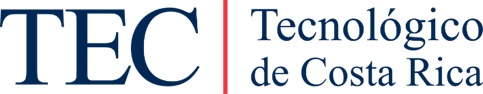
Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación



IC4810 – Compiladores e Intérpretes

Grupo 60

**Tarea - Producciones**

Profesora:

Allan Rodriguez Davila

Integrantes:

Andrew López Herrera - 2021062132

Hillary Malespín Ulloa - 2021106074

Marzo, 2025

Índice

[**Descripción del problema.** 3](#_Toc194059951)

[**Diseño del programa** 3](#_Toc194059952)

[**Lista de terminales** 5](#_Toc194059953)

[Lista de no terminales 5](#_Toc194059954)

[Símbolo inicial y producciones. 7](#_Toc194059955)

[Análisis de resultados 8](#_Toc194059956)

[Lecciones aprendidas 8](#_Toc194059957)

[Objetivos alcanzados 8](#_Toc194059958)

[Objetivos no logrados. 8](#_Toc194059959)

[Bitácora 8](#_Toc194059960)

# **Descripción del problema.**

Un grupo de desarrolladores desea crear un nuevo lenguaje imperativo, ligero, que le permita realizar operaciones básicas para la configuración de chips, ya que esta es una industria que sigue creciendo constantemente, y cada vez estos chips necesitan ser configurados por lenguajes más ligeros y potentes. Es por esto que este grupo de desarrolladores requiere desarrollar su propio lenguaje para el desarrollo de sistemas empotrados, y como primer paso necesitan desarrollar una gramática simple y poderosa.

# **Diseño del programa**

|  |  |
| --- | --- |
| Requisito | Descripción |
| Funciones y estructuras de control | Permite la creación de funciones, que pueden contener estructuras de control, bloques de código (\ y /), y sentencias de código. |
| Tipos de variables | Soporta enteros, flotantes, booleanos, caracteres, cadenas de caracteres (string) y arreglos estáticos bidimensionales. |
| Arreglos | Se permite la creación de arreglos de tipo entero, con acceso y modificación de elementos. Se usan ` |
| Sentencias | Permite la creación de variables, asignación de expresiones y ejecución de expresiones sin asignación en algunos casos. |
| Expresiones | Soporta la combinación de literales, variables y funciones dentro de los tipos reconocidos. |
| Operadores y precedencia | Admite operadores con precedencia matemática estándar y uso de paréntesis para agrupar expresiones. |
| Operaciones aritméticas binarias | Soporta suma (+), resta (-), división (// - entera o decimal según el tipo), multiplicación (\*), módulo (~) y potencia (\*\*). |
| Operaciones aritméticas unarias | Soporta - (negativo para enteros y flotantes), ++ y -- (para enteros y flotantes, antes del operando). |
| Expresiones relacionales | Permite menor (<), menor o igual (<=), mayor (>), mayor o igual (>=), igual (==) y diferente (!=). == y != también aplican a booleanos. |
| Expresiones lógicas | Conjunción (^), disyunción (#) y negación (!, de tipo carácter). |
| Sentencias de código y delimitadores | Todas las expresiones y combinaciones de expresiones deben finalizar con ?. Pueden usarse en estructuras de control. |
| Reglas gramaticales y tipado | El lenguaje sigue las reglas del paradigma imperativo, con tipado explícito y fuerte. Basado en C. |
| Tipado | El lenguaje tiene tipado explícito y fuerte. |
| Estructuras de control | Se soportan if-[elif]-[else], do-while, for, return y break. Las condiciones deben evaluar a booleanos. |
| Entrada y salida | Funciones para leer (int y float) y escribir (string, int, boolean, float). Puede imprimir literales o variables. |
| Funciones | Se permite la creación y uso de funciones con retorno (int, float, char, boolean) y parámetros tipados. |
| Procedimiento principal | Existe un único procedimiento main de tipo void, que inicia la ejecución del programa. |
| Comentarios | Comentarios de una línea (@) y múltiples líneas ({ }). |

## **Lista de terminales**

|  |  |  |  |
| --- | --- | --- | --- |
| Numero de terminal | Terminal | Numero de terminal | Terminal |
| 1 | / | 29 | do |
| 2 | \ | 30 | while |
| 3 | int | 31 | for |
| 4 | boolean | 32 | return |
| 5 | float | 33 | break |
| 6 | char | 34 | main |
| 7 | string | 35 | void |
| 8 | / | 36 | { |
| 9 | \ | 37 | } |
| 10 | | | 38 | , |
| 12 | = |  |  |
| 13 | + |  |  |
| 14 | - |  |  |
| 15 | \* |  |  |
| 16 | / |  |  |
| 17 | ~ |  |  |
| 18 | > |  |  |
| 19 | < |  |  |
| 20 | ^ |  |  |
| 21 | # |  |  |
| 22 | ! |  |  |
| 23 | @ |  |  |
| 24 | ? |  |  |
| 25 | " |  |  |
| 26 | if |  |  |
| 27 | elif |  |  |
| 28 | else |  |  |

