

Intérprete para un lenguaje A

Especificación del lenguaje e implementación mediante C, Lex y Yacc

Autor: Alberto Ruiz Andrés
Antonio Sanjuán de la Mano
Mario Villacorta García
Fecha de entrega: 12 de junio de 2020
Valladolid

Índice de Contenidos

1. Introducción	1
2. Especificaciones léxicas	1
2.1. Identificadores	1
2.2. Comentarios	1
2.3. Separadores	1
2.4. Palabras clave	2
2.5. Constantes	2
2.5.1. Constantes numéricas	2
2.5.2. Constantes lógicas	3
2.5.3. Constantes carácter y tiras de caracteres	3
2.6. Operadores	3
2.6.1. Operadores unarios	3
2.6.2. Operadores de asignación	3
2.6.3. Operadores aritméticos	3
2.6.4. Operadores lógicos	4
2.6.5. Precedencia y asociatividad de los operadores	5
3. Especificaciones sintácticas	5
3.1. Sentencias	5
3.1.1. Sentencias simples	5
3.1.2. Sentencias bloque	5
3.1.3. Sentencia de declaración	5
3.1.4. Sentencia de asignación	6
3.1.5. Sentencias de control	6
3.1.5.1. Estructuras de decisión	6
3.1.5.2. Estructuras de bucle	6
3.2. Expresiones	6
3.2.1. Expresiones unarias	6
3.2.2. Expresiones aritméticas	7
3.2.3. Expresiones lógicas	7
3.3. Funciones predefinidas	8
3.3.1. Funciones del sistema	8
3.3.2. Funciones numéricas	8
4. Implementación del lenguaje A	9
4.1. Ficheros para el reconocedor	9
4.1.1. <i>LenguajeA.l</i>	9
4.1.2. <i>LenguajeA.y</i>	10
4.2. Ficheros AST	11
4.2.1. <i>astree.c</i>	11
4.2.2. <i>symtab.c</i>	13
4.2.3. <i>stduse.c</i>	13

Anexos	14
A. Ficheros del proyecto	14
B. Metodología y división del trabajo	14
C. Materiales complementarios	14
3.1. Esquema de especificaciones	14

Índice de Figuras

1. Representación gráfica del árbol correspondiente a una estructura de control condicional.	12
2. Representación gráfica del árbol correspondiente a una estructura bucle condicional.	12
3. Representación gráfica del árbol correspondiente a una estructura bucle condicional que se ejecuta al menos una vez.	13

1. Introducción

El **lenguaje A** es una implementación simplificada del lenguaje de programación C. Se trata de un lenguaje de programación de alto nivel, interpretado, imperativo con sistema de tipos débil y estático. Está dedicado a las funciones básicas de todo lenguaje de programación.

2. Especificaciones léxicas

2.1. Identificadores

Los identificadores de *A* tienen las siguientes características: [1]

- Están formados por dígitos, letras y algunos signos de puntuación del juego ASCII, más concretamente:
 - Todos los caracteres alfanuméricos.
 - El carácter '`_`'.
- Comienzan por una letra o el carácter '`_`'.
- Son únicos y no son una palabra reservada.
- Se distingue entre mayúsculas y minúsculas (case-sensitive).

2.2. Comentarios

Encontramos dos tipos de comentario en *A*

- Comentarios de única línea: definidos a tras la construcción '`//`'.
- Comentarios multilínea: definidos entre '`/*`' y '`*/`'.

2.3. Separadores

Los separadores del lenguaje son:

- El espacio en blanco para las constantes, literales, expresiones y sentencias.
- El salto de línea para los comentarios.
- Los paréntesis para asociar expresiones.

2.4. Palabras clave

A continuación se presenta una tabla con las palabras reservadas del lenguaje *A* y su utilidad.

