Proyecto II: Parser
Diseño de Compiladores
Paula Sofía Soto Ayala
A01620155

Índice

1. Introducción

1.1. Resumen

Este informe describe el diseño e implementación de un analizador sintáctico de descenso recursivo(recursive descent) para el lenguaje de programación Pascal–. El analizador verifica la estructura gramatical y realiza análisis semántico para garantizar que los programas escritos en Pascal-- sean correctos y coherentes.

1.2. Notación

1.2.1. Gramáticas Libres de Contexto(CFG)

Las Gramáticas Libres de Contexto son un formalismo matemático utilizado para describir la estructura sintáctica de los lenguajes de programación. Se componen de terminales, no terminales, reglas de producción y símbolos especiales como el inicio y fin de la cadena.

1.2.2. Modelo y Justificación

- ★ **Descripción General:**
  - ○ El analizador sintáctico se encargará de verificar la estructura gramatical de los programas escritos en Pascal–.
  - ○ El enfoque se centrará en el análisis de las reglas sintácticas del lenguaje.
- ★ **Requisitos Funcionales:**
  - ○ Identificar las construcciones sintácticas específicas de Pascal-- (declaraciones, expresiones, estructuras de control, etc.).
  - ○ Definir las reglas de precedencia y asociatividad para las operaciones aritméticas y lógicas.
  - ○ El analizador debe ser capaz de procesar programas escritos en Pascal-- y verificar su estructura gramatical.
  - ○ Debe reconocer declaraciones de variables, expresiones, estructuras de control (como if-else, for-do, repeat-until y funciones como writeLn, readLn) y manejarlas correctamente.
  - ○ El analizador debe generar un árbol abstracto de sintaxis válido para programas válidos y proporcionar mensajes de error para programas inválidos.
  - ○ Detectar errores de sintaxis cuando la estructura del programa sea errónea con el token esperado y el token recibido.
- ★ **Diseño y Funcionamiento:**

- El analizador de descenso recursivo se basa en funciones recursivas que corresponden a las reglas de la gramática.
- Cada función analiza un no terminal y decide qué producción aplicar según el siguiente token en la entrada.

★ **Interfaz de Entrada:**
- Resultado de Scanner.py (tabla de símbolos y stream de tokens)

★ **Interfaz de Salida:**
- Mensajes de error y de aceptación en consola.
- Árbol abstracto de sintaxis
- Tabla de símbolos actualizada

1.2.3. Python

Se optó por el lenguaje de programación Python para la implementación debido a su flexibilidad, amplia comunidad de desarrolladores y que es un lenguaje de programación con el que estoy familiarizada. Los tipos dinámicos en Python permitieron crear una implementación flexible y el ir añadiendo gradualmente tipos para mantener más seguridad en las operaciones realizadas.

2. Análisis

2.1. Gramática Inicial

```
Unset
1. start → program ID ; declaration_block main_block

2. declaration_block → vars_block functions_block procedures_block

3. main_block → compound_statement •

4. vars_block → var var_declaration | ε

5.  var_declaration  →  var_declaration  var_list  :  type_specifier  ;  |

var_list : type_specifier ;

6. var_list → var_list , ID | ID

7. type_specifier → basic_type | array_type

8. basic_type → integer | real | string
```

```
9. array_type → array [ NUMBER • • NUMBER ] of basic_type

10. functions_block → functions_block function_declaration | ε

11. function_declaration → function ID ( params ) : type_specifier ;

local_declarations compound_stmt ;

12. procedures_block → procedures_block procedure_declaration | ε

13. procedure_declaration → procedure ID ( params ) ; local_declarations

compound_stmt ;

14. params → param_list | ε

15. param_list → param_list var_list : type_specifier ; | var_list :

type_specifier ;

16. local_declarations → vars_block | ε

17. compound_stmt → begin statement_list end

18. statement_list → statement_list statement | statement

19. statement → assignment_stmt | call _stmt | compound_stmt |

selection_stmt

| for_stmt | repeat_stmt | input_stmt | output_stmt

20. assignment_stmt → var := arithmetic_expression ; | var := STRING ;

21. call_stmt → call

22. selection_stmt → if ( logic_expression ) then statement ;

| if ( logic_expression ) then statement else statement ;

23. repeat_stmt → repeat statement_list until ( logic_expression )

24. for_stmt → for ID := NUMBER to NUMBER do statement ;

25. input_stmt → readln ( var_list )
```

```
26. output_stmt → writeln ( output_list )

27. output_list → output_list , output | output

28. output → arithmetic_expression | STRING

29. var → ID | ID [ arithmetic_expression ]

30. logic_expression → arithmetic_expression relop arithmetic_expression
31. relop → <= | < | > | >= | == | !=
32.  arithmetic  _expression  →  arithmetic_expression  arithmetic_operator
arithmetic_expression
| (arithmetic_expression ) | var | call | NUMBER
33. arithmetic_operator → + | - | * | /
34. call → ID ( args ) ;
35. args → arg_list | ε
36. arg_list → arg_list , arithmetic_expression | arithmetic_expression
```

2.2. Retirar ambigüedad

2.2.1. Ambigüedad en la regla 9, 24 y 32

★ En lugar de usar "NUMBER", especificaremos si se trata de un entero o un real. Así que actualizaremos la regla 9, 24 y 32 como sigue:

```
Unset
9. array_type → array [ NUMBER .. NUMBER ] of basic_type

24. for_stmt → for ID := NUMBER to NUMBER do statement ;

32.  arithmetic  _expression  →  arithmetic_expression  arithmetic_operator
arithmetic_expression


                                    →


9. array_type → array [ int_number .. int_number ] of basic_type
```

```
24. for_stmt → for ID := int_number to int_number do statement

34.  factor → ( arithmetic_expression ) | var | call | int_number  |
real_number
```

2.2.2. Eliminación de punto y coma redundante

★ Dado que `statement_list` ya contiene punto y coma, eliminaremos los puntos y coma innecesarios en las siguientes reglas:

```
Unset
22. selection_stmt → if ( logic_expression ) then statement ;
| if ( logic_expression ) then statement else statement ;

23. repeat_stmt → repeat statement_list until ( logic_expression )

24. for_stmt → for ID := NUMBER to NUMBER do statement ;


                              →


22. selection_stmt → if ( logic_expression ) then statement | if (
logic_expression ) then statement else statement

23. repeat_stmt → repeat statement_list until ( logic_expression )

24. for_stmt → for ID := int_number to int_number do statement
```

2.2.3. Manejo de WriteLn vacío

★ Para evitar que `writeln` falle cuando está vacío, modificaremos la regla 28:

```
Unset
28. output → arithmetic_expression | STRING

                                        →

28. output → arithmetic_expression | STRING | ε
```

2.2.4. Regla de precedencia para expresiones aritméticas

★ Eliminaremos la ambigüedad en las expresiones aritméticas definiendo una regla de precedencia. Así que actualizaremos la regla 33:

```
Unset
33. arithmetic_operator → + | - | * | /

                                        →

32. arithmetic_expression → term | term + term | term - term

33. term → factor | factor * factor | factor / factor

34. factor → ( arithmetic_expression ) | var | call | int_number |
real_number
```

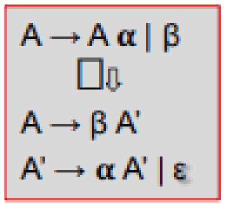
2.2.5. Gramática sin ambigüedades:

```
Unset

1. start → program ID ; declaration_block main_block

2. declaration_block → vars_block functions_block procedures_block

3. main_block → compound_statement .
```

```
4. vars_block → var var_declaration | ε

5.  var_declaration  →  var_declaration  var_list  :  type_specifier  ;  |
var_list : type_specifier ;

6. var_list → var_list , ID | ID

7. type_specifier → basic_type | array_type

8. basic_type → integer | real | string

9. array_type → array [ int_number • • int_number ] of basic_type

10. functions_block → functions_block function_declaration | ε

11.  function_declaration  →  function  ID(  params  )  :  type_specifier  ;
local_declarations   compound_stmt ;

12. procedures_block → procedures_block procedure_declaration | ε

13. procedure_declaration → procedure ID( params ) ; local_declarations
compound_stmt ;

14. params → param_list | ε

15.  param_list  →  param_list  var_list  :  type_specifier  ;  |  var_list  :
type_specifier ;

16. local_declarations → vars_block | ε

17. compound_stmt → begin statement_list end

18. statement_list → statement_list statement ; | statement ;

19.  statement  →  assignment_stmt  |  call  _stmt  |  compound_stmt  |
selection_stmt | for_stmt | repeat_stmt | input_stmt | output_stmt

20. assignment_stmt → var := arithmetic_expression | var := STRING

21. call_stmt → call

22.  selection_stmt  →  if  (  logic_expression  )  then  statement  |  if  (
logic_expression ) then statement else statement
```

```
23. repeat_stmt → repeat statement_list until ( logic_expression )

24. for_stmt → for ID := int_number to int_number do statement

25. input_stmt → readln ( var_list )

26. output_stmt → writeln ( output_list )

27. output_list → output_list , output | output

28. output → arithmetic_expression | STRING | ε

29. var → ID | ID [ arithmetic_expression ]

30. logic_expression → arithmetic_expression relop arithmetic_expression

31. relop → <= | < | > | >= | = | <>

32. arithmetic_expression → term | arithmetic_expression + term |
arithmetic_expression − term

33. term → factor | term * factor | term / factor

34. factor → ( arithmetic_expression ) | var | call | int_number |
real_number

35. call → ID ( args )

36. args → arg_list | ε

37. arg_list → arg_list , arithmetic_expression | arithmetic_expression
```

2.3. Left-recursion-free



$$A \rightarrow A\alpha \mid \beta$$
$$\Downarrow$$
$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Figura 2.3.1

2.3.1. Var Declaration

★ La regla var_declaration se reescribe como:

```
Unset
5.  var_declaration  →  var_declaration  var_list  :  type_specifier  ;  |
var_list : type_specifier ;


A'=  var_declaration

α =   var_list : type_specifier ;

β=  var_list : type_specifier ;


var_declaration → var_list : type_specifier ; var_declaration'

var_declaration' →  var_list : type_specifier ; var_declaration' |  ε
```

2.3.2. Var List

★ La regla var_list se reescribe como:

```
Unset
6. var_list → var_list , ID | ID


A'=  var_list

α =   , ID

β=  ID


var_list → ID var_list'

var_list' →  , ID var_list' |ε
```

2.3.3. Functions Block

★ La regla functions_block se reescribe como:

```
Unset
10. functions_block → functions_block function_declaration | ε


A'=  functions_block

α =   function_declaration

β=  ε


functions_block →   functions_block '

functions_block ' →  function_declaration   functions_block ' |ε
```

2.3.4. Procedures Block

★ La regla procedures_block se reescribe como:

```
Unset
12. procedures_block → procedures_block procedure_declaration | ε


A'= procedures_block

α = procedure_declaration

β=  ε


procedures_block→  ε procedures_block'

procedures_block' →  procedure_declaration   procedures_block' |ε
```

2.3.5. Param List

★ **La regla param_list se reescribe como:**

```
Unset
15.  param_list  →  param_list  var_list  :  type_specifier  ;  |  var_list  :
type_specifier ;


A=   param_list

α =   var_list : type_specifier ;

β=   var_list : type_specifier ;


 param_list→ var_list : type_specifier ;  param_list'

param_list' → var_list : type_specifier ;  param_list' | ε
```

2.3.6. Statement List

★ **La regla `statement_list` se reescribe como:**

```
Unset
18. statement_list → statement_list statement ; | statement ;


A'=  statement_list

α =   statement ;

β=   statement ;


 statement_list → statement ;   statement_list'

statement_list'→ statement ;   statement_list' | ε
```

2.3.7. Output List

★ **La regla `output_list` se reescribe como:**

```
Unset
27. output_list → output_list , output | output


A=   output_list

α =    , output

β=   output


output_list → , output    output_list '

output_list '→ , output   output_list ' | ε
```

2.3.8. Arithmetic Expression

★ La regla arithmetic_expression se reescribe como:

```
Unset
32.   arithmetic_expression    →   arithmetic_expression   +   term   |
arithmetic_expression - term | term


A=   arithmetic_expression

a1=  + term

a2= - term

b= term


arithmetic_expression → term   arithmetic_expression '

arithmetic_expression  '  →  +  term  arithmetic_expression  '  |  -  term
arithmetic_expression ' | ε
```

2.3.9. Term

★ La regla term se reescribe como:

```
Unset
33. term → term * factor | term / factor | factor


A=   term

a1=  * factor

a2= / factor

b= factor


term  → factor  term'

term' → * factor   term'  | / factor  term' | ε
```

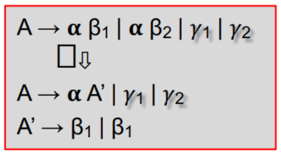
2.3.10. Gramática sin recursividad por la izquierda:

```
Unset

1. start → program ID ; declaration_block main_block

2. declaration_block → vars_block functions_block procedures_block

3. main_block → compound_statement •

4. vars_block → var var_declaration | ε

5. var_declaration → var_list : type_specifier ; var_declaration’

6. var_declaration’ → var_list : type_specifier ; var_declaration’ | ε

7. var_list → ID var_list’

8. var_list’ → , ID var_list’ | ε

9. type_specifier → basic_type | array_type

10. basic_type → integer | real | string

11.  array_type → array [ int_number • • int_number ] of basic_type
```

```
12.  functions_block → functions_block'

13.  functions_block' → function_declaration functions_block' | ε

14.  function_declaration → function ID( params ) : type_specifier ;
local_declarations compound_stmt ;

15.  procedures_block → procedures_block'

16.  procedures_block' → procedure_declaration procedures_block' | ε

17.  procedure_declaration → procedure ID( params ) ; local_declarations
compound_stmt ;

18.  params → param_list | ε

19.  param_list → var_list : type_specifier ; param_list'

20.  param_list' → var_list : type_specifier ; param_list' | ε

21.  local_declarations → vars_block | ε

22.  compound_stmt → begin statement_list end

23.  statement_list → statement ; statement_list'

24.  statement_list' → statement ; statement_list' | ε

25.  statement → assignment_stmt | call _stmt | compound_stmt |
selection_stmt | for_stmt | repeat_stmt | input_stmt | output_stmt

26.  assignment_stmt → var := arithmetic_expression | var := STRING

27.  call_stmt → call

28.  selection_stmt → if ( logic_expression ) then statement | if (
logic_expression ) then statement else statement

29.  repeat_stmt → repeat statement_list until ( logic_expression )

30.  for_stmt → for ID := int_number to int_number do statement ;

31.  input_stmt → readln ( var_list )
```

```
32.  output_stmt → writeln ( output_list )

33.  output_list → output output_list'

34.  output_list' → , output output_list' | ε

35.  output → arithmetic_expression | STRING | ε

36.  var → ID | ID [ arithmetic_expression ]

37.        logic_expression    →    arithmetic_expression    relop
arithmetic_expression

38.  relop → <= | < | > | >= | = | <>

39.  arithmetic_expression → term  arithmetic_expression'

40.   arithmetic_expression'  → + term arithmetic_expression'|  - term
arithmetic_expression'| ε

41.  term → factor term'

42.  term' → * factor  term' | / factor  term' | ε

43.  factor → ( arithmetic_expression ) | var | call | int_number |
real_number

44.  call → ID ( args )

45.  args → arg_list | ε

46.  arg_list → arithmetic_expression arg_list'

47.  arg_list' → , arithmetic_expression arg_list' | ε
```

