Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza bajaForma

**Instituto Tecnológico de Costa Rica**

**Compiladores e Intérpretes**

**Tarea 1: Gramática BNF**

**Profesor:**

**Allan Rodríguez Dávila**

**Estudiantes:**

**Jimena Méndez Morales - 2023113347**

**Ricardo Arce Aguilar - 2023215990**

**I Semestre, 2025**

## Descripción del problema

El objetivo de esta tarea es diseñar una gramática en notación BNF que defina un lenguaje de programación imperativo ligero. Este lenguaje debe soportar variables globales y locales, funciones, expresiones aritméticas, relacionales y lógicas, así como estructuras de control. El diseño busca que el lenguaje sea adecuado para la configuración de sistemas embebidos, manteniendo sintaxis clara y tipado fuerte.

## Diseño del programa

### Lista de terminales

|  |  |
| --- | --- |
| **Categoría** | **Símbolo o palabra reservada** |
| Palabras reservadas | |  | | --- | |  |  |  | | --- | | let, main, decide of, else, end decide, loop, exit when, end loop, for, step, to, downto, do, return, break, input, output | |
| Operadores aritméticos | +, -, \*, /, //, %, ^, ++, -- |
| Operadores relacionales | <, <=, >, >=, ==, != |
| Operadores lógicos | @ (AND), ~ (OR), Σ (NOT) |
| Delimitadores y símbolos | Є, э , $, [, ], ,, =, $ |
| Literales | |  | | --- | |  |  |  | | --- | | Números enteros (0,1,2…), flotantes (3.14), booleanos (true, false), caracteres ('a'), cadenas ("texto") | |
| Comentarios | | , ¡ , ! |

### Lista de no terminales

|  |  |
| --- | --- |
| **No terminal** | **Descripción** |
| program | Unidad completa del programa. Puede contener variables globales, la función principal y funciones auxiliares. |
| global\_vars | Conjunto de variables globales. Puede ser una secuencia de variables o vacío. |
| global\_var | Creación de una variable global con su tipo, identificador, opcionalmente arreglo e inicialización. |
| functions | Conjunto de funciones definidas por el usuario. Puede ser una sola o varias en secuencia. |
| function | Definición de una función con tipo de retorno, nombre, parámetros y bloque de instrucciones. |
| main | Definición del punto de entrada del programa (main) con un bloque de instrucciones. |
| params | Lista de parámetros de una función. Puede ser vacía o una secuencia separada por comas. |
| param | Un parámetro individual: tipo e identificador, opcionalmente un arreglo. |
| type | Tipos de datos básicos admitidos (int, float, char, bool, string). |
| opt\_array\_decl | Especifica si una **variable** es un arreglo ([n]) o no. |
| array\_param | Especifica si un parámetro de función es un arreglo ([n]) o no. |
| init | Inicialización de una variable (= expr). |
| var\_creation | Declaración de variables locales dentro de un bloque, con tipo, id, y opcionalmente arreglo e inicialización. |
| var\_asignment | Sentencia de asignación, ya sea simple, acceso a un elemento de arreglo o llamada a función. |
| function\_call | Invocación de una función con argumentos. |
| params | Lista de argumentos en una llamada a función, puede estar vacía. |
| control\_structure | Conjunto de estructuras de control de flujo: decideof, loop, for. |
| decide | Estructura condicional múltiple (similar a switch o case). |
| loop | Estructura de repetición loop ... exit when ... end loop. |
| for | Ciclo for con inicialización, paso, dirección (to/downto) y bloque. |
| expr | Expresiones, con precedencia definida (lógica, relacional, aritmética, unarios, primarios). |
| sentence | Una instrucción individual: declaración, asignación o estructura de control. |
| code\_block | Secuencia de instrucciones, que puede ser vacía. |
| int | Constante entera. |
| float | Constante numérica con decimales. |
| bool | Literal booleano (true o false). |
| string | Literal de cadena delimitada por comillas dobles. |
| char | Literal de carácter delimitado por comillas simples. |
| id | Identificador para variables, funciones o parámetros. |
| primary\_expr | Expresión primaria indivisible con significado propio. |
| inc\_dec\_expr | Operación de incremento o decremento. |
| base\_exp\_expr | Base o exponente de una potencia. |
| power\_expr | Expresión de potencia. |
| unary\_minus | Menos unario. |
| logic\_neg | Negación lógica. |
| mult\_term | Término de una expresión de multiplicación. |
| mult\_expr | Expresión de multiplicación. |
| adit\_term | Término de una expresión de adición. |
| adit\_expr | Expresión de adición. |
| arithm\_expr | Expresión aritmética en general. |
| hp\_rel\_term | Término de una expresión relacional de alta precedencia. |
| hp\_rel\_expr | Expresión relacional de alta precedencia. |
| lp\_rel\_term | Término de una expresión relacional de alta precedencia. |
| lp\_rel\_expr | Expresión relacional de alta precedencia. |
| logic\_term | Término de una expresión lógica. |
| and\_expr | Expresión de conjunción. |
| or\_expr | Expresión de disyunción. |
| code\_block | Bloque de código con sentencias y estructuras de control. |
| sentence | Sentencia que puede ser una declaración, asignación o llamada a función. |
| inputfn | Función de ingreso de datos por consola. |
| outputfn | Función de salida de datos por consola. |
| decide | Estructura de control similar al if de otros lenguajes. |
| loop | Estructura de control similar al while de otros lenguajes. |
| for | Estructura for particular de estilo Pascal. |
| oneline\_comment | Comentario de una sola línea. |
| multiline\_comment | Comentario de múltiples líneas. |

