en inglés, que posee las listas como estructura de datos considerada como tipo elemental, que además tiene como subprogramas a las funciones y que tiene como estructura de control adicional a la orden repeat-until.

**2. Descripción formal de la sintaxis del lenguaje usando BNF.**

< Programa > ::= < Cabecera\_programa > < bloque >

< bloque > ::= < Inicio de bloque >

< Declar\_de\_variables\_locales >

< Declar\_de\_subprogs >

< Sentencias >

< Fin\_de\_bloque >

< Declar\_de\_subprogs > ::= < Declar\_de\_subprogs >

< Declar\_subprog >

|

< Declar\_subprog > ::= < Cabecera\_subprograma >

< bloque >

< Declar\_de\_variables\_locales >::= < Variables\_locales >

|

< Cabecera\_programa > ::= “program” <identificador>

< Inicio\_de\_bloque > ::= “{“

< Fin\_de\_bloque > ::= “}”

< Variables\_locales > ::= < Variables\_locales >

< Cuerpo\_declar\_variables >

| < Cuerpo\_declar\_variables >

< Cuerpo\_declar\_variables > ::= < tipo\_dato > < identificador > ";"

< Cabecera\_subprog > ::= < tipo\_dato > < identificador > "("

< Parametros > ")"

< Parametros > ::= < Parametros > "," < Parametro >

| < Parametro >

|

< Parametro > ::= < tipo\_dato > < identificador >

< Sentencias > ::= < Sentencias > < Sentencia >

| < Sentencia >

< Sentencia > ::= < bloque >

| < sentencia\_asignacion >

| < sentencia\_if >

| < sentencia\_while >

| < sentencia\_entrada >

| < sentencia\_salida >

| < sentencia\_return >

| < sentencia\_repeat\_until >

| < sentencia\_mover\_lista>

| < sentencia\_comenzar\_lista >

| < sentencia\_lista\_añadir\_en>

< llamada\_subprog > ::= < identificador > "(" < lista\_expresiones\_o\_cadena > ")" “;”

< sentencia\_asignacion > ::= < identificador > “=” <expresion> ”;”

| < identificador > “=” “[”

<lista\_expresiones\_o\_cadena > “]” ”;”

< sentencia\_if > ::= "if" "(" < expresion > ")" < bloque >

< sentencia\_else >

< sentencia\_else > ::= "else" < bloque >

|

< sentencia\_while > ::= "while" "(" < expresion > ")"

< bloque >

< sentencia\_entrada > ::= “scanf” “(“ < lista\_variables > “)” “;”

< sentencia\_salida > ::= “printf" “(”

< lista\_expresiones\_o\_cadena > “)” “;”

< sentencia\_return > ::= “return” < expresion > “;”

< sentencia\_repeat\_until > ::= “repeat” < bloque > “until” “(“

< expresion > “)” “;”

< sentencia\_mover\_lista> ::= < identificador > <op\_unario\_der> “;”

< sentencia\_comenzar\_lista > ::= <op\_unario\_izq> < identificador > “;”

< sentencia\_lista\_añadir\_en> ::= < identificador > “@”

< expresion > “++” < expresion >;

< lista\_variables > ::= < identificador >

| < lista\_variables > ","

< identificador >

< lista\_expresiones\_o\_cadena >::= < expresion >

| < lista\_expresiones\_o\_cadena > ","

< expresion >

< expresion > ::= “(” < expresion > “)”

| < op\_unario\_izq > < expresion >

| < expresion > < op\_unario\_der >

| < expresion > < op\_binario >

< expresion >

| < identificador >

| < constante >

| < llamada\_subprog>

< tipo\_dato > ::= "int"

| "float"

| "char"

| "bool"

| "list" “<” < tipo\_dato > ”>”

< identificador > ::= [a-zA-Z\_][a-zA-Z0-9\_]\*

< constante > ::= < numero >

| < cadena >

| < boolean\_value >

< numero > ::= <numero\_int>

| <numero\_float>

< numero\_int > ::= [0-9]+

< numero\_float > ::= < numero\_int > “.” < numero\_int >”f”

< cadena > ::= “””([a-zA-Z0-9\_])+”””