2.4. Left-factor-free



$A \rightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \gamma_1 \mid \gamma_2$

$\Downarrow$

$A \rightarrow \alpha\, A' \mid \gamma_1 \mid \gamma_2$

$A' \rightarrow \beta_1 \mid \beta_1$

Figura 2.4.1

2.4.1. assignment_stmt

★ La regla `assignment_stmt` se reescribe como:

```
Unset
26. assignment_stmt → var := arithmetic_expression | var := STRING


A = assignment_stmt ;

α = var :=

β1= arithmetic_expression

β2= STRING


assignment_stmt → var :=  assignment_stmt'

assignment_stmt' → arithmetic_expression |  STRING
```

2.4.2. selection_stmt

★ La regla `selection_stmt` se reescribe como:

```
Unset
29. selection_stmt → if ( logic_expression ) then statement

| if ( logic_expression ) then statement else statement


A= selection_stmt

α = if ( logic_expression ) then statement

β1= ε

β2= else statement


selection_stmt   →   if   (   logic_expression   )   then   statement
selection_stmt'
```

```
selection_stmt' →  ε | else statement
```

2.4.3. var

- ★ La regla var se reescribe como:

```
Unset
36. var → ID | ID [ arithmetic_expression ]


A= var

α = ID

β1= ε

β2= [ arithmetic_expression ]


var → ID   var'

var' →  ε | [ arithmetic_expression ]
```

2.4.4. Simplificación

- ★ Simplificación y eliminación de funciones unitarias(son aquellas producciones en las que una variable se deriva directamente en otra variable sin ningún símbolo terminal intermedio.)

```
Unset
start → program ID ; declaration_block main_block

declaration_block → vars_block  functions_block  procedures_block

main_block → compound_statement .

compound_stmt → begin statement_list end
```

```
      start → program ID ; vars_block functions_block procedures_block
      begin statement_list end .

------------------------------------------------------------------------

vars_block → var var_declaration | ε

var_declaration → var_list : type_specifier ; var_declaration’

var_list → ID var_list’

      vars_block → var ID var_list’ : type_specifier ; var_declaration’
      | ε

------------------------------------------------------------------------

type_specifier → basic_type | array_type

basic_type → integer | real | string

array_type → array [ int_number .. int_number] of basic_type

      type_specifier → integer | real | string | array [ int_number ..
      int_number] of basic_type

------------------------------------------------------------------------

functions_block’ → function_declaration functions_block’ | ε

function_declaration  →  function  ID(  params  )  :  type_specifier  ;
local_declarations compound_stmt ;

compound_stmt → begin statement_list end

functions_block’  →   function  ID(  params  )  :  type_specifier  ;
local_declarations

begin statement_list end ; functions_block’ | ε

procedures_block’ → procedure_declaration procedures_block’ | ε

procedure_declaration  →  procedure  ID(  params  )  ;  local_declarations
compound_stmt ;
compound_stmt → begin statement_list end

      procedures_block’ → procedure ID( params ) ; local_declarations
      begin statement_list end ; procedures_block’ | ε

------------------------------------------------------------------------

params → param_list | ε

param_list → var_list : type_specifier ; param_list’

var_list → ID var_list’

      params → ID var_list’ : type_specifier ; param_list’ | ε


------------------------------------------------------------------------

statement → assignment_stmt | call _stmt | compound_stmt |
selection_stmt | for_stmt |

repeat_stmt | input_stmt | output_stmt

assignment_stmt → var := assignment_stmt’

var → ID   var'

call_stmt → call

call → ID ( args )

compound_stmt → begin statement_list end

selection_stmt → if ( logic_expression ) then statement selection_stmt’

for_stmt → for ID := int_number to int_number do statement

repeat_stmt → repeat statement_list until ( logic_expression )

input_stmt → readln ( var_list )

output_stmt → writeln ( output_list )

output_list → output output_list’
```

```
      statement → ID var' := assignment_stmt' | ID ( args ) | begin
      statement_list end | if ( logic_expression ) then statement
      selection_stmt' | for ID := NUMBER to NUMBER do statement | repeat
      statement_list until ( logic_expression )| readln ( var_list ) |
      writeln ( output output_list' )

------------------------------------------------------------------------

factor → ( arithmetic_expression ) | var | call | int_number |
real_number

var → ID   var'

call_stmt → call

call → ID ( args )

      factor → ( arithmetic_expression ) | ID   var' | ID ( args ) | |
      int_number | real_number
```

3. Diseño

3.1. Gramática Final BNF

```
Unset
1. start → program ID ; vars_block functions_block procedures_block
begin statement_list end .

2. vars_block → var ID var_list' : type_specifier ; var_declaration' | ε

3. var_declaration' → var_list : type_specifier ; var_declaration' | ε

4. var_list' → , ID var_list' | ε

5. type_specifier → integer | real | string | array [ int_number ..
int_number ] of basic_type

6. basic_type → integer | real | string
```

```
7. functions_block → functions_block’

8.  functions_block’  →  function  ID(  params  )  :  type_specifier  ;
local_declarations begin statement_list end ; functions_block’ | ε

9. procedures_block → procedures_block’

10.  procedures_block’  →  procedure  ID(  params  )  ;  local_declarations
begin statement_list end ; procedures_block’ | ε

11. params → ID var_list’ : type_specifier ; param_list’ | ε

12. param_list’ → var_list : type_specifier ; param_list’ | ε

13. local_declarations → vars_block | ε

14. statement_list → statement ; statement_list’

15. statement_list’ → statement ; statement_list’ | ε

16.  statement  →  ID  statement'  |  begin  statement_list  end|  if  (
logic_expression  )  then  statement     selection_stmt'  |  for  ID  :=
int_number to int_number do statement | repeat statement_list until (
logic_expression  )  |  readln  (  ID  var_list'  );  |  writeln  (  output
output_list') ;

17. statement' → var' :=  assignment_stmt' | ( args )

18. assignment_stmt’ → arithmetic_expression | STRING

19. selection_stmt’ → else statement | ε

20. output_list’ → , output output_list’ | ε

21. output → arithmetic_expression | STRING | ε

22. var' → [ arithmetic_expression ] | ε

23. logic_expression → arithmetic_expression relop arithmetic_expression

24. relop → <= | < | > | >= | = | <>

25. arithmetic_expression → term   arithmetic_expression’
```

```
26. arithmetic_expression’ → + term arithmetic_expression’| - term
arithmetic_expression’| ε

27. term → factor term’

28. term’ → * factor  term’ | / factor  term’ | ε

29. factor → ID factor' | int_number | real_number | (
arithmetic_operator )

30. factor' → ( args ) | var'

31.  args → arithmetic_expression arg_list’ | ε

32. arg_list’ → , arithmetic_expression arg_list’ | ε

33. var_list → ID var_list'
```

3.2. Gramática Final Pascal– EBNF

```
Unset
start = 'program', ID, ';', [vars_block], { subprog_block },
compound_stmt, '.';


vars_block = 'var', var_decl, { var_decl };

var_decl = ID, { ',', ID }, ':', type_ident, ';';


subprog_block = subprog_header, [vars_block], compound_stmt;

subprog_header = ('function' | 'procedure'), ID, '(', [param_list], ')',
type_spec, ';';

param_list = param, { ',', param };

param = ID, ':', type_spec;
```

```
stmt = (
    variable, ':=', expr |
    proc_stmt |
    compound_stmt |
    if_stmt |
    for_stmt |
    while_stmt |
    repeat_stmt
);
proc_stmt = ID, [ '(', [expr_list], ')' ];
compound_stmt= 'begin', [stmt, { ';', stmt }], 'end';
if_stmt = 'if', expr, 'then', stmt, ['else', stmt];
for_stmt = 'for', ID, ':=', int_literal, 'to', int_literal, 'do', stmt;
writeLn_stmt = 'writeLn', '(', 'expr_list', ')';
readLn_stmt = 'readLn', '(', 'expr_list', ')';

expr_list = expr, { ',', expr };
expr = cmp_expr;
cmp_expr = add_expr, [('<' | '<=' | '<>' | '>=' | '>' | '='), cmp_expr];
add_expr = mul_expr, [('+' | '-'), add_expr];
mul_expr = factor, [('*' | '/'), mul_expr];
factor = (
    ID, ['(', [expr_list], ')'] |
    int_literal |
    str_literal |
    '(', expr, ')'
);
```

```
type_spec = 'integer' | 'real' | 'string' | 'array', '[', int_literal,
'..', int_literal, ']', 'of', type_spec;


variable = ID, { '[', int_literal, ']' };

int_literal = INT | REAL;

str_literal = STRING;
```

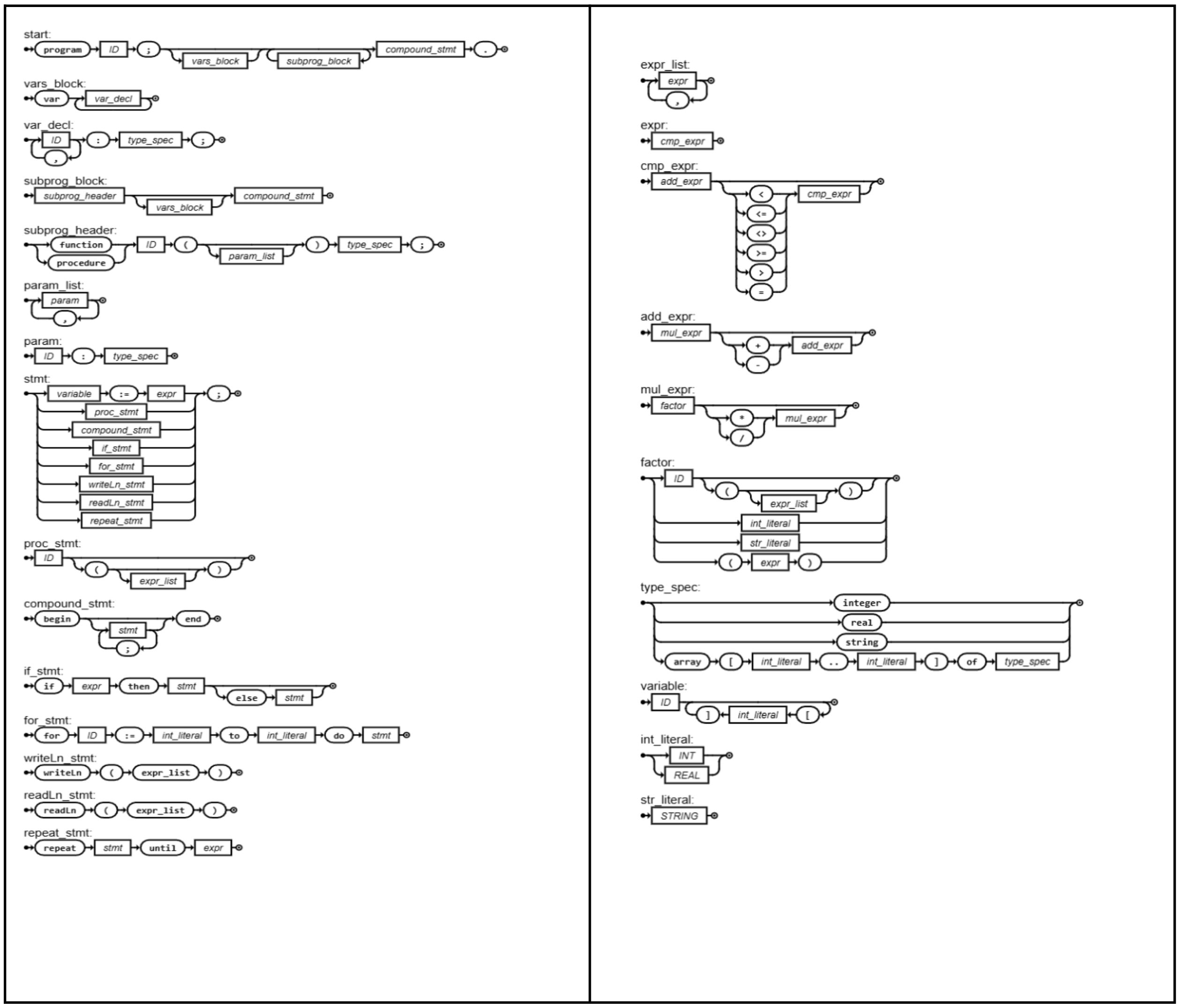
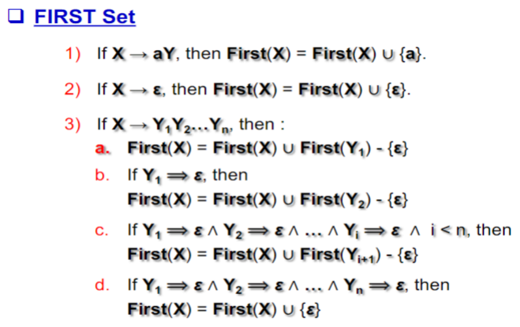

start: program ID ; vars_block subprog_block compound_stmt .
vars_block: var var_decl
var_decl: ID , : type_spec ;
subprog_block: subprog_header vars_block compound_stmt
subprog_header: function procedure ID (param_list) type_spec ;
param_list: param ,
param: ID : type_spec
stmt: variable := expr proc_stmt compound_stmt if_stmt for_stmt writeLn_stmt readLn_stmt repeat_stmt ;
proc_stmt: ID (expr_list)
compound_stmt: begin stmt ; end
if_stmt: if expr then stmt else stmt
for_stmt: for ID := int_literal to int_literal do stmt
writeLn_stmt: writeLn (expr_list)
readLn_stmt: readLn (expr_list)
repeat_stmt: repeat stmt until expr
expr_list: expr ,
expr: cmp_expr
cmp_expr: add_expr &lt; &lt;= &lt;&gt; &gt;= &gt; = cmp_expr
add_expr: mul_expr + - add_expr
mul_expr: factor * / mul_expr
factor: ID (expr_list) int_literal str_literal (expr)
type_spec: integer real string array [int_literal .. int_literal] of type_spec
variable: ID] int_literal [
int_literal: INT REAL
str_literal: STRING

Figura 3.2.1 (Diagrama de gramática EBNF)

3.3. Conjuntos de FIRST, FOLLOW y FIRST+

3.3.1. FIRST

El conjunto FIRST(X) para un símbolo no terminal X es el conjunto de todos los terminales que pueden aparecer como el primer símbolo en cualquier cadena derivada de X.



