TEC | Tecnológico de Costa Rica

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, Ɔ , $, [,], ,, =, $
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 ::= "-" ε (id|int|float|power_expr) з
logic_neg ::= "Σ" (id|bool)
logic_neg ::= "Σ" ε (id|bool) з

// 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 ::= ε mult_expr з

// 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 ::= ε adit_expr з

// Expresión aritmética general.
arithm_expr ::= adit_expr | mult_expr | power_expr | unary_minus
arithm_expr ::= ε arithm_expr з

// 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 ::= ε hp_rel_expr з

// 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 ::= ε lp_rel_expr з

// 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 ::= ε and _expr з
or_expr ::= (logic_term|and_expr) "~" (logic_term|and_expr)
or_expr ::= or_expr "~" (logic_term|and_expr)
or_expr ::= ε or_expr з

functions ::= function
functions ::= function functions
function ::= type id ε params з ¿ 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 ε params з
inputfn ::= "let" ("int"|"float") id "=" "input" ε з $
outputfn ::= "output" ε (id|char|string|int|bool|float) з $
control_structure ::= decide | loop | for
```

```
decide ::= "decide of" ( ϵ expr ϶ "->" ¿ 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" ϵ ϶ ¿ code_block ?

// Lee cualquier cosa que no sea un salto de línea hasta que aparezca
// un salto de línea.
oneline_comment ::= "|" ([^\\n]*(\\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 ɜ		
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