< boolean\_value > ::= “TRUE”

| “FALSE”

< op\_unario\_izq > ::= "++"

| “--”

| “-”

| "!"

| “#”

| “?”

| “$”

< op\_unario\_der > ::= "++"

| “--”

| “>>”

| “<<”

< op\_binario > ::= "+"

| "-"

| "\*"

| "/"

| "&&"

| "||"

| "=="

| "!="

| "<"

| ">"

| "<="

| ">="

| “^”

| “--”

| “%”

| “\*\*”

| “@”

**3. Definición de la semántica en lenguaje natural.**

Para empezar declarando un programa, tenemos que introducir la palabra clave programa seguida de un identificador que puede ser una letra o más y números (el nombre del programa), tras ello incluimos las llaves { } para delimitar el bloque que hay en su interior, donde puede haber declaradas variables locales, subprogramas o sentencias.

Dentro de las variables locales (que puede no haber) las declaramos indicando el tipo de dato, el identificador o nombre de la variable y terminamos añadiendo un punto y coma. Para los subprogramas tenemos su cabecera (formada por el tipo de dato de la función, un identificador o nombre, y sus parámetros, que pueden ser vacíos y que están formados por su tipo de dato y su identificador o nombre).

Para las sentencias existen diversos tipos, como puede ser un bloque, la asignación (formada por un elemento a la izquierda, un igual y otro elemento a la derecha), la sentencia condicional if (formada por la palabra if que evalúa una expresión y ejecuta el bloque en caso de ser verdadera o ejecuta el bloque opcional del else, en caso de no serla), la sentencia while (formada por la palabra while que mientras exista una condición se ejecutará el bloque indicado), la sentencia de entrada (consistente en la palabra scanf seguida de la lista de variables separadas por comas), la sentencia de salida (consiste en la palabra printf junto a la lista de variables), la sentencia de retorno (encargada de devolver una expresión utilizando la palabra return) y la sentencia repeat until (encargada de repetir el bloque tras la palabra repeat tantas veces como venga determinado por la expresión tras la palabra until).

Existen otras cosas esenciales, como las expresiones que pueden ser identificadores, constantes (formadas por números o cadenas), llamadas de subprogramas o operadores unarios o binarios utilizados adecuadamente; tipos de dato, que como su nombre indica, nos dicen de qué tipo va a ser el identificador a su continuación.

**4. Identificación de los tokens con el máximo nivel de abstracción.**

| Token | Atributos | Patrón | Identificador |
| --- | --- | --- | --- |
| INPUT |  | (“scanf”) | 257 |
| OUTPUT |  | (“printf”) | 258 |
| REPEAT |  | (“repeat”) | 259 |
| UNTIL |  | (“until”) | 260 |
| PROGRAM |  | (“program”) | 261 |
| WHILE |  | (”while”) | 262 |
| IF |  | (”if”) | 263 |
| ELSE |  | (“else”) | 264 |
| RETURN |  | (“return”) | 265 |
| BOOL\_VALUE | 0: TRUE, 1: FALSE | (“TRUE” | “FALSE”) | 266 |
| PYC |  | “;” | 267 |
| COMA |  | “,” | 268 |
| PUNTO |  | “.” | 269 |
| COMILLA |  | “ ” ” | 270 |
| OP\_ASIGNACION |  | “=” | 271 |
| OP\_LISTA | 0: >>, 1: << | (“>>” | “<<”) | 272 |
| OP\_DECREMNT |  | “--” | 273 |
| OP\_INCREMNT |  | “++” | 274 |
| OP\_ACCESO |  | “@” | 275 |
| OP1 |  | (“-”) | 276 |
| OP2 | 0: !, 1: ?, 2: #, 3: $ | (“!” | ”?” | “#” | “$”) | 277 |
| OP3 | 0: +, 1: \*, 2: /, 3: &&, 4: ||, 5: ==, 6: !=, 7: <, 8: >, 9: <=, 10: >=; 11: \*\*;12: %, 13: ^; | ("+" | "\**" | "/" | "&&" | "||" | "==" | "!=" | "<" | ">" | "<=" | ">=" | “* \*\* ” | “ %” | “^” ) | 278 |
| PAR\_IZQ |  | “(” | 279 |
| PAR\_DER |  | “)” | 280 |
| LLAVE\_IZQ |  | “{” | 281 |
| LLAVE\_DER |  | “}” | 282 |
| COR\_IZQ |  | “[“ | 283 |
| COR\_DER |  | “]” | 284 |
| TIPO\_DATO | 0: int, 1: float, 2: char, 3: bool | ("int" | "float" | "char" | "bool") | 285 |
| LISTA | ("list") | 286 | 286 |
| END\_FLOAT |  | (“f”) | 287 |
| ID | Lexema o dirección en la tabla de símbolos | “[a-zA-Z\_][a-zA-Z0-9\_]\*” | 288 |
| NUMERO |  | [0-9]+ | 289 |
| LETRA |  | [a-zA-Z] | 290 |
| CADENA |  | [a-zA-Z0-9\_]+ | 291 |