❑ FIRST Set

1) If $\mathbf{X} \rightarrow \mathbf{aY}$, then $\mathbf{First}(\mathbf{X}) = \mathbf{First}(\mathbf{X}) \cup \{\mathbf{a}\}$.
2) If $\mathbf{X} \rightarrow \varepsilon$, then $\mathbf{First}(\mathbf{X}) = \mathbf{First}(\mathbf{X}) \cup \{\varepsilon\}$.
3) If $\mathbf{X} \rightarrow \mathbf{Y_1 Y_2 \ldots Y_n}$, then :
   a. $\mathbf{First}(\mathbf{X}) = \mathbf{First}(\mathbf{X}) \cup \mathbf{First}(\mathbf{Y_1}) - \{\varepsilon\}$
   b. If $\mathbf{Y_1} \Rightarrow \varepsilon$, then
      $\mathbf{First}(\mathbf{X}) = \mathbf{First}(\mathbf{X}) \cup \mathbf{First}(\mathbf{Y_2}) - \{\varepsilon\}$
   c. If $\mathbf{Y_1} \Rightarrow \varepsilon \wedge \mathbf{Y_2} \Rightarrow \varepsilon \wedge \ldots \wedge \mathbf{Y_i} \Rightarrow \varepsilon \wedge i < n$, then
      $\mathbf{First}(\mathbf{X}) = \mathbf{First}(\mathbf{X}) \cup \mathbf{First}(\mathbf{Y_{i+1}}) - \{\varepsilon\}$
   d. If $\mathbf{Y_1} \Rightarrow \varepsilon \wedge \mathbf{Y_2} \Rightarrow \varepsilon \wedge \ldots \wedge \mathbf{Y_n} \Rightarrow \varepsilon$, then
      $\mathbf{First}(\mathbf{X}) = \mathbf{First}(\mathbf{X}) \cup \{\varepsilon\}$

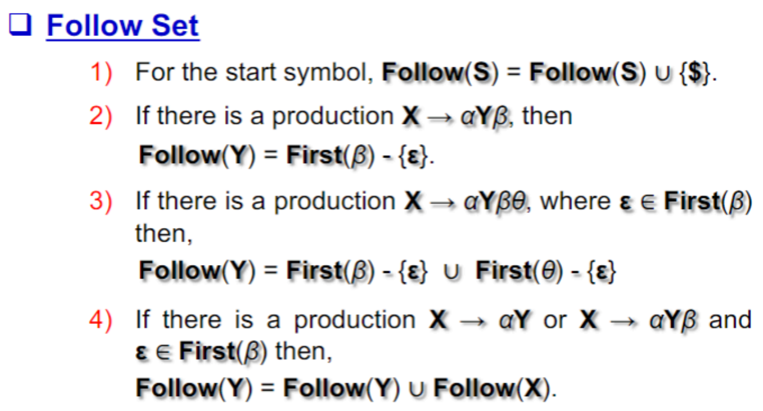
Figura 3.3.1.1

No terminal	**FIRST Set**
start	{ ‘program’ }
vars_block	{ ‘var ε’ }
var_declaration’	{ ID , ε }
var_list’	{ ‘,’, ε }
type_specifier	{ ‘integer’, ‘real’, ‘string’, ‘array’ }
basic_type	{ ‘integer’, ‘real’, ‘string’ }
functions_block	{ ‘function’ }
functions_block’	{ ‘function’, ε }
procedures_block	{ ‘procedure’ }
procedures_block’	{ ‘procedure’, ε }
params	{ ID, ε }
param_list’	{ ID, ε }
local_declarations	{ ‘var’, ε }
statement_list	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }

statement_list’	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’, ε }
statement	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }
statement’	{ ‘[’, ‘(’, ε }
assignment_stmt’	{ ID, INT, REAL, STRING, ‘(’ }
selection_stmt’	{ ‘else’, ε }
output_list’	{ ‘,’, ε }
output	{ ID, INT, REAL, STRING, ε }
var’	{ ‘[’, ε }
logic_expression	{ ID, INT, REAL, STRING, ‘(’ }
relop	{ ‘<=’, ‘<’, ‘>’, ‘>=’, ‘=’, ‘<>’ }
arithmetic_expression	{ ID, INT, REAL, STRING, ‘(’ }
arithmetic_expression’	{ ‘+’, ‘-’, ε }
term	{ ID, INT, REAL, ‘(’ }
term’	{ ‘*’, ‘/’, ε }
factor	{ ID, INT, REAL, ‘(’ }
factor’	{ ‘(’, ‘[’, ε }
args	{ ID, INT, REAL, ‘(’, ε }
arg_list’	{ ‘,’, ε }
var_list	{ ID }

3.3.2. FOLLOW

El conjunto FOLLOW(A) para un símbolo no terminal A es el conjunto de terminales que pueden aparecer inmediatamente después de A en cualquier derivación válida.



❑ Follow Set

1) For the start symbol, **Follow**(**S**) = **Follow**(**S**) ∪ {**$**}.
2) If there is a production $\mathbf{X} \rightarrow \alpha\mathbf{Y}\beta$, then
   Follow(**Y**) = **First**(β) - {**ε**}.
3) If there is a production $\mathbf{X} \rightarrow \alpha\mathbf{Y}\beta\theta$, where $\boldsymbol{\varepsilon} \in$ **First**(β) then,
   Follow(**Y**) = **First**(β) - {**ε**} ∪ **First**(θ) - {**ε**}
4) If there is a production $\mathbf{X} \rightarrow \alpha\mathbf{Y}$ or $\mathbf{X} \rightarrow \alpha\mathbf{Y}\beta$ and $\boldsymbol{\varepsilon} \in$ **First**(β) then,
   Follow(**Y**) = **Follow**(**Y**) ∪ **Follow**(**X**).

Figura 3.3.2.1

Unset
```
1. Follow(start) = {$}

2.  Follow(vars_block)  = First(functions_block) U First(procedures_block
) U Follow(local_declarations)
      = {function} U {procedure} U {begin}

3. Follow(var_declaration') = Follow(vars_block)
      = {function} U {procedure} U {begin}

4.  Follow(var_list')  = Follow(vars_block)  U Follow(var_declaration') U
Follow(params) U Follow(param_list')
      = { : } U { ) }

5. Follow(type_specifier)
      = { ; }

6. Follow(basic_type) = Follow(type_specifier)
      = { ; }

7. Follow(functions_block) = First(procedures_block) - {ε} U {begin}
      = {procedure} U {begin}

8. Follow(functions_block') = Follow(functions_block)
      = {procedure} U {begin}

9. Follow(procedures_block)
```

```
      = {begin}

10. Follow(procedures_block') = Follow (procedures_block)
      = {begin}

11. Follow(params)
      = { ) }

12. Follow(param_list') = Follow(params)
      = { ) }

13. Follow(local_declarations)
      = {begin}

14. Follow(statement_list)
      = {end} U {until}

15. Follow(statement_list') = Follow(statement_list)
      = {end} U {until}

16. Follow(statement) = { ; } U Follow(selection_stmt')
      = { ; } U {else}

17. Follow(statement') = Follow(statement)
      = { ; } U {else}

18. Follow(assignment_stmt') = Follow(statement')
      = { ; } U {else}

19.  Follow(selection_stmt')
      = { ; } U  {else}

20. Follow(output_list')
      = { ) }

21. Follow(output) = First (output_list') - {ε}
      = { , }

22. Follow(var') = { := } U Follow(factor')
      =  { := } U { * } U { / }
```

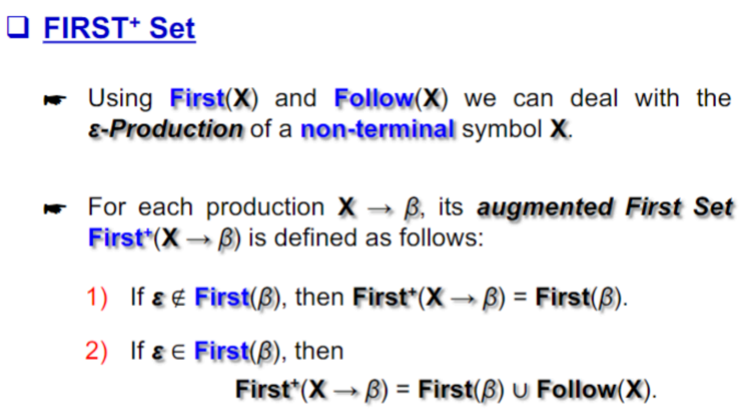
```
23. Follow(logic_expression)
      = { ) }

24. Follow(relop) = First(arithmetic_expression) - {ε}
      = { ID , int_number, real_number, ( }

25.   Follow(arithmetic_expression)   =   Follow(assignment_stmt')   U
Follow(output) U { ] } U First(relop) - {ε} U Follow(logic_expression) U
First(arg_list')   -   {ε}   U   Follow(arg)   U   Follow(arg_list’)   U
Follow(logic_expression) =
      = { ] } U { ; } U  {else} U  { , } U  { <= , < , > , >= , = , <> }
U  { , } U  { ) }
      = { ] , ; , else , , ,<= , < , > , >= , = , <>, ) }


26. Follow(arithmetic_expression’) = Follow(arithmetic_expression)
      = { ] , ; , else , , ,<= , < , > , >= , = , <>, ) }

27. Follow(term) = First(arithmetic_expression') - {ε}
      = { + , - }

28. Follow(term') = Follow(term)
      = { + , - }

29. Follow(factor) = First(term') - {ε}
      = { * } U {  / }

30. Follow(factor') = Follow(factor)
      = { * } U {  / }

31. Follow(args)
      = { ) }

32.  Follow(arg_list') = Follow(args)
      = { ) }

33. Follow(var_list) = Follow(var_declaration') U Follow(param_list')
      = { : }
```

No terminal	**FOLLOW Set**
start	{ $ }
vars_block	{ ‘function’, ‘procedure’, ‘begin’, ‘.’ }
var_declaration’	{ ‘function’, ‘procedure’, ‘begin’ }
var_list’	{ ‘:’ , ’)’ }
type_specifier	{ ‘;’ }
basic_type	{ ‘;’ }
functions_block	{ ‘procedure’, ‘begin’ }
functions_block’	{ ‘procedure’, ‘begin’ }
procedures_block	{ ‘begin’ }
procedures_block’	{ ‘begin’ }
params	{ ‘)’ }
param_list’	{ ‘)’ }
local_declarations	{ ‘begin’ }
statement_list	{ ‘end’, ‘until’ }
statement_list’	{ ‘end’, ‘until’ }
statement	{ ‘else’. ‘;’ }
statement’	{ ‘else’. ‘;’ }
assignment_stmt’	{ ‘else’. ‘;’ }
selection_stmt’	{ ‘else’. ‘;’ }
output_list’	{ ‘) ’ }
output	{ ‘ , ’ }
var’	{ ‘:=’, ‘*’, ‘/’, ‘+’, ‘-’ }
logic_expression	{ ‘) ’ }

relop	{ ID, INT, REAL , ‘ (’ }
arithmetic_expression	{ ‘else’, ‘)’, ‘<’, ‘<=’, ‘<>’, ‘>=’, ‘>’, ‘=’, ‘+’, ‘-’, ‘*’, ‘/’, ‘;’, ‘]’ }
arithmetic_expression′	{ ‘else’, ‘) ’, ‘<’, ‘<=’, ‘<>’, ‘>=’, ‘>’, ‘=’, ‘+’, ‘-’, ‘*’, ‘/’, ‘;’, ‘]’ }
term	{ ‘+’, ‘-’ }
term′	{ ‘+’, ‘-’ }
factor	{ ‘*’, ‘/’ }
factor′	{ ‘*’, ‘/’ }
args	{ ‘)’ }
arg_list ′	{ ‘)’ }
var_list	{ ‘ : ‘ }

3.3.3. FIRST+

La idea detrás de FIRST+ es extender el conjunto FIRST para incluir también los terminales que pueden aparecer después de un no terminal en una cadena derivada. En otras palabras, FIRST+ incluye tanto los terminales que pueden aparecer al principio como los que pueden aparecer después de un no terminal.



❑ **FIRST$^+$ Set**

☛ Using **First**(**X**) and **Follow**(**X**) we can deal with the ***ε-Production*** of a **non-terminal** symbol **X**.

☛ For each production **X** → β, its ***augmented First Set*** **First$^+$**(**X** → β) is defined as follows:

1) If $\varepsilon \notin$ **First**(β), then **First$^+$**(**X** → β) = **First**(β).
2) If $\varepsilon \in$ **First**(β), then
   First$^+$(**X** → β) = **First**(β) ∪ **Follow**(**X**).