## Lista de no terminales

|  |  |  |
| --- | --- | --- |
| **Numero de no terminal** | **No terminal** | **Descripción** |
| 1 | main\_func | Define la función main del programa. |
| 2 | func\_decl | Declaración de una función completa |
| 3 | func\_list | Lista de funciones definidas (0 o más) |
| 4 | param\_list | Lista de parametros de una función |
| 5 | param | Parametro individual con tipo y nombre |
| 6 | block | Bloque de código delimitado por \ y / |
| 7 | stmt\_list | Lista de sentencias dentro de un bloque. |
| 8 | stmt | Sentencia general: declaración, asignación, control, etc |
| 9 | var\_decl | Declaración de variable o arreglo. |
| 10 | assign\_stmt | Asignación de valor a variable o a una celda de arreglo. |
| 11 | expr | Expresión general: aritmética, lógica, relacional, literal, llamada. |
| 12 | arith\_expr | Expresiones con operadores matemáticos. |
| 13 | rel\_expr | Comparaciones relacionales entre expresiones. |
| 14 | logic\_expr | Combinaciones lógicas de expresiones booleanas. |
| 15 | control\_struct | Agrupa todas las estructuras de control. |
| 16 | if\_stmt | Condicional con if, elif, y else. |
| 17 | do\_while\_stmt | Estructura do-while |
| 18 | for\_stmt | Ciclo for con inicialización, condición y actualización. |
| 19 | return\_stmt | Retorno de valor desde una función. |
| 20 | break\_stmt | Sentencia de ruptura de ciclo (break). |
| 21 | type | |  | | --- | |  |  |  | | --- | | Tipos de datos definidos por el lenguaje. | |
| 22 | array\_decl | Declaración de un arreglo bidimensional de enteros. |
| 23 | comment | Comentario de una línea o bloque. |
| 24 | func\_call | Llamada a función con argumentos. |
| 25 | arg\_list | Lista de argumentos separados por coma. |
| 26 | write\_stmt | Función para imprimir en pantalla |
| 27 | read\_stmt | Función para leer desde el teclado la información |

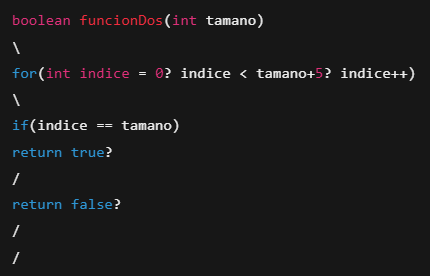
## Símbolo inicial y producciones.

* Inicio -> código ::= main\_func | func\_decl
* main\_func ::= void main ( ) block
* func\_decl ::= type id "(" param\_list ")" block
* func\_list ::= func\_decl func\_list | ε
* param\_list ::= param "," param\_list | param | ε
* param ::= type id
* block ::= "\\" stmt\_list "/"
* stmt\_list ::= stmt stmt\_list | ε
* stmt ::= var\_decl "?" | assign\_stmt "?" | control\_struct | read\_stmt "?" | write\_stmt "?" | return\_stmt "?" | break\_stmt "?" | comment | var\_decl\_asisgn\_stmt
* var\_decl ::= type id | array\_decl
* array\_decl ::= | int id[ int\_literal ] [ int\_literal ] | ε
* assign\_stmt ::= id "=" expr | id "[" expr "]" "[" expr "]" "=" expr | id++ | id-- | ε
* var\_decl\_asisgn\_stmt ::= var\_decl "=" expr
* expr ::= arith\_expr | rel\_expr | logic\_expr | literal | func\_call | id
* arith\_expr ::= expr "+" expr | expr "-" expr | expr "//" expr | expr "\*" expr | expr "\*\*" expr | expr "~" expr | -expr | ε
* rel\_expr ::= expr " <" expr | expr " <=" expr | expr " >" expr | expr " >=" expr | expr " ==" expr | expr " !=" expr | ε
* logic\_expr ::= " (" expr "^" expr ")" | " (" expr "#" expr ")" | !expr
* literal ::= int\_literal | float\_literal | boolean\_literal | string\_literal | char\_literal
* int\_literal ::= -?( 0 | [1-9][0-9]\*)
* float\_literal ::= -?( 0 | [1-9][0-9]\*\.[0-9]\*[1-9])
* boolean\_literal ::= true | false
* string\_literal ::= \".\*\"
* char\_literal ::= \'.\ '
* control\_struct ::= if\_elif\_else\_stmt | if\_elif\_stmt | if\_stmt | do\_while\_stmt | for\_stmt
* if\_elif\_else\_stmt ::= "if" "(" expr ")" block elif\_blocks "else" block
* if\_elif\_stmt ::= "if" "(" expr ")" block elif\_block
* if\_stmt ::= "if" "(" expr ")" block
* elif\_blocks ::= elif\_block elif\_blocks | elif\_block | ε
* elif\_block ::= "elif" "(" expr ")" block
* do\_while\_stmt ::= "do" block "while" "(" expr ")" "?"
* for\_stmt ::= "for" "(" assign\_stmt "?" expr "?" assign\_stmt ")" block
* return\_stmt ::= "return" expr
* break\_stmt ::= "break"
* type ::= "int" | "float" | "boolean " | "char " | "string"
* comment ::= "@" text | "{" text "}"
* text ::= .\*
* func\_call ::= identifier "(" arg\_list ")"
* arg\_list ::= expr "," arg\_list | expr | ε
* write\_stmt::= "write" "(" ( id | int\_literal | string\_literal | float\_literal ) ")"
* read\_stmt ::= "readInt()" | "readFloat()"