**PRÁCTICA 2: ESPECIFICACIÓN E IMPLEMENTACIÓN DEL ANALIZADOR LÉXICO**

**1. Confeccionar la tabla de tokens con el máximo nivel de abstracción para lenguaje diseñado en la práctica anterior con la información siguiente:**

| **Nombre del token** | **Expresión regular** | **Código del token** | **Atributos** |
| --- | --- | --- | --- |
| INPUT | (“scanf”) | 257 |  |
| OUTPUT | (“printf”) | 258 |  |
| REPEAT | (“repeat”) | 259 |  |
| UNTIL | (“until”) | 260 |  |
| PROGRAM | (“program”) | 261 |  |
| WHILE | (”while”) | 262 |  |
| IF | (”if”) | 263 |  |
| ELSE | (“else”) | 264 |  |
| RETURN | (“return”) | 265 |  |
| BOOL\_VALUE | (“TRUE” | “FALSE”) | 266 | 0: TRUE, 1: FALSE |
| PYC | “;” | 267 |  |
| COMA | “,” | 268 |  |
| PUNTO | “.” | 269 |  |
| COMILLAS | “ ” ” | 270 |  |
| OP\_ASIGNACION | “=” | 271 |  |
| OP\_LISTA | (“>>” | “<<”) | 272 | 0: >>, 1:<< |
| OP\_DECREMNT | “--” | 273 |  |
| OP\_INCREMNT | “++” | 274 |  |
| OP\_ACCESO | “@” | 275 |  |
| OP\_SUM\_RES | (“+” | “-”) | 276 | 0: +, 1: - |
| OP\_R | (“!” | ”?” | “#”) | 277 | 0: !, 1: ?, 2: #, 3: $ |
| OP\_INIT\_LISTA | “$” | 278 |  |
| OP\_MULT | “\*” | 279 |  |
| OP\_LOGIC | (“&&” | ”||” | “|” | “^”) | 280 | 0: &&, 1: ||, 2: |, 3:^ |
| OP\_AND | “&” | 281 |  |
| OP\_COMP | (“==” | ”!=” | “<=” | “>=”) | 282 | 0: ==, 1: !=, 2: <=, 3: >= |
| MENOR\_QUE | “<” | 283 |  |
| MAYOR\_QUE | “>” | 284 |  |
| OP\_ARIT | (“/” | ”\*\*” | “%”) | 285 | 0: /, 1: \*\*, 2: % |
| PAR\_IZQ | “(” | 286 |  |
| PAR\_DER | “)” | 287 |  |
| LLAVE\_IZQ | “{” | 288 |  |
| LLAVE\_DER | “}” | 289 |  |
| COR\_IZQ | “[“ | 290 |  |
| COR\_DER | “]” | 291 |  |
| TIPO\_DATO | ("int" | "float" | "char" | "bool") | 292 | 0: int, 1: float, 2: char, 3: bool |
| LISTA | ("list") | 293 |  |
| END\_FLOAT | (“f”) | 294 |  |
| ID | [a-zA-Z\_][a-zA-Z0-9\_]\* | 295 | Lexema o dirección en la tabla de símbolos |
| NUMERO | [0-9]+ | 296 |  |
| LETRA | [a-zA-Z] | 297 |  |
| CADENA | [a-zA-Z0-9\_]+ | 298 |  |
| COMENT | “//” | 299 |  |

(Para obtener los tokens con el máximo nivel de abstracción tenemos que asegurarnos que no se puede definir otro token que incluya a tokens ya definidos previamente. Cuanto más alto sea el nivel de abstracción aplicado

para definir los tokens, menor será la complejidad en la fase de análisis léxico y sintáctico.)