Figura 3.3.4.1

```
Unset
1.  FirstPlus(start  →  program  ID  ;  vars_block  functions_block
procedures_block begin statement_list end •) = { program }


2.  FirstPlus(vars_block  →  var  ID  var_list'  :  type_specifier  ;
var_declaration') = {var}
FirstPlus (vars_block → ε ) = { function , procedure , begin , ε }

3.  FirstPlus(var_declaration'  →  ID  var_list'  :  type_specifier  ;
var_declaration') = { ID }
FirstPlus (var_declaration' → ε) = { function , procedure , begin , ε }

4. FirstPlus(var_list' → , ID var_list') = { , }
FirstPlus (var_list' → ε) = { : , ) , ε }

5.  FirstPlus(type_specifier  →  integer  |  real  |  string  |  array  [
int_number • • int_number] of basic_type) = { integer , real , string ,
array }

6. FirstPlus(basic_type → integer | real | string)=  { integer , real ,
string }

7.    FirstPlus(functions_block    →    functions_block'    )    =
First(functions_block') ={ function }

8. FirstPlus(functions_block' → function ID( params ) : type_specifier ;
local_declarations  begin statement_list end;   functions_block') = {
function }
FirstPlus (functions_block' → ε) U { ε } = { procedure , begin , ε }

9.    FirstPlus(procedures_block→    procedures_block'    )    =
First(procedures_block') = { procedure }

10.  FirstPlus(procedures_block'  →  procedure  ID(  params  )  ;
local_declarations  begin statement_list end ;  procedures_block') = {
procedure }
FirstPlus (procedures_block' → ε) U { ε } =  { begin , ε }

11. FirstPlus(params → ID var_list' : type_specifier ;  param_list') = {
ID }
```

FirstPlus (params → ε) = {) , ε }

12. FirstPlus(param_list' → ID var_list' : type_specifier ; param_list') = { ID }
FirstPlus (param_list' → ε) = {) , ε }

13. FirstPlus(local_declarations → vars_block) = first(vars_block) = { var , ε }
FirstPlus (local_declarations→ ε) ={ begin , ε }

14. FirstPlus(statement_list → statement ; statement_list') = First(statement_list) = { ID , begin , if , for , repeat , readln , writeln }

15. FirstPlus(statement_list'→ statement ; statement_list' | ε) = { ID, begin, if , for , repeat , readln , writeln} U Follow(statement_list') = { ID , begin , if , for , repeat , readln , writeln , end ,until }

16. FirstPlus(statement → ID statement' | begin statement_list end| if (logic_expression) then statement selection_stmt' | for ID := int_number to int_number do statement | repeat statement_list until (logic_expression)| readln (ID var_list'); | writeln (, output output_list');) = First(statement) = { ID, begin , if , for , repeat , readln , writeln }

17. FirstPlus(statement' → var' := assignment_stmt' | (args)) = First(var’) = { [, ε , (}

18. FirstPlus(assignment_stmt' → arithmetic_expression | STRING) = First(assignment_stmt') = { ID , real_number , int_number, (, STRING }

19. FirstPlus(selection_stmt' → else statement) = { else }
FirstPlus(selection_stmt'→ ε) = { ; }

20. FirstPlus(output_list'→ , output output_list') = { , }
FirstPlus(output_list'→ ε) ={) }

21. FirstPlus(output → arithmetic_expression | STRING | ε) = First(arithmetic_expression)= { STRING } U { ID, real_number , int_number, (}

```
FirstPlus(output → ε) = { , }

22. FirstPlus(var' → [ arithmetic_expression ]) = { [ }
Follow(var'→ ε) = { := ,  * , / , ε }

23.   FirstPlus(logic_expression   →   arithmetic_expression   relop
arithmetic_expression)   =   First(arithmetic_expression)   =   {   ID,
real_number , int_number, ( }

24. FirstPlus(relop → <= | < | > | >= | = | <> ) = {<= , < , > , >= , =
, <>}

25.  FirstPlus(arithmetic_expression  →  term  arithmetic_expression')  =
First(term) = { ID , real_number , int_number, ( }

26. FirstPlus(arithmetic_expression' → + term arithmetic_expression' | -
term arithmetic_expression') = { + , - }
FirstPlus(arithmetic_expression'→ ε) = { ] , ; , else , , ,<= , < , > ,
>= , = , <>, ) , ε }

27. FirstPlus(term → factor  term') = First(factor) = { ID , real_number
, int_number,  ( }

28. FirstPlus(term' → * factor   term'  | / factor  term') = { * , / }
FirstPlus(term'→ ε) = { + , - }

29.  FirstPlus(factor  →  ID  factor'  |  real_number  |  int_number|
(arithmetic_operator)) = { ID , real_number , int_number, ( }

30. FirstPlus( factor' → ( args ) | var' ) = { ( } U First(var’) ={ ( ,
[ , ε }


31.    FirstPlus(args    →    arithmetic_expression    arg_list')    =
First(arithmetic_expression) = { ID , real_number , int_number, ( }
FirstPlus(args→ ε) = { ) , ε }

32. FirstPlus(arg_list'→ , arithmetic_expression  arg_list ' | ε )={ , }
FirstPlus(arg_list'→ ε) = {  ) , ε }
```

```
33. FirstPlus(var_list → ID var_list') = { ID }
```

No terminal	FIRST+ Set
start	{ ‘program’ }
vars_block	{ ‘var’, ‘function’, ‘procedure’, ‘begin’, ε }
var_declaration’	{ ID, ‘function’, ‘procedure’, ‘begin’, ε }
var_list’	{ ‘)’, ‘,’, ‘:’, ε }
type_specifier	{ ‘integer’, ‘real’, ‘string’, ‘array’ }
basic_type	{ ‘integer’, ‘real’, ‘string’ }
functions_block	{ ‘function’ }
functions_block’	{ ‘function’, ‘procedure’, ‘begin’, ε }
procedures_block	{ ‘procedure’ }
procedures_block’	{ ‘procedure’, ‘begin’, ε }
params	{ ID, ‘)’, ε }
param_list’	{ ID, ‘)’, ε }
local_declarations	{ ‘var’, ‘begin’, ’ε }
statement_list	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }

statement_list’	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’, ‘end’, ‘until’ }
statement	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }
statement’	{ ‘(’, ‘[’, ε }
assignment_stmt’	{ ID, INT, REAL, STRING, ‘(’ }
selection_stmt’	{ ‘else’, ‘ ; ’ }
output_list’	{ ‘ , ’ , ‘) ’ }
output	{ ID, INT, REAL, STRING, ‘ , ’ }
var’	{ ‘[’, ‘ := ‘ , ‘ * ‘ , ‘ / ‘ , ε }
logic_expression	{ ID, INT, REAL, STRING, ‘(’ }
relop	{ ‘<=’, ‘<’, ‘>’, ‘>=’, ‘=’, ‘<>’ }
arithmetic_expression	{ ID, INT, REAL, STRING, ‘(’ }
arithmetic_expression’	{ ‘] ’, ‘ ; ’ , ‘ else ’, ‘ , ‘ , ‘<=’, ‘<’, ‘>’, ‘>=’, ‘=’, ‘<>’ , ‘)’, ε }
term	{ ID, INT, REAL, STRING, ‘(’ }
term’	{ ‘*’, ‘/’, ‘ + ’, ‘ - ’ }
factor	{ ID, INT, REAL, STRING, ‘(’ }
factor’	{ ‘(’, ‘[’, ε }
args	{ ID, INT, REAL, STRING, ‘(’, ε }
arg_list’	{ ‘ , ’ , ‘) ‘ , ε }
var_list	{ ID }

3.3.4. Tabla de Conjuntos FIRST, FOLLOW y FIRST+)

No terminal	FIRST Set	FOLLOW Set	FIRST+ Set
start	{ ‘program’ }	{ $ }	{ ‘program’ }
vars_block	{ ‘var ε’ }	{ ‘function’, ‘procedure’, ‘begin’, ‘.’ }	{ ‘var’, ‘function’, ‘procedure’, ‘begin’, ε }
var_declaration’	{ ID, ‘,’, ε }	{ ‘function’, ‘procedure’, ‘begin’ }	{ ID, ‘function’, ‘procedure’, ‘begin’, ε }
var_list’	{ ‘,’, ε }	{ ‘:’ , ’)’ }	{ ‘)’, ‘,’, ‘:’, ε }
type_specifier	{ ‘integer’, ‘real’, ‘string’, ‘array’ }	{ ‘;’ }	{ ‘integer’, ‘real’, ‘string’, ‘array’ }
basic_type	{ ‘integer’, ‘real’, ‘string’ }	{ ‘;’ }	{ ‘integer’, ‘real’, ‘string’ }
functions_block	{ ‘function’ }	{ ‘procedure’, ‘begin’ }	{ ‘function’ }
functions_block’	{ ‘function’, ε }	{ ‘procedure’, ‘begin’ }	{ ‘function’, ‘procedure’, ‘begin’, ε }
procedures_block	{ ‘procedure’ }	{ ‘begin’ }	{ ‘procedure’ }
procedures_block’	{ ‘procedure’, ε }	{ ‘begin’ }	{ ‘procedure’, ‘begin’, ε }
params	{ ID, ε }	{ ‘)’ }	{ ID, ‘)’, ε }
param_list’	{ ID, ε }	{ ‘)’ }	{ ID, ‘)’, ε }
local_declarations	{ ‘var’, ε }	{ ‘begin’ }	{ ‘var’, ‘begin’, ’ε }
statement_list	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }	{ ‘end’, ‘until’ }	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }
statement_list’	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’, ε }	{ ‘end’, ‘until’ }	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’, ‘end’, ‘until’ }

statement	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }	{ ‘else’. ‘;’ }	{ ID, ‘begin’, ‘if’, ‘for’, ‘repeat’, ‘readln’, ‘writeln’ }
statement’	{ ‘[’, ‘(’, ε }	{ ‘else’. ‘;’ }	{ ‘(’, ‘[’, ε }
assignment_stmt’	{ ID, INT, REAL, STRING, ‘(’ }	{ ‘else’. ‘;’ }	{ ID, INT, REAL, STRING, ‘(’ }
selection_stmt’	{ ‘else’, ε }	{ ‘else’. ‘;’ }	{ ‘else’, ‘ ; ’ }
output_list’	{ ‘,’, ε }	{ ‘) ’ }	{ ‘ , ’ , ‘) ’ }
output	{ ID, INT, REAL, STRING, ε }	{ ‘ , ’ }	{ ID, INT, REAL, STRING, ‘ , ’ }
var’	{ ‘[’, ε }	{ ‘:=’, ‘*’, ‘/’, ‘+’, ‘-’ }	{ ‘[’, ‘ := ‘ , ‘ * ‘ , ‘ / ‘ , ε }
logic_expression	{ ID, INT, REAL, STRING, ‘(’ }	{ ‘) ’ }	{ ID, INT, REAL, STRING, ‘(’ }
relop	{ ‘<=’, ‘<’, ‘>’, ‘>=’, ‘=’, ‘<>’ }	{ ID, INT, REAL , ‘ (’ }	{ ‘<=’, ‘<’, ‘>’, ‘>=’, ‘=’, ‘<>’ }
arithmetic_expression	{ ID, INT, REAL, STRING, ‘(’ }	{ ‘else’, ‘)’, ‘<’, ‘<=’, ‘<>’, ‘>=’, ‘>’, ‘=’, ‘+’, ‘-’, ‘*’, ‘/’, ‘;’, ‘]’ }	{ ID, INT, REAL, STRING, ‘(’ }
arithmetic_expression′	{ ‘+’, ‘-’, ε }	{ ‘else’, ‘) ’, ‘<’, ‘<=’, ‘<>’, ‘>=’, ‘>’, ‘=’, ‘+’, ‘-’, ‘*’, ‘/’, ‘;’, ‘]’ }	{ ‘] ’, ‘ ; ’ , ‘ else ’, ‘ , ‘ , ‘<=’, ‘<’, ‘>’, ‘>=’, ‘=’, ‘<>’ , ‘)’, ε }
term	{ ID, INT, REAL, ‘(’ }	{ ‘+’, ‘-’ }	{ ID, INT, REAL, STRING, ‘(’ }
term′	{ ‘*’, ‘/’, ε }	{ ‘+’, ‘-’ }	{ ‘*’, ‘/’, ‘ + ’, ‘ - ’ }
factor	{ ID, INT, REAL, ‘(’ }	{ ‘*’, ‘/’ }	{ ID, INT, REAL, STRING, ‘(’ }
factor′	{ ‘(’, ‘[’, ε }	{ ‘*’, ‘/’ }	{ ‘(’, ‘[’, ε }

args	{ ID, INT, REAL, ‘(’, ε }	{ ‘)’ }	{ ID, INT, REAL, STRING, ‘(’, ε }
arg_list ′	{ ‘,’, ε }	{ ‘)’ }	{ ‘ , ’ , ‘) ‘ , ε }
var_list	{ ID }	{ ‘ , ‘ }	{ ID }

3.4. Algoritmo de Análisis Descendente (Descenso Recursivo/Recursive Descend)

El análisis descendente es un enfoque para construir un parser que comienza desde el símbolo inicial de la gramática y se desplaza hacia abajo en su jerarquía. A diferencia de los parsers LL(1), no se requiere predecir el siguiente símbolo de entrada de manera anticipada(parsing table). Aquí están las razones para elegir este enfoque:

★ **Facilidad de implementación:**
  - Los parsers de análisis descendente son más sencillos de implementar en comparación con otros tipos de parsers, como los LR parsers.
  - Las reglas de producción son más directas y no hay necesidad de construir tablas de análisis.
  - La simplicidad facilita la escritura del código.

★ **Flexibilidad:**
  - El análisis descendente permite adaptarse a gramáticas más generales, incluso aquellas que no cumplen con las restricciones de LL(1).
  - No estamos limitados por la predictibilidad de los símbolos siguientes .

★ **Generación de árboles de análisis:**
  - Generación de Árboles de Análisis:
  - El enfoque descendente facilita la construcción de árboles de análisis sintáctico.

3.5. Arquitectura Preliminar y Estructuras de Datos

Me basé en este modelo genérico para el diseño de mi implementación del parser de tipo **recursive descent.**

★ **Gramática simple:**

```
Unset
                    <SNum>   ::= + <num> | - <num> | <num>
                       <num>    ::= <digit> { <digit> }
              <digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

★ **Convertida a EBNF**

```
Unset

<SNum>   ::= ('+' | '-')? <num>

<num>    ::= <digit> { <digit> }

<digit>  ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

★ **Código genérico:**

```
Python
# Inicialización: token apunta al primer token en la secuencia de
entrada
token = get_next_token()

def SNum():
    if token == '+':
        match('+')
    elif token == '-':
        match('-')
    num()
    match('$')  # Verifica el final de la secuencia de tokens

def num():
    digit()
    while token in ['0', '1', ..., '9']:
        digit()

def digit():
    if token in ['0', '1', ..., '9']:
        match(token)
    else:
        error("Se esperaba un dígito")

def match(t):
    if token == t:
        advance_token_pointer()
    else:
        error(f"Se esperaba '{t}'")

def advance_token_pointer():
    # Avanza al siguiente token
    nonlocal token
    token = get_next_token()

def error(msg):
    # Manejo de errores
    print(f"Error: {msg}")

# Llamada inicial
SNum()

print("\nInput\t\t\tAction")
print("--------------------------------")

if E() and not cursor:
    print("--------------------------------")
    print("La cadena se analizó correctamente.")
else:
    print("--------------------------------")
    print("Error al analizar la cadena.")
```

- ★ **Explicación:**
    - Cada función (SNum, num, digit) representa un no terminal de la gramática.
    - La función match(t) verifica si el token actual coincide con el esperado y avanza el puntero de tokens si es así.