# Análisis de resultados

A continuación, se presentan tres derivaciones que ayudan a demostrar que la gramática es correcta.

1. Derivación 1



1. main\_func

2. func\_decl

3. type id "(" param\_list ")" block

4. boolean funcionDos("(" param\_list ")" block)

5. boolean funcionDos("(" param ")" block)

6. boolean funcionDos("(" type id ")" block)

7. boolean funcionDos("(int tamano)" block)

8. boolean funcionDos("(int tamano) \ stmt\_list /")

9. boolean funcionDos("(int tamano) \ stmt stmt\_list /")

10. boolean funcionDos("(int tamano) \ control\_struct stmt\_list /")

11. boolean funcionDos("(int tamano) \ for\_stmt stmt\_list /")

12. boolean funcionDos("(int tamano) \ for (assign\_stmt ? expr ? assign\_stmt) block stmt\_list /")

13. boolean funcionDos("(int tamano) \ for (int indice = 0? expr ? assign\_stmt) block stmt\_list /")

En esta derivación se logra apreciar los siguientes aspectos:

14. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? assign\_stmt) block stmt\_list /")

15. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) block stmt\_list /")

16. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ stmt\_list / stmt\_list /")

17. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ stmt stmt\_list / stmt\_list /")

18. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ control\_struct stmt\_list / stmt\_list /")

19. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if\_stmt stmt\_list / stmt\_list /")

20. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (expr) block stmt\_list / stmt\_list /")

21. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) block stmt\_list / stmt\_list /")

22. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) \ stmt\_list / stmt\_list / /")

23. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) \ stmt stmt\_list / stmt\_list / /")

24. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) \ return\_stmt stmt\_list / stmt\_list / /")

25. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) \ return true? stmt\_list / stmt\_list / /")

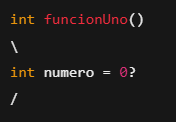
26. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) \ return true? / return\_stmt / /")

27. boolean funcionDos("(int tamano) \ for (int indice = 0? indice < tamano + 5? indice++) \ if (indice == tamano) \ return true? / return false? / /")

* Las producciones son capaces de detectar las funciones definidas.
* El for tiene se detecta de manera correcta.
* Se pueden devolver booleanos.
* Se pueden ingresar parámetros de tipo entero.
* Los bloques pueden tener estructuras de control y realizar retornos.

Todo esto se realiza según el diseño del lenguaje de programación.

1. Derivación 2



1. main\_func

2. func\_decl

3. type id "(" param\_list ")" block

4. int funcionUno("(" param\_list ")" block)

5. int funcionUno("(" ε ")" block)

6. int funcionUno("()" block)

7. int funcionUno("() \ stmt\_list /")

8. int funcionUno("() \ stmt stmt\_list /")

9. int funcionUno("() \ var\_decl stmt\_list /")

10. int funcionUno("() \ type id stmt\_list /")

11. int funcionUno("() \ int id stmt\_list /")

12. int funcionUno("() \ int numero stmt\_list /")

13. int funcionUno("() \ int numero = expr stmt\_list /")

14. int funcionUno("() \ int numero = int\_literal stmt\_list /")

15. int funcionUno("() \ int numero = 0 stmt\_list /")

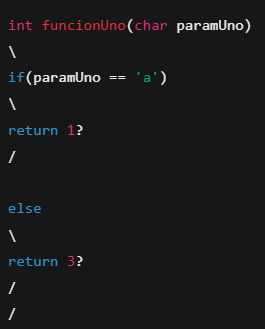
16. int funcionUno("() \ int numero = 0? stmt\_list /")

17. int funcionUno("() \ int numero = 0? /")

En esta derivación se puede observar lo siguiente:

* Se puede crear y asignar variables.
* Se pueden escribir funciones sin estructuras de control.
* Se pueden escribir cero parámetros.