Palabra clave	Descripción
if	Hace referencia a la estructura de control que comprueba si se cumple la condición especificada.
else	Hace referencia a la sentencia alternativa a la condición de un bucle if.
do	Hace referencia a la sentencia principal de una estructura de bucle do ... while
while	Hace referencia a la condición que seguirá el bucle anterior. Además hace referencia a un bucle formado unicamente por esta palabra reservada while
print	Hace referencia a la función de salida estándar.
scan	Hace referencia a la función de entrada estándar.
false	Hace referencia al valor lógico false.
var	Hace referencia a la declaración de una variable.
true	Hace referencia al valor lógico true.
sin	Hace referencia a la operación trigonométrica del seno.
cos	Hace referencia a la operación trigonométrica del coseno.
tan	Hace referencia a la operación trigonométrica de la tangente.
arcsin	Hace referencia a la operación trigonométrica del arcoseno.
arccos	Hace referencia a la operación trigonométrica del arcocoseno.
arctan	Hace referencia a la operación trigonométrica de la arcotangente.
mcm [2]	Hace referencia a el cálculo del mínimo común múltiplo de dos expresiones numéricas.
mcd [3]	Hace referencia al cálculo del máximo común divisor de dos expresiones numéricas.
log	Hace referencia al cálculo del logaritmo de una expresión numérica y toma como base otra expresión numérica.

2.5. Constantes

2.5.1. Constantes numéricas

Las constantes numéricas son enteras y reales implementadas a través de tipo flotante de doble precisión. Las constantes positivas no llevan ningún símbolo adicional y las negativas son anteceditas por el operador unario '-'.

Una constante numérica esta compuesta por una parte entera, el carácter '.' y una parte decimal. La parte entera es obligatoria, mientras que el carácter '.' y la parte decimal pueden aparecer o no en función de si la cantidad a representar es entera o no.

Tanto la parte entera como la parte decimal están constituidas por los dígitos del 0 al 9 con una o más apariciones.

2.5.2. Constantes lógicas

Las constantes lógicas se definen a través de los literales *true* y *false*, que representan los valores lógicos.

2.5.3. Constantes carácter y tiras de caracteres

Las constantes carácter corresponden a un único carácter del juego ASCII en entrecomillado simple. Su valor es interpretado como numérico.

Las tiras de caracteres son cualquier concatenación de constantes carácter en entrecomillado doble.

2.6. Operadores

2.6.1. Operadores unarios

El lenguaje define tres unarios:

- '-': Operador de negatividad de constantes numéricas.
- '++': Operador incremento unitario de constantes numéricas.
- '--': Operador decremento unitario de constantes numéricas.
- '!': Operador de negación lógico.

2.6.2. Operadores de asignación

Los operadores de asignación implementados son 5:

- Asignación básica: mediante el operador '='. Se sitúa entre una variable previamente declarada y una expresión cuya evaluación será el nuevo valor.
- Asignaciones operacionales: mediante los operadores unarios '++', '--', '+=', y '-='.

2.6.3. Operadores aritméticos

Los operadores aritméticos se sitúan entre dos expresiones aritméticas. Son los siguientes:

- '+': implementa la operación suma.
- '-': implementa la operación resta.
- '*': implementa la operación producto.
- '/': implementa la operación división real.
- '^': implementa la operación potenciación.
- '%': implementa la operación modulo.
- '++': implementa la operación incremento unitario.

- `'--'`: implementa la operación decremento unitario.
- `'+='`: implementa la operación incremento.
- `'-= '`: implementa la operación decremento.

2.6.4. Operadores lógicos

Los operadores lógicos se sitúan entre dos expresiones lógicas. Son los siguientes:

- `'&&'`: implementa la operación lógica AND.
- `'||'`: implementa la operación lógica OR.
- `'!'`: implementa la operación lógica NOT.
- `'=='`: implementa la operación lógica de comparación de igualdad.
- `'!='`: implementa la operación lógica de comparación de desigualdad.
- `'>'` : implementa la operación lógica de comparación $>$.
- `'>='`: implementa la operación lógica de comparación \geq .
- `'<'`: implementa la operación lógica de comparación $<$.
- `'<='`: implementa la operación lógica de comparación \leq .