- **Estructuras de datos:**
    - **Deque[Token]:**
        - Se utilizó un deque para la extracción de tokens de manera **FIFO**, similar a un queue. La ventaja del deque es que permite también indexar sobre la estructura, lo que simplificó de manera significativa las operaciones de peek necesarias para tomar decisiones sobre los tokens.
    - **Dict[str, list[Symbol]]:**
        - Se utilizó un diccionario con llaves de tipo string y valores de tipo **list[Symbol]** para simplificar el acceso según el tipo de token necesario (integer, real, string o identificador).
    - **Token**:
        - Estructura que representa los tokens obtenidos de la fase de análisis léxico. Cuentan con propiedades de valor y tipo para su análisis sintáctico.
    - **Symbol**:
        - Estructura que representa los distintos símbolos encontrados durante la fase de análisis léxico. Estos símbolos son actualizados durante el análisis semántico con su subtipo (función, procedimiento o programa) y con su tipo de variable o de retorno.
    - **AST Node**:
        - Estructura que sirve como raíz para el árbol abstracto de sintaxis. Cada uno de los nodos del árbol es una subclase de esta estructura y cuenta con propiedades que permiten obtener información semántica del nodo, ejemplo: **ForStmt**, **IfStmt**, **Expression**, etc.
- **Componente de Sintaxis en la Arquitectura del Compilador**
    - Tokenizar el código de entrada (análisis léxico):
        - El diseño inicial del lexer contemplaba partir los lexemas del código fuente usando expresiones regulares, las cuales fueron después reemplazadas por un parseo manual usando una máquina de estado y una tabla de transiciones.
    - Comparar el flujo de tokens con las reglas gramaticales del lenguaje (análisis sintáctico):
        - El diseño preliminar del analizador sintáctico contemplaba el uso de una arquitectura de tipo **LL1**, la cual fue complicada de implementar. Por ello se decidió cambiar a un parser de tipo recursive descent con una implementación que mezcla elementos imperativos y recursivos según sea más sencillo.

- Construir un árbol de análisis sintáctico o un AST (árbol de sintaxis abstracta):
    - La implementación inicial hacía uso de objetos no tipados con IDs únicos para reconocer qué elemento semántico se estaba analizando. Esto resultó difícil de hacer funcionar con las múltiples estructuras, por lo que se reemplazó por una implementación usando clases y subclases en Python para la implementación de un **patrón de visitante**, donde sólo se analizan los nodos que nos interesan en la fase actual.
- **Detectar y reportar errores de sintaxis:**
    - El analizador de sintaxis reporta errores específicos según la regla gramatical que se esté analizando. Esto incluye errores al tener sentencias incompletas o al no contar con el símbolo esperado, por ejemplo al faltar comillas o puntos y coma.