1. Derivación 3



1. main\_func

2. func\_decl

3. type id "(" param\_list ")" block

4. int funcionUno("(" param\_list ")" block)

5. int funcionUno("(" param ")" block)

6. int funcionUno("(" type id ")" block)

7. int funcionUno("(" char paramUno ")" block)

8. int funcionUno("(" char paramUno ") \ stmt\_list /")

9. int funcionUno("(" char paramUno ") \ stmt stmt\_list /")

10. int funcionUno("(" char paramUno ") \ control\_struct stmt\_list /")

11. int funcionUno("(" char paramUno ") \ if\_stmt stmt\_list /")

12. int funcionUno("(" char paramUno ") \ if (expr) block stmt\_list /")

13. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') block stmt\_list /")

A continuación, se presentan los resultados de las derivaciones:

15. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ stmt stmt\_list / stmt\_list /")

16. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return\_stmt stmt\_list / stmt\_list /")

17. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? stmt\_list / stmt\_list /")

18. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / control\_struct /")

19. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / if\_stmt /")

20. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / else block /")

21. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / else \ stmt\_list / /")

22. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / else \ stmt stmt\_list / /")

23. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / else \ return\_stmt stmt\_list / /")

24. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / else \ return 3? stmt\_list / /")

25. int funcionUno("(" char paramUno ") \ if (paramUno == 'a') \ return 1? / else \ return 3? / /")

* Se pueden realizar estructuras if-else sin la necesidad de utilizar elif, esto solo se utiliza opcionalmente cuando se necesita.
* Se pueden utilizar literales de tipo char para en el lenguaje.
* Se pueden ingresar parámetros de tipo char en las funciones.

## Lecciones aprendidas

1. Aprendimos a construir una gramática desde cero utilizando la notación BNF. Al principio parecía complejo, pero poco a poco fuimos entendiendo cómo cada regla ayuda a darle estructura al lenguaje.
2. Logramos diferenciar claramente entre terminales (como palabras clave o símbolos) y no terminales (como funciones, bloques o expresiones), lo cual fue clave para organizar bien la gramática.
3. Entendimos cómo usar la recursividad para definir listas, como por ejemplo funciones, parámetros o sentencias. Esto fue fundamental para que el lenguaje pudiera tener estructuras flexibles y reutilizables.
4. Reforzamos el concepto de tipado fuerte y explícito: cada variable o parámetro debe tener un tipo bien definido, lo que mejora la claridad y reduce errores en el lenguaje.
5. Supimos cómo representar estructuras de control como if, elif, else, do-while y for dentro de la gramática, lo cual nos permitió diseñar un lenguaje capaz de manejar decisiones y ciclos.
6. Diseñamos una forma personalizada de delimitar bloques con \ y /, lo cual nos obligó a pensar en cómo adaptar la gramática a nuestras propias decisiones de sintaxis.
7. Notamos la importancia de usar “?” al final de cada sentencia. Aunque es un detalle simple, afectó muchas partes de la gramática y nos hizo ser consistentes con las reglas del lenguaje.
8. Aplicamos expresiones regulares para definir cómo deben verse los literales, como los enteros, flotantes, booleanos, cadenas y caracteres. Esto nos ayudó a ser más precisos en el análisis léxico.
9. Valoramos el reto de diseñar un lenguaje completo desde cero. No es solo cuestión de poner palabras clave: cada elección tiene implicaciones en cómo se analiza, interpreta y ejecuta el código.

## Objetivos alcanzados y no alcanzados

|  |  |  |  |
| --- | --- | --- | --- |
| Requisito | Alcanzado | No alcanzados | Razón |
| Funciones y estructuras de control | X |  |  |
| Tipos de variables | X |  |  |
| Arreglos | X |  |  |
| Sentencias | X |  |  |
| Expresiones | X |  |  |
| Operadores y precedencia | X |  |  |
| Operaciones aritméticas binarias | X |  |  |
| Operaciones aritméticas unarias | X |  |  |
| Expresiones relacionales | X |  |  |
| Expresiones lógicas | X |  |  |
| Sentencias de código y delimitadores | X |  |  |
| Reglas gramaticales y tipado | X |  |  |
| Tipado | X |  |  |
| Estructuras de control | X |  |  |
| Entrada y salida | X |  |  |
| Funciones | X |  |  |
| Procedimiento principal | X |  |  |
| Comentarios | X |  |  |

# Bitácora

<https://github.com/AndrewLopezHerrera/TC1_HillaryMalespinUlloa_-AndrewLopezHerrera>