2.6.5. Precedencia y asociatividad de los operadores

Basada en la precedencia y asociatividad original de los operadores de C.[4] [5]

Tipo de operador	Operador	Nivel de precedencia	Asociatividad
Asignación	'='	8	derecha
	'++'	1	izquierda
	'--'	1	izquierda
	'+='	1	izquierda
	'-='	1	izquierda
Aritméticos	'+'	3	izquierda
	'-'	3	izquierda
	'-' (unario)	1	derecha
	'*'	2	izquierda
	'/'	2	izquierda
	'%'	2	izquierda
	'^'	1	derecha
	' '	7	izquierda
Lógicos	'&&'	6	izquierda
	'!'	1	derecha
	'=='	5	izquierda
	'!='	5	izquierda
	'>'	4	izquierda
	'≥'	4	izquierda
	'<'	4	izquierda
	'≤'	4	izquierda

3. Especificaciones sintácticas

3.1. Sentencias

Las sentencias son las instrucciones que no devuelven valor o devuelven valor de tipo *void*. Se consideran sentencias tanto las sentencias simples como las sentencias bloque.

3.1.1. Sentencias simples

Las sentencias simples están constituidas por una instrucción terminada en ';'.

3.1.2. Sentencias bloque

Las sentencias bloque están constituidas por una sucesión de sentencias encerradas entre llaves '{' '}'.

3.1.3. Sentencia de declaración

No es necesario declarar una variable en el lenguaje A.

3.1.4. Sentencia de asignación

- Asignación: Se almacena un valor en una variable. Su estructura es: *var* *<variable>* = *<expresión>*.
- Asignación condicional: Se almacena un valor en una variable según el valor de una expresión lógica: *var* *<variable>* = *<expresión lógica>* ? *<expresión si true>* : *<expresión si false>*
- Expresiones unarias: incremento y decremento unitarios.
- Expresiones aritméticas: incremento y decremento.

3.1.5. Sentencias de control

3.1.5.1. Estructuras de decisión

- Bifurcación simple: *if* (*<expresión lógica>*) *<sentencia>*
- Bifurcación doble: *if* (*<expresión lógica>*) *<sentencia bloque>* *else* *<sentencia bloque>*

3.1.5.2. Estructuras de bucle

- Bucle que se ejecuta al menos una vez: *do* *<sentencia bloque>* *while* (*<expresión lógica>*)
- Bucle condicional: *while* (*<expresión lógica>*) *<sentencia bloque>*

3.2. Expresiones

Las expresiones son las instrucciones que devuelven valor de algún tipo distinto de *void*. Se asocian mediante los caracteres de paréntesis '(', ')':

3.2.1. Expresiones unarias

- Incremento unitario: Se suma 1 al valor de una variable numérica y se le asigna ese nuevo valor. La estructura es: *<variable>* ++.
- Decremento unitario: Se suma -1 al valor de una variable numérica y se le asigna ese nuevo valor. La estructura es: *<variable>* --.
- Cambio de signo: Para obtener el opuesto de una expresión, la estructura a seguir es: - *<expresión numérica>*. Devuelve el valor de la expresión numérica original multiplicado por -1.
- Negación: De un modo similar al opuesto, para la negación de una expresión, la estructura que se debe seguir es: ! *<expresión lógica>*. Devuelve el valor contrario al de la expresión lógica original.

3.2.2. Expresiones aritméticas

Devuelven constantes numéricas reales de punto flotante de doble precisión.