## Gramática

### Símbolo inicial

program

### Producciones

Para evitar confusiones, siempre que se use un operador “|” dentro de paréntesis, se refiere al operador “|” de regex, en caso contrario se refiere al operador de la notación BNF. Se hace esta aclaración porque ambos operadores, aunque usan el mismo símbolo, difieren ligeramente en su funcionamiento.

program ::= global\_vars main functions

program ::= global\_vars functions main

program ::= global\_vars functions main functions

global\_vars ::= global\_var global\_vars | ε

global\_var ::= "let" type id opt\_array\_decl (init|ε) $

type ::= "int" | "float" | "bool" | "char" | "string"

id ::= [a-zA-Z\_]([a-zA-Z\_]|[0-9])\*

// opt\_array\_decl es para declarar arreglos, como “let arr[1] = ... $”

opt\_array\_decl ::= "[" int "]" | ε

init ::= "=" expr

// Precedencia (menor a mayor): ~ (OR) >> @ (AND) >> ==,!=

// >> >,>=,<,<= >> +,- >> \*,/,//,% >> ^ >> -,Σ (menos unario

// y negación lógica) >> ++,-- >> ().

expr ::= primary\_expr | arithm\_expr | hp\_rel\_expr | lp\_rel\_expr |

expr ::= and\_expr | or\_expr

// Expresión primaria (elementos indivisibles).

primary\_expr ::= id | int | float | bool | char | string |

primary\_expr ::= function\_call | id "[" int "]"

int ::= [1-9][0-9]\*|0

float ::= int "." [0-9]\*[1-9]

bool ::= "true" | "false"

char ::= "'" ([^\\]]|\['"ntr]) "'"

// char acepta, entre comillas simples, cualquier carácter que no sea una barra invertida o alguno de los caracteres \', \", \n, \t o \r.

string ::= "\"" (char)\* "\""

// Expresión de incremento o decremento.

inc\_dec\_expr ::= (id|int|float)("++"|"--")

// Expresión de potencia. Está aparte porque tiene mayor precedencia.

base\_exp\_expr ::= id | int | float | inc\_dec\_expr

power\_expr ::= base\_exp\_expr "^" base\_exp\_expr

// Expresiones unarias.

unary\_minus ::= "-" (id|int|float|power\_expr)

unary\_minus ::= "-" ɛ (id|int|float|power\_expr) ɜ

logic\_neg ::= "Σ" (id|bool)

logic\_neg ::= "Σ" ɛ (id|bool) ɜ

// Multiplicación, división (entera o decimal) y módulo.

mult\_term ::= id | int | float | power\_expr | unary\_minus

mult\_expr ::= mult\_term (\\*|/|//|%) mult\_term

mult\_expr ::= mult\_expr (\\*|/|//|%) mult\_term

mult\_expr ::= ɛ mult\_expr ɜ

// Suma y resta.

adit\_term ::= id | int | float | mult\_expr | power\_expr | unary\_minus

adit\_expr ::= adit\_term (\+|-) adit\_term

adit\_expr ::= adit\_expr (\+|-) adit\_term

adit\_expr ::= ɛ adit\_expr ɜ

// Expresión aritmética general.