**2. Confeccionar la especificación LEX [Lesk75, Levi92] que reconozca los tokens obtenidos en el apartado anterior. Por defecto, cuando se detecta un error en el análisis de léxico, la recuperación del mismo se produce volviendo**

**al estado inicial despreciando la cadena de símbolos donde ha surgido el error.**

**PRÁCTICA 3: ANALIZADOR SINTÁCTICO**

**1. Definir la gramática abstracta del lenguaje, a partir de la especificación BNF obtenida en la primera práctica y teniendo en cuenta la tabla de tokens con el máximo nivel de abstracción.**

<Programa> ::= PROGRAM ID <bloque>

| error <bloque>

<bloque> ::= LLAVE\_IZQ <Declaracion> <Sentencias> LLAVE\_DER

| LLAVE\_IZQ <Sentencias> <Declaracion> LLAVE\_DER

| LLAVE\_IZQ <Declaracion> <Sentencias> <Declaracion> LLAVE\_DER

| LLAVE\_IZQ <Declaracion> LLAVE\_DER

| LLAVE\_IZQ <Sentencias> LLAVE\_DER

| LLAVE\_IZQ LLAVE\_DER

<Declaracion> ::= <declarar\_variable>

| <declarar\_funcion>

| <Declaracion> <declarar\_variable>

| <Declaracion> <declarar\_funcion>

<declarar\_variable> ::= <tipo\_dato> <lista\_ID> PYC

| <tipo\_dato> <sentencia\_asignacion>

| <tipo\_dato> <lista\_ID> error

| error <lista\_ID> PYC

<declarar\_funcion> ::= <tipo\_dato> ID PAR\_IZQ <Parametros> PAR\_DER <bloque>

| <tipo\_dato> ID PAR\_IZQ PAR\_DER <bloque>

<lista\_ID> ::= ID

| <lista\_ID> COMA ID

| <lista\_ID> ID error

<Parametros> ::= <tipo\_dato> ID

| <tipo\_dato> ID OP\_ASIGNACION <expresion>

| <Parametros> COMA <tipo\_dato> ID

| <Parametros> COMA <tipo\_dato> ID OP\_ASIGNACION <expresion>

| ID error

| <Parametros> <tipo\_dato> error

| <tipo\_dato> error

<Sentencias> ::= <Sentencias> <Sentencia>

| <Sentencia>

<Sentencia> ::= <sentencia\_asignacion>

| <sentencia\_if>

| <sentencia\_while>

| <sentencia\_entrada>

| <sentencia\_salida>

| <sentencia\_return>

| <sentencia\_repeat\_until>

| <sentencia\_mover\_lista>

| <sentencia\_comenzar\_lista>

| <sentencia\_lista\_aniadir\_en>

| error PYC

<llamada\_subprog> ::= ID PAR\_IZQ <lista\_expresiones\_o\_cadena> PAR\_DER

| ID PAR\_IZQ PAR\_DER

| ID PAR\_IZQ <lista\_expresiones\_o\_cadena> error

<sentencia\_asignacion> ::= ID OP\_ASIGNACION <expresion> PYC

| ID OP\_ASIGNACION COR\_IZQ

<lista\_expresiones\_o\_cadena> COR\_DER PYC

| ID OP\_ASIGNACION error

<sentencia\_if> ::= IF PAR\_IZQ <expresion> PAR\_DER <bloque>

<sentencia\_else\_if>

| IF PAR\_IZQ <expresion> <bloque>

<sentencia\_else\_if> ::= ELSE IF PAR\_IZQ <expresion> PAR\_DER <bloque>

<sentencia\_else\_if>

| <sentencia\_else>

| ELSE IF PAR\_IZQ <expresion> <bloque>

<sentencia\_else> ::= ELSE <bloque>

| ε

<sentencia\_while> ::= WHILE PAR\_IZQ <lista\_expresiones\_o\_cadena> PAR\_DER

<bloque>

| WHILE PAR\_IZQ <lista\_expresiones\_o\_cadena> <bloque>

<sentencia\_entrada> ::= INPUT PAR\_IZQ <lista\_expresiones\_o\_cadena> PAR\_DER