- Suma: $\langle \text{expresión numérica} \rangle + \langle \text{expresión numérica} \rangle$.
- Resta: $\langle \text{expresión numérica} \rangle - \langle \text{expresión numérica} \rangle$.
- Incremento: Se suma una cantidad al valor de una variable numérica y se le asigna ese nuevo valor. La estructura es: $\langle \text{variable} \rangle += \langle \text{expresión numérica} \rangle$.
- Decremento unitario: Se resta una cantidad al valor de una variable numérica y se le asigna ese nuevo valor. La estructura es: $\langle \text{variable} \rangle -= \langle \text{expresión numérica} \rangle$.
- Multiplicación: $\langle \text{expresión numérica} \rangle * \langle \text{expresión numérica} \rangle$.
- División: $\langle \text{expresión numérica} \rangle / \langle \text{expresión numérica} \rangle$.
 - Error si la segunda expresión es 0.
- Módulo: $\langle \text{expresión numérica} \rangle \% \langle \text{expresión numérica} \rangle$
 - Error si la segunda expresión es 0.
- Potenciación: $\langle \text{expresión numérica} \rangle \wedge \langle \text{expresión numérica} \rangle$
 - Error si se trata de $0 \wedge 0$.

3.2.3. Expresiones lógicas

Devuelven valores lógicos (*true* o *false*)

- Conjunción: $\langle \text{expresión lógica} \rangle \ \&\&\ \langle \text{expresión lógica} \rangle$.
- Disyunción: $\langle \text{expresión lógica} \rangle \ | \ \langle \text{expresión lógica} \rangle$.
- Negación: $! \langle \text{expresión lógica} \rangle$.
- Comparación de igualdad: $\langle \text{expresión de tipo } X \rangle == \langle \text{expresión de tipo } X \rangle$
- Comparación de desigualdad: $\langle \text{expresión de tipo } X \rangle != \langle \text{expresión de tipo } X \rangle$
- Comparación $>$: $\langle \text{expresión numérica} \rangle > \langle \text{expresión numérica} \rangle$.
- Comparación \geq : $\langle \text{expresión numérica} \rangle \geq \langle \text{expresión numérica} \rangle$.
- Comparación $<$: $\langle \text{expresión numérica} \rangle < \langle \text{expresión numérica} \rangle$.
- Comparación \leq : $\langle \text{expresión numérica} \rangle \leq \langle \text{expresión numérica} \rangle$.

3.3. Funciones predefinidas

3.3.1. Funciones del sistema

- Impresión por salida estándar:
 - $print(<expresion>) : void$
- Lectura por entrada estándar:
 - $scan(<expresión>) : void$

3.3.2. Funciones numéricas

- Logaritmo: $log(<expresión\ numérica>, <expresión\ numérica>) : <expresión\ numérica>$
 $logaritmo(a, b) \rightarrow \log_a b$
 - Error si alguna de las expresiones es ≤ 0
- Máximo común divisor: $mcd(<expresión\ numérica>, <expresión\ numérica>) : <expresión\ numérica>$
 El resultado es el máximo valor que divide exactamente a ambas expresiones.
 - Error si alguna de las dos expresiones $\notin \mathbb{Z}^+$.
- Mínimo común múltiplo: $mcm(<expresión\ numérica>, <expresión\ numérica>) : <expresión\ numérica>$
 El resultado es un múltiplo común de ambas expresiones cuyo valor es el más pequeño posible.
 - Error si alguna de las dos expresiones $\notin \mathbb{Z}^+$.
- Funciones trigonométricas:
 - Seno: $sin(<expresión\ numérica>) : <expresión\ numérica>$
 La expresión de entrada representa un ángulo en radianes.
 - Coseno: $cos(<expresión\ numérica>) : <expresión\ numérica>$
 La expresión de entrada representa un ángulo en radianes.
 - Tangente: $tan(<expresión\ numérica>) : <expresión\ numérica>$
 La expresión de entrada representa un ángulo en radianes.
 - Error si la expresión es $\frac{\pi}{2} + k\pi, k \in \mathbb{Z}$.
 - Arcoseno: $arcsin(<expresión\ numérica>) : <expresión\ numérica>$
 - Error si la expresión de entrada es > 1 .
 - Arcocoseno: $arccos(<expresión\ numérica>) : <expresión\ numérica>$
 - Error si la expresión de entrada es > 1 .
 - Arcotangente: $arctan(<expresión\ numérica>) : <expresión\ numérica>$