arithm\_expr ::= adit\_expr | mult\_expr | power\_expr | unary\_minus

arithm\_expr ::= ɛ arithm\_expr ɜ

// Operadores relacionales de precedencia alta (>, >=, <, <=).

hp\_rel\_term ::= id | int | float | arithm\_expr

hp\_rel\_expr ::= hp\_rel\_term (>|>=|<|<=) hp\_rel\_term

hp\_rel\_expr ::= ɛ hp\_rel\_expr ɜ

// Operadores relacionales de precedencia baja (==, !=).

lp\_rel\_term ::= id | int | float | bool | arithm\_expr | hp\_rel\_term

lp\_rel\_expr ::= lp\_rel\_term (==|!=) lp\_rel\_term

lp\_rel\_expr ::= ɛ lp\_rel\_expr ɜ

// Expresiones AND y OR lógicas (AND lógico). El AND y el OR van

// aparte porque tienen niveles de precedencia diferentes.

logic\_term ::= id | bool | hp\_rel\_expr | lp\_rel\_expr

and\_expr ::= logic\_term "@" logic\_term

and\_expr ::= and\_expr "@" logic\_term

and\_expr ::= ɛ and \_expr ɜ

or\_expr ::= (logic\_term|and\_expr) "~" (logic\_term|and\_expr)

or\_expr ::= or\_expr "~" (logic\_term|and\_expr)

or\_expr ::= ɛ or\_expr ɜ

functions ::= function

functions ::= function functions

function ::= type id ɛ params ɜ ¿ code\_block ?

params ::= param

params ::= param "," params

// opt\_array\_param es para parámetros como “int arr[10]”.

param ::= type id opt\_array\_param

opt\_array\_param ::= "[" int "]" | ε

code\_block ::= code\_block\_body opt\_code\_block\_return

code\_block\_body ::= ε // Una función puede no tener instrucciones.

code\_block\_body ::= sentence | control\_structure

code\_block\_body ::= (sentence | control\_structure) code\_block\_body

sentence ::= var\_creation | var\_asignment | function\_call $

sentence ::= inputfn $ | outputfn $

var\_creation ::= "let" type id opt\_array\_decl (init|ε) $

var\_asignment ::= id opt\_array\_decl init $

function\_call ::= id ɛ params ɜ

inputfn ::= "let" ("int"|"float") id "=" "input" ɛ ɜ $

outputfn ::= "output" ɛ (id|char|string|int|bool|float) ɜ $

control\_structure ::= decide | loop | for

decide ::= "decide of" ( є expr э "->" ¿ code\_block ? )\*

( "else" "->" ¿ code\_block ? )? "end decide" $

loop ::= "loop" ¿ code\_block ? "exit when" expr $ "end loop" $

for ::= "for" var\_asignment "step" int ("to"|"downto") expr "do" ¿ code\_block ?

opt\_code\_block\_return ::= "return" expr $

main ::= "main" є э ¿ code\_block ?

// Lee cualquier cosa que no sea un salto de línea hasta que aparezca

// un salto de línea.

oneline\_comment ::= "|" ([^\\n]\*([\\n)](file:///\\n)))

// Lee cualquier cosa que no sea un signo de exclamación de cierre

// hasta que aparezca uno.

multiline\_comment ::= "¡" ([^!]\*) "!"

## Análisis de resultados

### Lecciones aprendidas

* Se logró cubrir las principales características de un lenguaje imperativo (tipos, expresiones, funciones y control de flujo).
* Como mejora futura, se podría detallar aún más la semántica de operadores y la gestión de arrays.
* El objetivo principal de tener un lenguaje imperativo ligero se cumplió.

### Objetivos

|  |  |  |
| --- | --- | --- |
| **Objetivo** | **Alcanzado** | **No alcanzado** |
| Diseñar una gramática en notación BNF |  |  |
| Permitir la creación de funciones y un procedimiento principal (main) |  |  |
| Incorporar estructuras de control (decide of, loop, for) |  |  |
| Definir bloques de código con ¿ y ? y expresiones agrupadas con є y э |  |  |
| Soportar variables globales y locales |  |  |
| Soportar variables globales y locales |  |  |
| Definir expresiones aritméticas, relacionales y lógicas |  |  |
| Incluir sentencias de entrada y salida |  |  |
| Permitir comentarios de una y varias líneas |  |  |
| Asegurar que la gramática pueda generar programas válidos |  |  |

## Bitácora

<https://github.com/jimendezm/tarea1-ce>