4. Implementación

Github: https://github.com/Paula-Sofia-Soto-Ayala/Compiladores

4.1. Código fuente

★ **Parser.py**

- **ANOTACIONES DESPUÉS DEL EXAMEN ORAL:**
  Si es un recursive descend aunque no cada función se alinee perfectamente a una sola producción, la función start es donde se llaman todos las demás funciones que parsean diferentes constructos un programa de Pascal–, esas funciones llaman a otras y esto funciona de manera jerárquica y mutuamente recursiva(`parse_statement` llama a `compound_statement` y `compound_statement` llama a `parse_statement`) hasta que parsea las cosas correctamente o falla(con ayuda de métodos de la clase Parser que funcionan como lookahead(`peek_token(no consume el token)` , `expect_type` y `expect_value(consumen el token usando el método next_token que si consume el token)`)), y por simplicidad en la forma de ingeniería decidí partir una producción en varias funciones que al final se llaman en una sola función o varias producciones en una sola función, el output es AST y cada estructura que se parsea crea un nodo del tipo que luego se recorre para actualizar la tabla de símbolos.

```python
from __future__ import annotations
from collections import deque
import sys
```

```python
from Visitor import TreeVisitor
from Scanner import Lexer
from AstNodes import Token
from AstNodes import *

# Represents a basic parser, takes a deque of Token as input during
initialization
class Parser:
    # Initializes the Parser with a deque of Token objects
    def __init__(self, tokens: deque[Token]) -> None:
        self.tokens = tokens

    # Returns the first token in the queue without removing it
    def peek_token(self) -> Token | None:
        return self.tokens[0] if self.tokens else None

    # Removes and returns the first token in the queue
    def next_token(self) -> Token | None:
        return self.tokens.popleft() if self.tokens else None

    # Checks if the queue is empty
    def empty_queue(self) -> bool:
        return not self.tokens or len(self.tokens) == 0

    # Checks if there are tokens in the queue
    def have_tokens(self) -> bool:
        return not self.empty_queue()

    # Expects the next token to have a specific type
    def expect_type(self, expected_type: str, error_message: str):
        if self.empty_queue():
            self.error(error_message)

        token = self.next_token()
        if token.type != expected_type:
            self.error(f"Expected {expected_type} but got: {token}")
        return token
    # Expects the next token to have a specific value
    def expect_value(self, expected_val: str, error_message: str):
        if self.empty_queue():
            self.error(error_message)

        token = self.next_token()
        if token.val != expected_val:
            self.error(f"Expected {expected_val} but got: {token}")
        return token

    # Raises a SyntaxError with the given message
    def error(self, message: str):
        raise SyntaxError(message)

    # start → program ID ; vars_block functions_block
procedures_block begin statement_list end .
    def start(self) -> SourceNode:
        # Parses the program header
        prog = self.parse_program_head()

        # Parse variable declarations
        vars = self.parse_variables()

        # Parse functions and procedures
        subprogs = self.parse_subprogs()

        # Parse program body
        code = self.compound_statement(ignoreSemicolon=True)
        self.expect_value(".", "Missing end '.'")

        print("Finished parsing program")

        # Return the generated AST Nodes
        return SourceNode(prog, vars, subprogs, code)

    # Parses the program head
    def parse_program_head(self) -> ProgramNode:
        self.expect_value("program", "Empty program")
        # Parse the program identifier
        ident = self.expect_type("identifier", "No program 
identifier")
        self.expect_type("semicolon", "Missing semicolon")

        print("Parsed program head correctly")

        # Return the header AST Node
        return ProgramNode(ident)

    # Parses a variable declaration block inside a 
program/function/procedure
    # vars_block → var ID var_list’ : type_specifier ; 
var_declaration’ | ε
    def parse_variables(self) -> VariablesNode:
        start = None
        vars: list[VarNode] = []

        # Expect a declaration list if we see the 'VAR' keyword
        if self.peek_token().val == "var":
            start = self.expect_value("var", "Expected variable 
block start")

            while self.peek_token().type == "identifier":
                # Parse a variable list separated by commas
                vars.extend(self.parse_var_list())
                self.expect_value(";", "Missing semi-colon")

            print("Parsed variables block correctly")
            return VariablesNode(start, vars)

        # Else just return an empty VAR section Node
        print("No variable declarations")
        return VariablesNode()

    # Parses a variable list separated by commas
    # var_declaration’ → var_list : type_specifier ; 
var_declaration’ | ε
    def parse_var_list(self) -> list[VarNode]:
        output: list[Token] = []
        while self.have_tokens():
            # Expect identifiers until we see a colon char ':'
            output.append(self.expect_type("identifier", "Expected
variable identifier"))

            # Expect a type definition
            if self.peek_token().val == ":":
                break
            # Else expect another variable
            else:
                self.expect_value(",", "Missing comma in variable
declaration")

        # Parse the declaration list type
        self.expect_value(":", "Missing colon")
        type = self.parse_type()

        # Return a list of identifiers with their types
        return [ VarNode(name, type) for name in output ]

    # Parses the program's subprograms (function|procedure)
    # functions_block’ → function ID( params ) : type_specifier ;
local_declarations begin statement_list end ; functions_block’ | ε
    # procedures_block’ → procedure ID( params ) ;
local_declarations begin statement_list end ; procedures_block’ | ε
    def parse_subprogs(self) -> list[SubprogNode]:
        progs = []
        while (token := self.peek_token().val) in ("function",
"procedure"):
            progs.append(self.parse_subprog(token))

        return progs

    # Parses a subprogram (function|procedure) with their vars,
code, and start
    def parse_subprog(self, token: str) -> SubprogNode:
        # Parse function header
        node = self.parse_subprog_header(token, FunctionNode if
token == "function" else ProcedureNode)

        # Parse local declarations
        # local_declarations → vars_block | ε
        node.vars = self.parse_variables()

        # Parse function body
        node.code = self.compound_statement()

        return node

    # Parses the function or procedure header, including:
identifier, args, and return type
    def parse_subprog_header(self, token: str, node:
type[SubprogNode]) -> SubprogNode:
        start = self.expect_value(token, f"Missing {token} keyword")
        print(f"Parsing {token}: {self.peek_token().val}")

        # Parses the program identifier name
        name = self.expect_type("identifier", "Missing function
identifier")
        self.expect_value("(", "Missing open paren")

        # Parse function parameters
        params = []
        while self.peek_token().type == "identifier":
            # Parse a list of arguments separated by semicolons
            # params → ID var_list’ : type_specifier ; param_list’
| ε
            params.extend(self.parse_var_list())
            # Reached end of function params
            if self.peek_token().val == ")":
                break
            # Separate params with a ';'
            else:
                self.expect_value(";", "Missing semicolon")

        self.expect_value(")", "Missing closing paren")
        # Parse a return type if parsing a function
        if token == "function":
            self.expect_value(":", "Missing colon")
            ret_type = self.parse_type()
        # Else ignore if parsing a procedure
        else: ret_type = None

        self.expect_value(";", "Missing semi-colon")

        # Return the subprogram AST Node
        return node(start, name, params, ret_type)

    # Parse a statement AST Node
    # statement_list → statement ; statement_list’
    def parse_statement(self, ignoreSemicolon = False) -> StmtNode:
        # If we see a begin then parse a compound statement
        # statement_list’ → statement ; statement_list’ | ε
        if self.peek_token().val == "begin":
            return self.compound_statement(ignoreSemicolon)
        # Else parse a simple statement (no semicolon needed)
        else:
            return self.simple_statement()

    # Parses a simple statement, which doesn't need a semicolon
    # statement → ID statement’
    #    | begin statement_list end
    #    | if ( logic_expression ) then statement selection_stmt’
    #    | for ID := int_number to int_number do statement
    #    | repeat statement_list until ( logic_expression )
    #    | readln ( ID var_list’ );
    #    | writeln ( output output_list’) ;
    def simple_statement(self) -> StmtNode:
        match self.peek_token():
            case Token(val="if"):
                # Parses an if-then-else block
                return self.parse_if_block()
            case Token(val="for"):
                # Parses a for-loop block
                return self.parse_for_loop()
            case Token(val="repeat"):
                # Parses a repeat-until block
                return self.parse_repeat()
            case Token(val="readLn"):
                # Parses a readLn statement
                return self.parse_readln()
            case Token(val="writeLn"):
                # Parses a writeLn statement
                return self.parse_writeln()
            case Token(type="identifier"):
                # Parses either a function call
                if self.tokens[1].val == "(":
                    return self.parse_call()
                # Or an assignment statement
                else:
                    # assignment_stmt' → arithmetic_expression |
STRING
                    return self.parse_assignment()
            case _:
                # Errors when failing to match any previous
statement
                self.error("Expected a simple statement")

    # Parses a compound statement, which requires a begin-else
    # And a list of semicolon separated simple statements.
    def compound_statement(self, ignoreSemicolon = False) ->
StmtNode:
        start = self.expect_value("begin", "Missing begin keyword")
        stmts = []

        # Parse the nested statements inside the compound statement
        while self.peek_token().val != "end":
            stmts.append(self.parse_statement(ignoreSemicolon))
            self.expect_value(";", "Missing semi-colon")

        # Stop once we find an end keyword
        self.expect_value("end", "Missing end keyword")
        # Some constructs have optional semicolons at the end
        # Ignore if `ignoreSemicolon = True`
        if not ignoreSemicolon:
            self.expect_value(";", "Missing semi-colon")

        # Return a compound statement AST Node with its statements
list
        return CompoundStmtNode(start, stmts)

    # Parse a variable assignment statement
    # var’ → [ arithmetic_expression ] | ε
    def parse_assignment(self) -> StmtNode:

        # Expect an identifier to assign to (lhs)
        name = self.parse_identifier()

        # If the left side is an array expect an index
        if self.peek_token().val == "[":
            self.expect_value("[", "Missing indexing open bracket")

            # Arrays accept expressions on their indexers, i.e:
array[expr]
            index = self.parse_expression()
            # Create an indexer expression AST Node to report
            name = IndexExpr(name, index)
            self.expect_value("]", "Missing indexing closing
bracket")

        # Assignment operator and right side expression
        self.expect_value(":=", "Expected assignment operator")

        # Check right side of the assignment
        expr = self.parse_expression()

        # Return an assignment statement AST Node
        return AssignStmtNode(name, expr)

    # Parse a writeLn statement with its argument list
    def parse_writeln(self) -> WriteLnNode:
        start = self.expect_value("writeLn", "Expected WRITELN 
keyword")
        self.expect_value("(", "Missing open parenthesis")

        # Parse the arguments to be written to IO
        # output_list’ → output output_list’ | ε
        # output → arithmetic_expression | STRING | ε
        args = self.parse_arguments(optional=True)
        self.expect_value(")", "Missing closing parenthesis")

        # Return an AST Node with the statement start and args
        return WriteLnNode(start, args)

    # Parse an argument list for subprogram calls / writeLn 
statements
    # args → arithmetic_expression arg_list’ | ε
    def parse_arguments(self, optional: bool) -> list[ExprNode]:
        # If arguments are optional no error on empty arguments
        if self.peek_token().val == ")":
            if optional:
                print("No function or procedure arguments")
                return []
            else:
                self.error("Empty arguments on call")

        exprs: list[ExprNode] = []
        while self.have_tokens():
            # arg_list’ → arithmetic_expression arg_list’ | ε
            # Parse the argument expressions individually
            exprs.append(self.parse_expression())

            # If we reach a closing paren then exit
            if self.peek_token().val == ")":
                break
            # Otherwise expect a comma and more expressions
            else:
                self.expect_value(",", "Missing comma in output 
list")
```

```
        # Return the arguments list as an ExprNode list
        return exprs

    # Parse a readLn statement
    def parse_readln(self) -> ReadLnNode:
        start = self.expect_value("readLn", "Expected READLN
keyword")
        self.expect_value("(", "Missing open parenthesis")

        # Expect an identifier that will receive the value of the IO
read
        id = self.expect_type("identifier", "Missing target
identifier for readLn")
        self.expect_value(")", "Missing closing parenthesis")

        # Return an AST Node with the ReadLn statement
        return ReadLnNode(start, id)

    # Parse an if-then-else statement
    def parse_if_block(self) -> IfNode:

        # Save the if branch start
        start = self.expect_value("if", "Expected IF keyword")
        else_branch = None

        self.expect_value("(", "Missing open parenthesis")

        # Parse the condition expr evaluated for the branch
        condition = self.parse_condition()
        self.expect_value(")", "Missing closing parenthesis")
        self.expect_value("then", "Missing THEN keyword")

        # Parse the IF-THEN branch with NO semicolon
        then_branch = self.parse_statement(ignoreSemicolon=True)

        # Optionally parse the ELSE branch with NO semicolon
        # selection_stmt’ → else statement | ε
        if self.peek_token().val == "else":
            self.expect_value("else", "Missing ELSE keyword")
            # Save the else branch statement
            else_branch = self.parse_statement(ignoreSemicolon=True)

        # Return an IfBlock AST Node with its start, condition, and
branches
        return IfNode(start, condition, then_branch, else_branch)

    # Parse a for-loop statement with its range and code block
    def parse_for_loop(self) -> ForNode:

        # Save the start node for the loop
        start = self.expect_value("for", "Expected FOR keyword")

        # Parse the loop index variable
        name = self.parse_identifier()
        self.expect_value(":=", "Expected assignment operator")

        # Parse the for-loop index range
        to_num = self.parse_int_lit()
        self.expect_value("to", "Expected TO keyword")
        from_num = self.parse_int_lit()

        self.expect_value("do", "Expected DO keyword")

        # Parse its statement list and save them
        stmts = [self.parse_statement(ignoreSemicolon=True)]

        # Return a ForLoop AST Node with its start, index var,
range, and statements
        return ForNode(start, name, to_num, from_num, stmts)

    # Parse a repeat-until code block with its condition
    def parse_repeat(self) -> RepeatNode:
        # Save the repeat block start
        start = self.expect_value("repeat", "Expected REPEAT
keyword")

        stmts = []
        while self.peek_token().val != "until":
            # Parse each statement individually
            stmts.append(self.parse_statement(ignoreSemicolon=True))
            self.expect_value(";", "Missing semicolon")

        # Stop once we find the UNTIL keyword
        self.expect_value("until", "Expected UNTIL keyword")
        self.expect_value("(", "Missing open parenthesis")

        # Parse the UNTIL loop condition
        condition = self.parse_condition()
        self.expect_value(")", "Missing closing parenthesis")

        # Return a RepeatUntil AST Node with its start, condition,
and statements
        return RepeatNode(start, condition, stmts)

    # Parse a subprogram call with its arguments
    def parse_call(self) -> StmtNode:

        # Parse the function identifier
        name = self.parse_identifier()
        self.expect_value("(", "Missing open parenthesis")

        # Parse its argument list
        args = self.parse_arguments(optional=True)
        self.expect_value(")", "Missing closing parenthesis")

        # Return a CallStmtNode with the subprogram name and args
        return CallStmtNode(name, args)

    # Parse a condition expression
    def parse_condition(self) -> ExprNode:
        return self.parse_equality_expr()

    # Parse an expression Node
    # i.e: func call, identifiers, literals, conditions, array
index, etc...
    # arithmetic_expression → term   arithmetic_expression’
    def parse_expression(self) -> ExprNode:
        return self.parse_equality_expr()

    # Lowest precedence expr: equality < relational
    # relop → <= | < | > | >= | = | <>
    def parse_equality_expr(self) -> ExprNode:
        lhs = self.parse_relational_expr()

        token = self.peek_token()
        match token.val:
            case "=": node = EqExpr
            case "<>": node = NeExpr
            case _: return lhs

        # Found the expr operator
        if node:
            self.next_token()

        rhs = self.parse_expression()
        return node(lhs, rhs)

    # Higher precedence than equality: relational < addition
    # relop → <= | < | > | >= | = | <>
    def parse_relational_expr(self) -> ExprNode:
        lhs = self.parse_add_expr()

        token = self.peek_token()
        match token.val:
            case "<": node = LeExpr
            case "<=": node = LtExpr
            case ">": node = GtExpr
            case ">=": node = GeExpr
            case _: return lhs

        # Found the expr operator
        if node:
            self.next_token()

        rhs = self.parse_expression()
        return node(lhs, rhs)
    # Higher precedence than relational expr: addition <
multiplication
    # arithmetic_expression’ ⟶ + term arithmetic_expression’| -
term arithmetic_expression’| ε
    def parse_add_expr(self) -> ExprNode:
        lhs = self.parse_mult_expr()

        token = self.peek_token()
        match token.val:
            case "+": node = AddExpr
            case "-": node = SubExpr
            case _: return lhs

        # Found the expr operator
        if node:
            self.next_token()

        rhs = self.parse_expression()
        return node(lhs, rhs)

    # Higher precedence than additive expr: multiplication <
simple_expr
    # term’ ⟶ * factor term’ | / factor  term’ | ε
    def parse_mult_expr(self) -> ExprNode:
        lhs = self.parse_simple_expr()

        token = self.peek_token()
        match token.val:
            case "*": node = MultExpr
            case "/": node = DivExpr
            case _: return lhs

        # Found the expr operator
        if node:
            self.next_token()

        rhs = self.parse_expression()
        return node(lhs, rhs)
    # Highest precedence expression
    # Includes: literals, calls, indexing, and nested expressions
    # Factor → ID factor' | int_number | real_number | (
arithmetic_operator )
    def parse_simple_expr(self) -> ExprNode:
        token = self.peek_token()

        # We've reached a leaf, so we expect a Factor
        if token.type in ["integer", "real", "string"]:
            return self.parse_literal()
        # factor' → ( args ) | var'
        elif token.type == "identifier":
            if self.tokens[1].val == "(":
                # function call
                return self.parse_call()
            if self.tokens[1].val == "[":
                # indexing expression
                return self.parse_array_indexing()
            else:
                # identifier expression
                return self.parse_identifier()
        # Otherwise parse an expr inside parenthesis
        # factor' → ( args ) | var'
        elif token.val == "(":
            self.next_token()  # consume '('
            output = self.parse_expression()
            self.expect_value(")", "Expected closing parenthesis")
            return output
        # If we fail to match an expression then throw an error
        else:
            self.error("Expected simple expression")

    # Parse an array indexing operation
    def parse_array_indexing(self) -> IndexExpr:
        # Parse the array name
        name = self.parse_identifier()
        self.expect_value("[", "Missing open bracket")
        # Parse the indexing expression inside brackets
        index = self.parse_expression()
        self.expect_value("]", "Missing closing bracket")

        # Return an IndexExpr AST Node with the array and indexer
        return IndexExpr(name, index)

    # Parse a variable indentifier
    # factor’ → ( args ) | var’
    def parse_identifier(self) -> IdentNode:
        name = self.expect_type("identifier", "Expected an
identifier")

        # Return an indentifier AST Node with the variablenam
        return IdentNode(name)

    # Parse a literal: string | integer | real
    # Examples: "hello", 4, 3.1416
    def parse_literal(self) -> ExprNode:
        match self.peek_token().type:
            case "string": return self.parse_string_lit()
            case "integer": return self.parse_int_lit()
            case "real": return self.parse_real_lit()

    # Parse a string literal AST Node, i.e: "world"
    def parse_string_lit(self) -> StringLiteral:
        token = self.expect_type("string", "Expected a string")
        return StringLiteral(token)

    # Returns a real literal AST Node, i.e: 12.34
    def parse_real_lit(self) -> RealLiteral:
        token = self.expect_type("real", "Expected a real")
        return RealLiteral(token)

    # Returns an integer literal AST Node, i.e: 25
    def parse_int_lit(self) -> IntLiteral:
        token = self.expect_type("integer", "Expected an integer")
        return IntLiteral(token)
    # Parse a type declaration
    # type_specifier → integer | real | string | array [ int_number
.. int_number ] of basic_type
    def parse_type(self) -> TypeNode:
        token = self.peek_token()

        match token.val:
            # Match either a simple type
            case "integer" | "real" | "string":
                return self.parse_simple_type()
            # Or an array type
            case "array":
                return self.parse_array_type()
            # Or throw an error on a failing match
            case _:
                self.error(f"Expected type declaration but got:
{token}")

    # Parses a simple type declaration
    # basic_type → integer | real | string
    def parse_simple_type(self) -> TypeNode:
        token = self.next_token()
        match token.val:
            # Returns a Type AST Node with the simple type
            case "integer" | "real" | "string":
                return TypeNode(token)
            # Th
            case _:
                self.error(f"Expected type but got: {token}")
                return None

    # Parses an array type with its range + simple type
    def parse_array_type(self) -> TypeNode:
        # Parse the type start node
        start = self.expect_value("array", "Missing array keyword")
        self.expect_value("[", "Missing open bracket")

        # Parse the array length / range
        from_num = self.parse_int_lit()
        self.expect_value("..", "Missing range operator")
        to_num = self.parse_int_lit()

        self.expect_value("]", "Missing close bracket")
        self.expect_value("of", "Missing 'of' keyword")

        # Parse the array subtype (a simple type, i.e: real, int,
string)
        subtype = self.parse_simple_type()

        # Return an ArrayType AST Node with its start, subtype, and
range
        return ArrayTypeNode(start, subtype, from_num, to_num)


def main():
    args = sys.argv

    # Validate we got a file name arg
    if len(args) > 1:
        # Run the tokenization step
        print("\n--Starting lexical anaylisis--\n\n")
        lexer = Lexer()
        lexer.start(args[1] or "Test5.txt")

        print("\n--Finished lexical anaylisis--\n")

        print("--Started syntax anaylisis--\n")
        # Create an instance of the parser
        parser = Parser(deque(lexer.tokens))
        tree = parser.start()

        print("\n--Finished syntax anaylisis--\n")

        # Print the generated AST Nodes
        print(f"AST: {tree}")

        print("\n--Started semantical analysis--\n")
        # Visit AST and update symbol table
        visitor = TreeVisitor(symbols=lexer.symbols)
        visitor.visit(tree)

        # Print the updated symbol table
        lexer.print_symbols()
    else:
        print("No file provided for parsing")

if __name__ == "__main__":
    main()

"""
3.1.- Gramática final
1.   start → program ID ; vars_block functions_block
procedures_block begin statement_list end •
2.   vars_block → VAR var_list' : type_specifier ; var_declaration'
| ε
3.   var_declaration' → var_list : type_specifier ;
var_declaration' | ε
4.   var_list' → , ID var_list' | ε
5.   type_specifier → integer | real | string | array [ int_number
• • int_number ] of basic_type
6.   basic_type → integer | real | string
7.   functions_block → functions_block'
8.   functions_block' → function ID( params ) : type_specifier ;
local_declarations begin statement_list end ; functions_block' | ε
9.   procedures_block → procedures_block'
10.  procedures_block' → procedure ID( params ) ;
local_declarations begin statement_list end ; procedures_block' | ε
11.  params → ID var_list' : type_specifier ; param_list' | ε
12.  param_list' → var_list : type_specifier ; param_list' | ε
13.  local_declarations → vars_block | ε
14.  statement_list → statement ; statement_list'
15.  statement_list' → statement ; statement_list' | ε
16.  statement → ID statement' | begin statement_list end | if (
logic_expression ) then statement selection_stmt' | for ID :=
int_number to int_number do statement | repeat statement_list until
( logic_expression ) | readln ( ID var_list' ); | writeln ( output
output_list') ;
```

```
17.  statement' → var' :=  assignment_stmt' | ( args )
18.  assignment_stmt' → arithmetic_expression | STRING
19.  selection_stmt' → else statement | ε
20.  output_list' → , output output_list' | ε
21.  output → arithmetic_expression | STRING | ε
22.  var' → [ arithmetic_expression ] | ε
23.  logic_expression → arithmetic_expression relop
arithmetic_expression
24.  relop → <= | < | > | >= | = | <>
25.  arithmetic_expression → term   arithmetic_expression'
26.  arithmetic_expression' → + term arithmetic_expression'| - term
arithmetic_expression'| ε
27.  term → factor term'
28.  term' → * factor   term' | / factor  term' | ε
29.  factor → ID factor' | int_number | real_number  |
(arithmetic_operator)
30.  factor' → ( args ) | var'
31.  args → arithmetic_expression arg_list' | ε
32.  arg_list' → , arithmetic_expression arg_list' | ε
"""
```