- *COMSIMP*: Comentario de una sola línea. (`\ / \ / .*`)
- *COMMULT*: Comentario de varias líneas. `/*"([^*])*+[\^*/])**+/"`

También encontramos algunos métodos de utilidad para la lectura de cadenas, la conversión de cadenas a cadenas en minúsculas o el almacenaje de la cadena en un puntero..

Hemos aportado el reconocimiento de comentarios multilínea [6], de caracteres, de bucles *while* y *do while* y de condicionales doble. Sobre el original, hemos modificado también la separación de comandos mediante nueva línea a través del separador ';' y se permite que los identificadores contengan '_'.

4.1.2. *Lenguaje A.y*

Este es el fichero *yacc*. Este fichero se usa para identificar las partes del programa y crear la estructura de árbol para su posterior ejecución (es el analizador sintáctico del programa). Las partes en las que dividimos los programas son:

- *expresion*: es una expresión que tiene algún tipo asociado (en nuestro caso puede ser una cadena, un carácter o un número real almacenado en un tipo flotante de doble precisión). Algunos ejemplos son la suma, la resta, la función que calcula el logaritmo, el mínimo común múltiplo..
- *ternario*: puede ser una *expresion* o la estructura de control condicional en formato C que tiene la forma *condicion ? cuerpo1 : cuerpo2*. Se ejecutaría *cuerpo1* si la condición tiene valor de verdad y *cuerpo2* en el caso contrario.
- *sentencia*: son expresiones de tipo *void*, es decir, ejecutables. Una sentencia puede ser *asignacion*, *funcionSistema*, *condicional* o *mientras*.
- *asignacion*: son sentencias que declaran y/o dan valor a identificadores. Se encuentran las asignaciones típicas así como las de incremento y decremento. En las asignaciones de un valor a una variable se usa un *ternario* para el valor. El resto de modificaciones de valor de variables solo usan *FLOAT* para el valor del incremento.
- *funcionSistema*: son las funciones de lectura y escritura. Estas se engloban en *scan* y *print* respectivamente.
- *scan*: Función que lee un valor y lo almacena en una variable. Hay dos tipos, la que imprime por pantalla una cadena que recibe por parámetro antes de pedir el valor y la que directamente pide el valor.
- *print*: Función que imprime por pantalla. Puede imprimir el valor que tiene asociado una variable, una cadena o la cadena seguida del valor asociado a una variable.
- *cuerpo*: son las partes que se ejecutan de las estructuras de control condicionales y bucles. Están formadas por una *sentencia* o varias sentencias (definido recursivamente), en el segundo caso han de estar entre los caracteres '{' '}'.
- *mientras*: bucles condicionados y bucles condicionados que se ejecutan al menos una vez. La condición de salida de estos es un *ternario* y el cuerpo es un *cuerpo*.

- *condicional*: estructuras de tipo *if* o *if else*. La condición del *if* es un *ternario* y el cuerpo es un *cuerpo*.
- *elemprog*: Es un elemento del programa que se puede ejecutar. Puede ser *sentencia* o una sentencia vacía (;).
- *prog*: Es el programa en sí, que está formado por un *elemprog* o varios (definido recursivamente).

Todos los no terminales del reconocedor son de un mismo tipo estructurado denominado *sStackType*, con lo que se evita problemas de compatibilidad entre cabeza y cuerpo de las producciones.