PYC

| INPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

COMILLAS COMA <lista\_ID> PAR\_DER PYC

| INPUT PAR\_IZQ <lista\_expresiones\_o\_cadena> PAR\_DER

| INPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

COMILLAS COMA <lista\_ID> PAR\_DER error

| INPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

COMILLAS <lista\_ID> PAR\_DER PYC

| INPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

PAR\_DER

<sentencia\_salida> ::= OUTPUT PAR\_IZQ <lista\_expresiones\_o\_cadena> PAR\_DER

PYC

| OUTPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

COMILLAS PAR\_DER PYC

| OUTPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

COMILLAS COMA <lista\_ID> PAR\_DER PYC

| OUTPUT PAR\_IZQ <lista\_expresiones\_o\_cadena> PAR\_DER

error

| OUTPUT PAR\_IZQ COMILLAS <lista\_expresiones\_o\_cadena>

PAR\_DER

<sentencia\_return> ::= RETURN <expresion> PYC

| RETURN PYC

<sentencia\_repeat\_until> ::= REPEAT <bloque> UNTIL PAR\_IZQ <expresion>

PAR\_DER PYC

| REPEAT <bloque> UNTIL PAR\_IZQ <expresion>

PAR\_DER error

<sentencia\_mover\_lista> ::= ID OP\_LISTA PYC

| ID OP\_LISTA ID PYC

<sentencia\_comenzar\_lista> ::= OP\_INIT\_LISTA ID PYC

| ID OP\_INIT\_LISTA <expresion> PYC

<sentencia\_lista\_aniadir\_en> ::= ID OP\_ACCESO <expresion> PYC

| ID OP\_ACCESO <expresion> OP\_INCREMENT

<expresion> PYC

| ID OP\_ACCESO <expresion> error

<lista\_expresiones\_o\_cadena> ::= <expresion>

| <lista\_expresiones\_o\_cadena> COMA

<expresion>

<expresion> ::= <expresion\_binaria>

| <expresion\_unaria>

| <llamada\_subprog>

| <constante>

| ID

| PAR\_IZQ <expresion> PAR\_DER

| PAR\_IZQ <expresion> error

<expresion\_binaria> ::= <expresion> OP\_SUM\_RES <expresion>

| <expresion> OP\_MULT <expresion>

| <expresion> MAYOR\_QUE <expresion>

| <expresion> MENOR\_QUE <expresion>

| <expresion> OP\_LOGIC <expresion>

| <expresion> OP\_AND <expresion>

| <expresion> OP\_COMP <expresion>

| <expresion> OP\_ARIT <expresion>

<expresion\_unaria> ::= OP\_INCREMENT <expresion>

| OP\_DECREMENT <expresion>

| OP\_R <expresion>

| OP\_SUM\_RES <expresion>

| OP\_MULT <expresion>

<tipo\_dato> ::= TIPO\_DATO

| LISTA MENOR\_QUE <tipo\_dato> MAYOR\_QUE

<constante> ::= NUMERO

| NUMERO END\_FLOAT

| CADENA

| BOOL\_VALUE

| LETRA

**2. Convertir la definición anterior en un fichero de entrada YACC [Levi92]. Procesar este fichero con el generador YACC/BYACC/BISON hasta que hayamos eliminado todos los conflictos reduce/reduce y desplaza/reduce.**

**3. Unir este analizador sintáctico obtenido con YACC/BYACC/BISON con el analizador léxico resultado de FLEX (en la segunda práctica), obteniéndose un programa ejecutable que implementa el analizador sintáctico. Comprobar**

**este analizador con ejemplos de programas fuente escritos en nuestro lenguaje (ver información adicional).**