★ **AstNodes.py**

```
from __future__ import annotations
from typing import Protocol

class Token:
    def __init__(self, val: str, type: str, id: int | None = None)
-> None:

        """
        Initializes a Token object.

        Args:
            val (str): The value of the token.
            type (str): The type of the token.
            id (int | None, optional): An optional identifier.
Defaults to None.
        """

        self.val = val
        self.type = type
        self.id = id

    def __str__(self) -> str:

        """
        Returns a string representation of the Token.

        Returns:
            str: A string representation of the Token.
        """

        return str((self.val, self.type))

class AstNode(Protocol):
    id: str

    def __str__(self) -> str:

        """
        Returns a string representation of the AstNode.

        Returns:
            str: A string representation of the AstNode.
        """

        return self.id

class IdentNode(AstNode):
    id: str = 'ident'

    def __init__(self, value: Token):

        """
        Initializes an IdentNode.

        Args:
            value (Token): The token representing the identifier.
        """
        self.value = value

    def __str__(self) -> str:

        """

        Returns a string representation of the IdentNode.

        Returns:
            str: A string representation of the IdentNode.
        """

        return f'(identifier {self.value.val!r})'

class SourceNode(AstNode):
    id = 'source'

    def __init__(self, prog: 'ProgramNode', vars: 'VariablesNode',
subprogs: list['SubprogNode'], code: 'StmtNode'):

        """

        Initializes a SourceNode.

        Args:
            prog (ProgramNode): The program node.
            vars (VariablesNode): The variables node.
            subprogs (list[SubprogNode]): List of subprogram nodes.
            code (StmtNode): The code node.
        """

        self.prog = prog
        self.vars = vars
        self.subprogs = subprogs
        self.code = code

    def __str__(self) -> str:

        """

        Returns a string representation of the SourceNode.
        Returns:
            str: A string representation of the SourceNode.
        """

        return f'(source {self.prog} {self.vars} {" ".join(map(str,
self.subprogs))} {self.code})'

class ProgramNode(AstNode):
    id = 'program'

    def __init__(self, name: Token):

        """
        Initializes a ProgramNode.

        Args:
            name (Token): The token representing the program name.
        """

        self.name = name

    def __str__(self) -> str:

        """
        Returns a string representation of the ProgramNode.

        Returns:
            str: A string representation of the ProgramNode.
        """

        return f'(program {self.name.val})'

class VariablesNode(AstNode):
    id = 'variables'

    def __init__(self, start: Token = None, vars: list['VarNode'] =
None):
        """
        Initializes a VariablesNode.

        Args:
            start (Token, optional): The starting token. Defaults to
None.
            vars (list[VarNode], optional): List of variable nodes.
Defaults to an empty list.
        """

        self.start = start
        self.vars = vars or []

    def __str__(self) -> str:

        """
        Returns a string representation of the VariablesNode.

        Returns:
            str: A string representation of the VariablesNode.
        """

        return f'(vars {", ".join(map(str, self.vars))})'

class SubprogNode(AstNode):
    id = 'subprog'

    def __init__(self, type: Token, name: Token, params:
list['VarNode'], ret_type: 'TypeNode' | None, vars: VariablesNode =
None, code: 'StmtNode' = None):

        """
        Initializes a SubprogNode.

        Args:
            type (Token): The type of the subprogram (e.g.,
"function" or "procedure").
            name (Token): The name of the subprogram.
            params (list[VarNode]): List of parameters (variables)
for the subprogram.
            ret_type ('TypeNode' | None): The return type (for
functions) or None (for procedures).
            vars (VariablesNode, optional): Optional variables
defined within the subprogram. Defaults to None.
            code (StmtNode, optional): The body of the subprogram
(statements). Defaults to None.
        """

        self.type = type
        self.name = name
        self.params = params
        self.ret_type = ret_type
        self.vars = vars
        self.code = code
        self.arity = len(params)

class FunctionNode(SubprogNode):
    id = 'function'

    def __str__(self) -> str:

        """
        Returns a string representation of the FunctionNode.

        Returns:
            str: A string representation of the FunctionNode.
        """

        return f'(function {self.name.val} {", ".join(map(str,
self.params))} {self.ret_type} {self.vars} {self.code})'

class ProcedureNode(SubprogNode):
    id = 'procedure'

    def __str__(self) -> str:

        """
        Returns a string representation of the ProcedureNode.

        Returns:
            str: A string representation of the ProcedureNode.
        """

        return f'(procedure {self.name.val} {", ".join(map(str, 
self.params))} {self.vars} {self.code})'

class VarNode(AstNode):
    id = 'var'

    def __init__(self, name: Token, type: 'TypeNode'):

        """
        Initializes a VarNode.

        Args:
            name (Token): The token representing the variable name.
            type ('TypeNode'): The type of the variable.
        """

        self.name = name
        self.type = type

    def __str__(self) -> str:

        """
        Returns a string representation of the VarNode.

        Returns:
            str: A string representation of the VarNode.
        """

        return f'(var {self.name.val}: {self.type})'

class StmtNode(AstNode):
    pass
```

```
class CallStmtNode(StmtNode):
    id = 'call'

    def __init__(self, ident: IdentNode, args: list['ExprNode']):
        self.ident = ident
        self.args = args

    def __str__(self) -> str:
        return f'(call {self.ident} {", ".join(map(str,
self.args))})'

class AssignStmtNode(StmtNode):
    id = 'assign'

    def __init__(self, ident: IdentNode, value: 'ExprNode'):

        """
        Initializes a CallStmtNode.

        Args:
            ident (IdentNode): The identifier (function or procedure
name) being called.
            args (list[ExprNode]): A list of expression nodes
representing the arguments passed to the call.
        """

        self.ident = ident
        self.value = value

    def __str__(self) -> str:

        """
        Returns a string representation of the CallStmtNode.

        Returns:
            str: A string representation of the CallStmtNode.
        """

        return f'(:= {self.ident} {self.value})'
```

```
class IndexExpr(AstNode):
    id = 'index'

    def __init__(self, ident: IdentNode, index: 'ExprNode'):

        """
        Initializes an AssignStmtNode.

        Args:
            ident (IdentNode): The identifier (variable name) being
assigned.
            value ('ExprNode'): An expression node representing the
value being assigned.
        """

        self.ident = ident
        self.index = index

    def __str__(self) -> str:

        """
        Returns a string representation of the AssignStmtNode.

        Returns:
            str: A string representation of the AssignStmtNode.
        """

        return f'(index {self.ident} {self.index})'

class CompoundStmtNode(StmtNode):
    id = 'compound'

    def __init__(self, start: Token, stmts: list[StmtNode]):

        """
        Initializes a CompoundStmtNode.

        Args:
            start (Token): A token representing the start of the
compound statement.
            stmts (list[StmtNode]): A list of statement nodes within
the compound statement.
        """

        self._start = start
        self.stmts = stmts

    def __str__(self) -> str:

        """
        Returns a string representation of the CompoundStmtNode.

        Returns:
            str: A string representation of the CompoundStmtNode.
        """

        return f'({" ".join(map(str, self.stmts))})'

class RepeatNode(StmtNode):
    id = 'repeat'

    def __init__(self, start: Token, condition: AstNode, statements:
list[AstNode]):

        """
        Initializes a RepeatNode.

        Args:
            start (Token): A token representing the start of the
repeat block.
            condition (AstNode): An expression node representing the
loop condition.
            statements (list[AstNode]): List of statements within
the repeat block.
        """

        self._start = start
        self.test = condition
        self.stmts = CompoundStmtNode(start, statements)

    def __str__(self) -> str:

        """
        Returns a string representation of the RepeatNode.

        Returns:
            str: A string representation of the RepeatNode.
        """

        return f'(repeat-until {self.test} {self.stmts})'

class WriteLnNode(StmtNode):
    id: str = 'writeLn'

    def __init__(self, start: Token, args: list['ExprNode']):

        """
        Initializes a WriteLnNode.

        Args:
            start (Token): A token representing the start of the
write-ln statement.
            args (list[ExprNode]): A list of expression nodes to be
printed.
        """

        self._start = start
        self.args = args

    def __str__(self) -> str:

        """
        Returns a string representation of the WriteLnNode.

        Returns:
            str: A string representation of the WriteLnNode.
        """

        return f'(write-ln {" ".join(map(str, self.args))})'

class ReadLnNode(StmtNode):
    id: str = 'readLn'

    def __init__(self, start: Token, var_name: Token):

        """
        Initializes a ReadLnNode.

        Args:
            start (Token): A token representing the start of the
read-ln statement.
            var_name (Token): A token representing the variable
where input is stored.
        """

        self._start = start
        self.var = var_name

    def __str__(self) -> str:

        """
        Returns a string representation of the ReadLnNode.

        Returns:
            str: A string representation of the ReadLnNode.
        """

        return f'(read-ln {self.var})'

class ForNode(StmtNode):
    id: str = 'for'

    def __init__(self, start: Token, name: IdentNode, to_num:
'IntLiteral', from_num: 'IntLiteral', statements: list[AstNode]):
        """
        Initializes a ForNode.

        Args:
            start (Token): A token representing the start of the for
loop.
            name (Token): The loop variable name.
            to_num (IntLiteral): The upper bound of the loop.
            from_num (IntLiteral): The lower bound of the loop.
            statements (list[AstNode]): List of statements within
the for loop.
        """

        self._start = start
        self.name = name
        self.to_num = to_num
        self.from_num = from_num
        self.stmts = CompoundStmtNode(start, statements)

    def __str__(self) -> str:

        """
        Returns a string representation of the ForNode.

        Returns:
            str: A string representation of the ForNode.
        """

        return f'(for ({self.name} {self.from_num} {self.to_num})
{self.stmts})'

class IfNode(StmtNode):
    id: str = 'if'

    def __init__(self, start: Token, condition: 'ExprNode',
then_branch: StmtNode, else_branch: StmtNode | None = None):

        """
        Initializes an IfNode.
        Args:
            start (Token): A token representing the start of the if 
statement.
            condition (ExprNode): An expression node representing 
the condition.
            then_branch (StmtNode): The statement executed if the 
condition is true.
            else_branch (StmtNode | None, optional): The statement 
executed if the condition is false. Defaults to None.
        """

        self._start = start
        self.test = condition
        self.then_branch = then_branch
        self.else_branch = else_branch

    def __str__(self) -> str:

        """
        Returns a string representation of the IfNode.

        Returns:
            str: A string representation of the IfNode.
        """

        if self.else_branch is not None:
            return f'(if ({self.test} {self.then_branch}) (else 
{self.else_branch}))'
        else:
            return f'(if ({self.test} {self.then_branch}))'

class ExprNode(AstNode):
    id: str = 'expr'
    type: str | None = None

    def __init__(self, start: Token, type: str):

        """
        Initializes an ExprNode.

        Args:
            start (Token): A token representing the start of the
expression.
            type (str): The type of the expression.
        """

        self._start = start
        self.type = type

    def __str__(self) -> str:

        """

        Returns a string representation of the ExprNode.

        Returns:
            str: A string representation of the ExprNode.
        """

        return f"(expr ({self.start} {self.type}))"

class AddExpr(ExprNode):
    id = 'add_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """

        Initializes an AddExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:
        """
        Returns a string representation of the AddExpr.

        Returns:
            str: A string representation of the AddExpr.
        """

        return f'(+ {self.lhs} {self.rhs})'

class SubExpr(ExprNode):
    id = 'sub_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes a SubExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the SubExpr.

        Returns:
            str: A string representation of the SubExpr.
        """

        return f'(- {self.lhs} {self.rhs})'

class MultExpr(ExprNode):
    id = 'mult_expr'
    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes a MultExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the MultExpr.

        Returns:
            str: A string representation of the MultExpr.
        """

        return f'(* {self.lhs} {self.rhs})'

class DivExpr(ExprNode):
    id = 'div_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes a DivExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the DivExpr.

        Returns:
            str: A string representation of the DivExpr.
        """

        return f'(/ {self.lhs} {self.rhs})'

class EqExpr(ExprNode):
    id = 'eq_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes an EqExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the EqExpr.

        Returns:
            str: A string representation of the EqExpr.
        """

        return f'(= {self.lhs} {self.rhs})'
```

```
class NeExpr(ExprNode):
    id = 'ne_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """

        Initializes a NeExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """


        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """

        Returns a string representation of the NeExpr.

        Returns:
            str: A string representation of the NeExpr.
        """


        return f'(<> {self.lhs} {self.rhs})'

class LeExpr(ExprNode):
    id = 'le_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """

        Initializes a LeExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the LeExpr.

        Returns:
            str: A string representation of the LeExpr.
        """

        return f'(<= {self.lhs} {self.rhs})'

class LtExpr(ExprNode):
    id = 'lt_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes an LtExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the LtExpr.

        Returns:
            str: A string representation of the LtExpr.
```

```python
        """

        return f'(< {self.lhs} {self.rhs})'

class GeExpr(ExprNode):
    id = 'ge_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes a GeExpr.

        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the GeExpr.

        Returns:
            str: A string representation of the GeExpr.
        """

        return f'(>= {self.lhs} {self.rhs})'

class GtExpr(ExprNode):
    id = 'gt_expr'

    def __init__(self, lhs: ExprNode, rhs: ExprNode):

        """
        Initializes a GtExpr.
        Args:
            lhs (ExprNode): The left-hand side expression.
            rhs (ExprNode): The right-hand side expression.
        """

        self.lhs = lhs
        self.rhs = rhs

    def __str__(self) -> str:

        """
        Returns a string representation of the GtExpr.

        Returns:
            str: A string representation of the GtExpr.
        """

        return f'(> {self.lhs} {self.rhs})'

class StringLiteral(ExprNode):
    id = 'str_lit'
    value: str

    def __init__(self, token: Token):

        """
        Initializes a StringLiteral.

        Args:
            token (Token): The token representing the string
literal.
        """

        self.token = token
        self.value = token.val
        self.raw = token.val[1:-1]

    def __str__(self) -> str:
        """
        Returns a string representation of the StringLiteral.

        Returns:
            str: A string representation of the StringLiteral.
        """

        return self.value

class IntLiteral(ExprNode):
    id = 'int_lit'
    value: int

    def __init__(self, token: Token):

        """
        Initializes an IntLiteral.

        Args:
            token (Token): The token representing the integer
literal.
        """

        self.token = token
        self.value = int(token.val)

    def __str__(self) -> str:

        """
        Returns a string representation of the IntLiteral.

        Returns:
            str: A string representation of the IntLiteral.
        """

        return str(self.value)

class RealLiteral(ExprNode):
    id = 'real_lit'
    value: float

    def __init__(self, token: Token):

        """

        Initializes a RealLiteral.

        Args:
            token (Token): The token representing the real
(floating-point) literal.
        """

        self.token = token
        self.value = float(token.val)

    def __str__(self) -> str:

        """

        Returns a string representation of the RealLiteral.

        Returns:
            str: A string representation of the RealLiteral.
        """

        return str(self.value)

class TypeNode(AstNode):
    id = 'type'

    def __init__(self, value: Token):

        """

        Initializes a TypeNode.

        Args:
            value (Token): The token representing the type.
        """

        self.value = value
    def __str__(self) -> str:

        """
        Returns a string representation of the TypeNode.

        Returns:
            str: A string representation of the TypeNode.
        """

        return f'(type {self.value.val})'

class ArrayTypeNode(TypeNode):
    id = 'array'

    def __init__(self, start: Token, subtype: TypeNode, from_num:
IntLiteral, to_num: IntLiteral):

        """
        Initializes an ArrayTypeNode.

        Args:
            start (Token): A token representing the start of the
array type.
            subtype (TypeNode): The subtype (element type) of the
array.
            from_num (IntLiteral): The lower bound of the array.
            to_num (IntLiteral): The upper bound of the array.
        """

        super().__init__(start)

        self.subtype = subtype
        self.from_num = from_num
        self.to_num = to_num

    def __str__(self) -> str:

        """
        Returns a string representation of the ArrayTypeNode.

        Returns:
            str: A string representation of the ArrayTypeNode.
        """

        return f'(type (array {self.subtype} {self.from_num}
{self.to_num}))'
```