4.2. Ficheros AST

4.2.1. *astree.c*

Este fichero es una modificación del fichero *astree.c* creado por Dušan Kolář. En este fichero se encuentra la estructura de datos que empleamos para el almacenaje de las instrucciones del programa y su posterior ejecución. La estructura de datos es de tipo árbol. Esta estructura se crea al comienzo de la ejecución de un programa. Los nodos pueden ser de tipo *float*, *string*, *char* o de tipo árbol AST. Los que tienen los tipos básicos sólo se usan para almacenar valores de dicho tipo e identificadores. Los de tipo AST son nodos que tienen dos hijos (aunque alguno de ellos puede ser vacío); se usan para estructuras más complejas (bucles, estructuras condicionales), expresiones (operaciones aritméticas, lógicas, etc), asignaciones y otras sentencias. Los nodos tienen una etiqueta que indica qué se está almacenando (*WHILE* para los bucles *while*, '=' para las asignaciones, por ejemplo). Esta etiqueta nos permite mediante un switch evaluar expresiones y ejecutar las sentencias. Las modificaciones realizadas sobre el fichero original son las siguientes:

- Hemos añadido otro tipo de árbol, el tipo carácter. Así como la función que lo crea y la que devuelve el valor almacenado.
- En la función *expr()* se han añadido los casos para las funciones que calculan el máximo común divisor y el mínimo común múltiplo, así como las de logaritmo y las trigonométricas que no realizaba Dušan Kolář. La implementación del máximo común divisor se basa en el algoritmo de Euclides.
- En la función *proc()* se han añadido los casos para las bifurcaciones condicionales y bucles. El árbol que representa las bifurcaciones condicionales (Figura 1) tiene dos hijos, en el izquierdo la condición, y en el derecho tiene otro nodo con dos hijos, en el hijo izquierdo se encuentra el cuerpo que se ejecuta si la condición es cierta y en el derecho el cuerpo que se ejecuta si la condición no es cierta. En el caso de no tener segundo cuerpo, simplemente ese hijo está vacío. En los casos para bucles, el árbol tiene un hijo izquierdo con dos hijos si el bucle no tiene por qué ejecutarse, como puede verse en la Figura 2 (bucle *while*), en el izquierdo la condición de salida y en el derecho el cuerpo del bucle. Si se trata de un bucle que se ejecuta al menos una vez (tipo *do while*), tiene un nodo derecho y el nodo izquierdo está vacío (Figura 3). Este nodo tiene un hijo derecho y un hijo izquierdo con el cuerpo del bucle y la condición de salida respectivamente. Para diferenciarlos se comprueba en el nodo principal qué hijo está vacío.

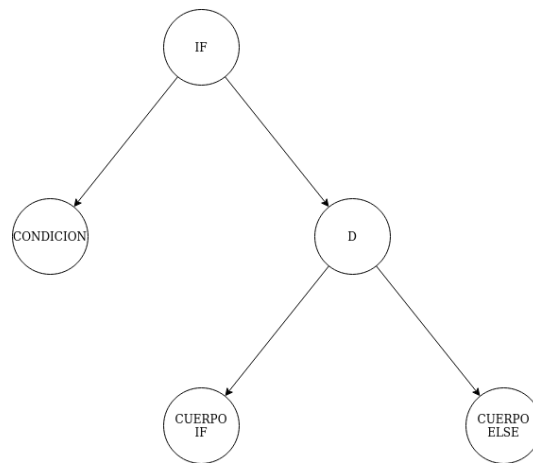


Figura 1: Representación gráfica del árbol correspondiente a una estructura de control condicional.

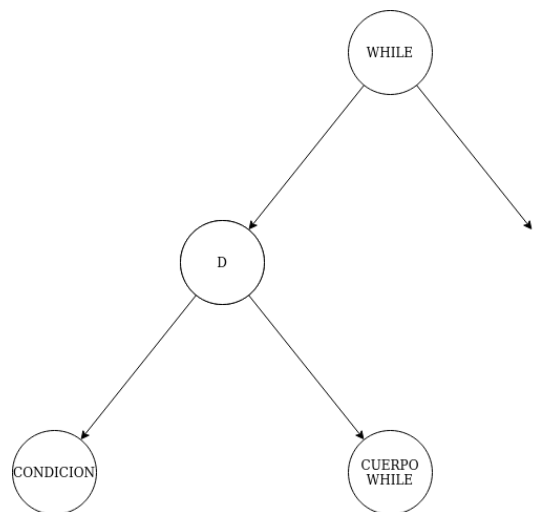


Figura 2: Representación gráfica del árbol correspondiente a una estructura bucle condicional.

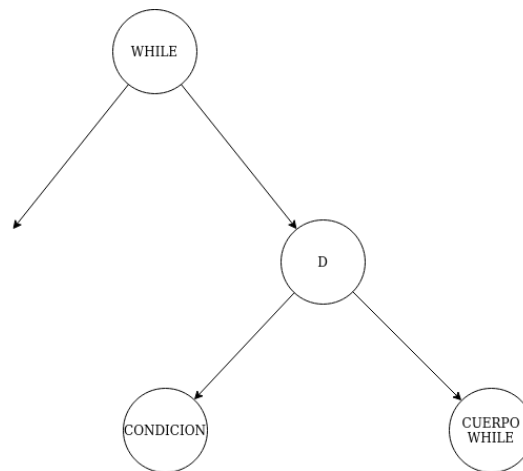


Figura 3: Representación gráfica del árbol correspondiente a una estructura bucle condicional que se ejecuta al menos una vez.

4.2.2. *symtab.c*

Este archivo ha sido creado por Dušan Kolář y es empleado para almacenar las variables y sus valores. La estructura es una lista doblemente enlazada de pares clave-valor. La clave es el nombre de la variable y el valor es el valor asociado a la variable. Este archivo no lo hemos modificado.

4.2.3. *stduse.c*

Este archivo es una librería que incluye métodos para la impresión de errores, reservar memoria, etc. Al basarnos en el código de Dušan Kolář, todas estas funciones que él emplea hay que incluirlas también.

Anexos

A. Ficheros del proyecto

Encontramos los ficheros del proyecto en el siguiente repositorio GitLab:
<https://gitlab.inf.uva.es/albruiz/proyectofinalglf>.

B. Metodología y división del trabajo

Para la realización del proyecto se ha empleado el editor *Visual Studio Code* con la extensión *Live Share* [7], además de llamada de *Discord*; por lo que se han modificado todos los ficheros de manera simultánea por todos los miembros del equipo y las aportaciones son conjuntas.

C. Materiales complementarios

3.1. Esquema de especificaciones

El esquema de las especificaciones léxicas y sintácticas del lenguaje se ha basado en el presentado para Java en *Java 7: Los fundamentos del lenguaje Java, ediciones ENI marzo 2012* de Thierry Groussard y para C en *C/C++: Curso de programación 2015, Anaya 2014* de Miguel Ángel Acera.

Referencias

- [1] C. Pes, “Identificadores en lenguaje c.” Online. http://www.carlospes.com/curso_de_lenguaje_c/01_05_identificadores.php.
- [2] L. C. Benito. <https://gist.github.com/parzibyte/f1232318aa3d1fab57441c30d57fa24c>.
- [3] parzibyte. <https://parzibyte.me/blog/2019/12/18/maximo-comun-divisor-c-algoritmo-euclides/>.
- [4] Wikipedia, “Anexo: Operadores de c y c++,” jun 2020. https://es.wikipedia.org/wiki/Anexo:Operadores_de_C_y_C++.
- [5] Microsoft, “Precedencia y orden de evaluación,” 2019. <https://docs.microsoft.com/es-es/cpp/c-language/precedence-and-order-of-evaluation?view=vs-2019>.
- [6] C. Dodd, “Complete comments regex for lex,” 2012. <https://stackoverflow.com/questions/13569827/complete-comments-regex-for-lex>.
- [7] Microsoft, “Liveshare.” <https://visualstudio.microsoft.com/es/services/live-share/>.