★ **Visitor.py**

```
from typing import Any, List
from Scanner import Symbol
from AstNodes import *

# Define a symbol table type
SymbolTable = dict[str, List[Symbol] | set[Symbol]]

class TreeVisitor:
    """
    Symbol table has shape:
    symbol_table: dict[str, List[Symbol] | set[Symbol]] = {
        'identifier': set([]),
        'real': [],
        'integer': [],
        'string': []
    }
    """
    symbols: dict[str, List[Symbol] | set[Symbol]]


    def __init__(self, symbols: dict[str, List[Symbol] |
set[Symbol]]) -> None:

        """
        TreeVisitor class for traversing an abstract syntax tree
(AST).
        """
        self.symbols = symbols

    def visit(self, node: AstNode) -> Any:

        """

        Dispatch method for different node types.

        Args:
            node (AstNode): The AST node to visit.
        """

        # Use pattern matching to handle different node types
        match node:
            case SourceNode(): return self.visit_source(node)
            case ProgramNode(): return self.visit_program(node)
            case IdentNode(): return self.visit_ident(node)
            case VariablesNode(): return self.visit_variables(node)
            case SubprogNode(): return self.visit_subprogram(node)
            case VariablesNode(): return self.visit_variables(node)
            case VarNode(): return self.visit_variable(node)
            case _:
                # Handle any unhandled node type
                print(f"Unhandled node of type: {node.id}")

    def visit_program(self, node: ProgramNode):

        """

        Processes the program node and updates the program symbol in
the symbol table.

        Args:
            node (ProgramNode): The program node.
        """

        print(f"Program: {node.name.val}")
        identifiers: set[Symbol] = self.symbols["identifier"]

        # Update the program symbol
        for symbol in identifiers:
            if symbol.name == node.name.val:
                symbol.sub_type = "program"

    def visit_ident(self, node: IdentNode):

        """
        Visits an identifier node.

        Args:
            node (IdentNode): The identifier node.
        """

        self.visit(node.value)

    def visit_source(self, node: SourceNode):

        """
        Visits the entire source node, including program, variables,
subprograms, and code.

        Args:
            node (SourceNode): The source node.
        """

        # Visit the program node
        self.visit(node.prog)

        # Visit the variables node
        self.visit(node.vars)

        # Visit each subprogram in the subprogs list
        for subprogram in node.subprogs:
            self.visit(subprogram)

        # Visit the code node
        self.visit(node.code)


    def visit_subprogram(self, node: SubprogNode):
        """
        Processes subprograms (functions or procedures) and updates
their symbols.

        Args:
            node (SubprogNode): The subprogram node.
        """

        # Get the set of symbols associated with identifiers
        identifiers: set[Symbol] = self.symbols["identifier"]
        print(f"Subprogram: {node.name}")

        # Iterate through each symbol
        for symbol in identifiers:
            # Check if the symbol name matches the subprogram name
            if symbol.name == node.name.val:
                # Update the symbol's return type based on the
subprogram's return type
                ret_type = node.ret_type
                symbol.ret_type = ret_type.value.val if ret_type
else None
                # Set the subprogram type (function or procedure)
                symbol.sub_type = "function" if ret_type else
"procedure"

        # Visit the subprogram's variables
        self.visit(node.vars)

        # Visit each parameter in the subprogram
        for param in node.params:
            self.visit(param)


    def visit_variable(self, node: VarNode):

        """

        Handles variable declarations, including array types.
        Args:
            node (VarNode): The variable node.
        """

        # Get the set of symbols associated with identifiers
        identifiers: set[Symbol] = self.symbols["identifier"]

        # Iterate through each symbol
        for symbol in identifiers:
            # Check if the symbol name matches the variable name
            if symbol.name != node.name.val:
                continue
            # If the variable type is an array
            if isinstance(node.type, ArrayTypeNode):
                array_type = node.type
                subtype = array_type.subtype.value
                # Construct the array type string
                type =
f"{subtype.val}[{array_type.from_num}..{array_type.to_num}]"
                symbol.var_type = type
            else:
                # Set the symbol's variable type to the non-array
type
                symbol.var_type = node.type.value.val

        # print(f"{node.token.val}")

    def visit_variables(self, node: VariablesNode):

        """
        Processes variable declarations.

        Args:
            node (VariablesNode): The variables node.
        """

        print(f"Variable Declarations")

        for var in node.vars:
            print(var)
            self.visit(var)

    def visit_assignment(self, node: AssignStmtNode):

        """
        Placeholder for handling assignment statements (not fully
implemented).

        Args:
            node (AssignStmtNode): The assignment statement node.
        """

        print(node)
        self.visit(node.value)
```

4.2. Gestión de Tabla de Símbolos(Symbol Table Management)

La tabla de símbolos es un diccionario que mapea de tipo de símbolo a una lista de símbolos. Registra el subtipo del símbolo (función, procedimiento, programa), así como su tipo de variable, tipo de retorno y el contenido del símbolo. Durante el paso de análisis semántico se actualizan estos valores al recorrer el árbol abstracto de sintaxis de manera recursiva haciendo uso de un patrón de visitante a base de clases y subclases de Nodos. Los símbolos son inicialmente creados en la fase de análisis léxico, sin embargo aún no tienen información más allá de lo básico (tipo general: identificador, real, entero, etc.), su nombre y su ID de ingreso en la tabla.

4.3. Casos No Implementados

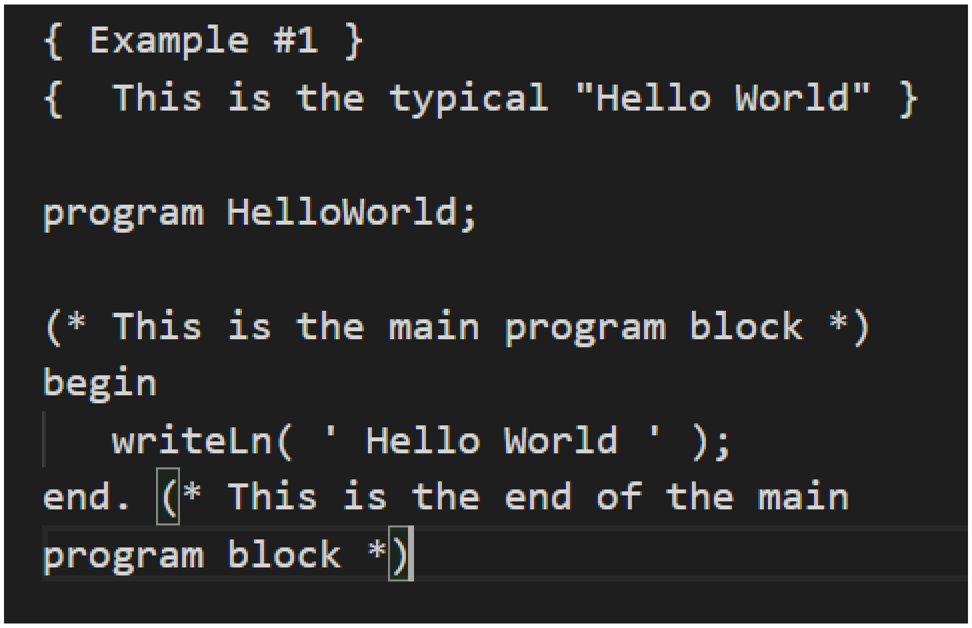
- ★ Checar tipos, las variables ya tienen tipos asignados, pero no se revisa que se usen correctamente
- ★ Ejemplo, no se reporta si al asignar el tipo coincide:
- ★ x: string := 5 no reporta error
- ★ El usar operadores o funciones tampoco reporta errores por tipos incompatibles, ejemplo: 5 * "hola" o min(5, "pez")
- ★ Todo esto a raíz de que no pusimos visitors para sacar el tipo de las expresiones.

5. Verificación y Validación

5.1. Casos de Prueba

5.1.1. Test1.txt

Escenario probado: Un programa simple que no tiene variables ni subprogramas. Utilizado para tener una base sólida para reconocer las estructuras base de un programa.

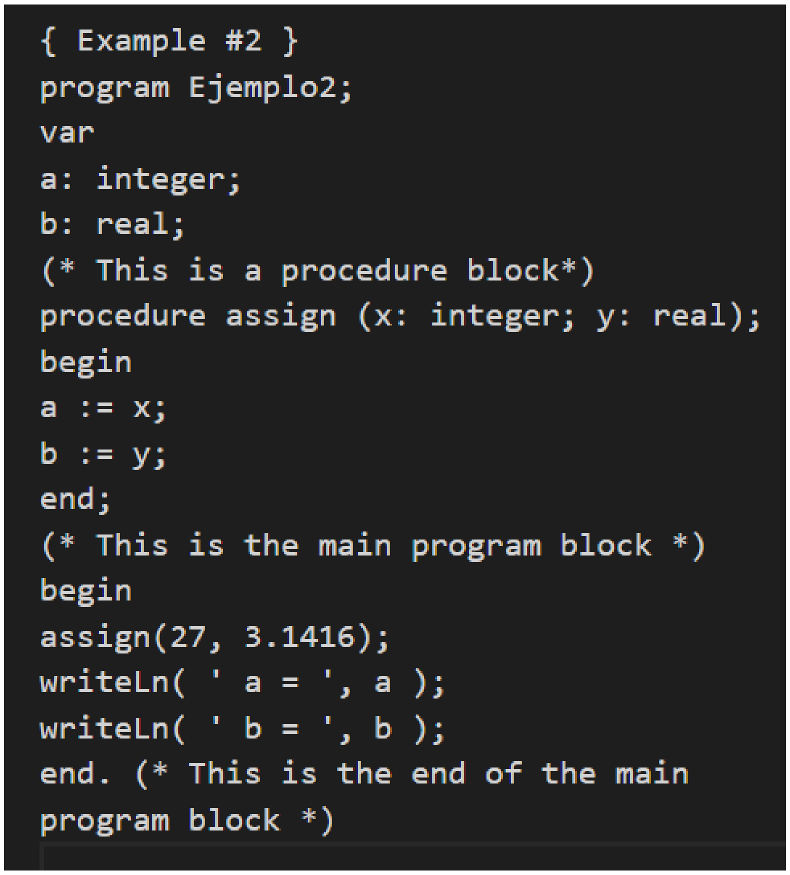


```
{ Example #1 }
{  This is the typical "Hello World" }

program HelloWorld;

(* This is the main program block *)
begin
   writeLn( ' Hello World ' );
end. (* This is the end of the main
program block *)
```

FIgura 5.1.1.1

5.1.2. Test2.txt

Escenario probado: Programa sencillo con declaraciones mínimas que prueba el reconocimiento de bloques de variables y subprogramas, así como la correcta actualización de identificadores en la tabla de símbolos.

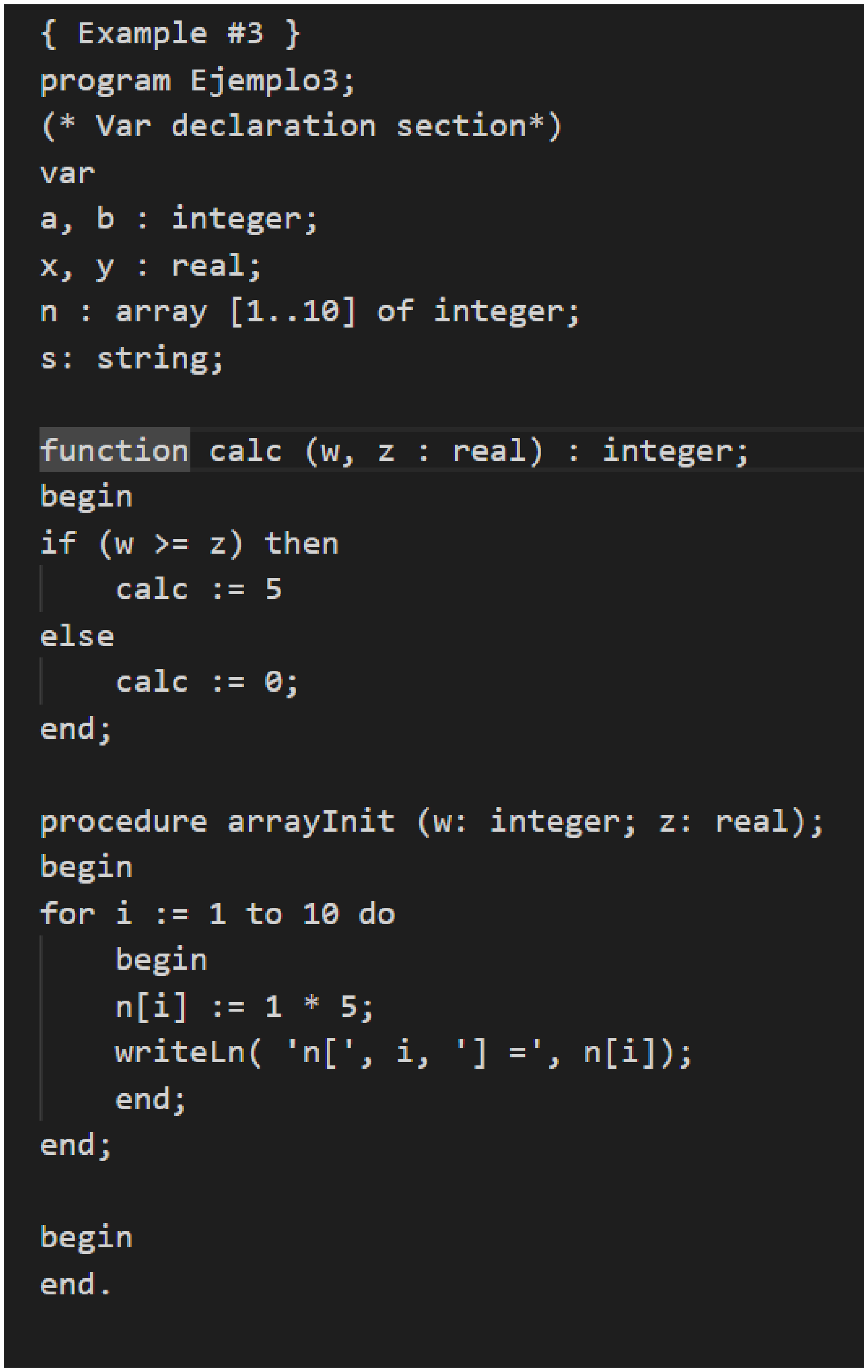


```
{ Example #2 }
program Ejemplo2;
var
a: integer;
b: real;
(* This is a procedure block*)
procedure assign (x: integer; y: real);
begin
a := x;
b := y;
end;
(* This is the main program block *)
begin
assign(27, 3.1416);
writeLn( ' a = ', a );
writeLn( ' b = ', b );
end. (* This is the end of the main
program block *)
```

Figura 5.1.2.1

5.1.3. Test3.txt

Escenario probado: Programa que cuenta con todas estructuras válidas de un programa de Pascal. Se utilizó para probar el correcto reconocimiento de sentencias y expresiones anidadas, así como de argumentos de función.

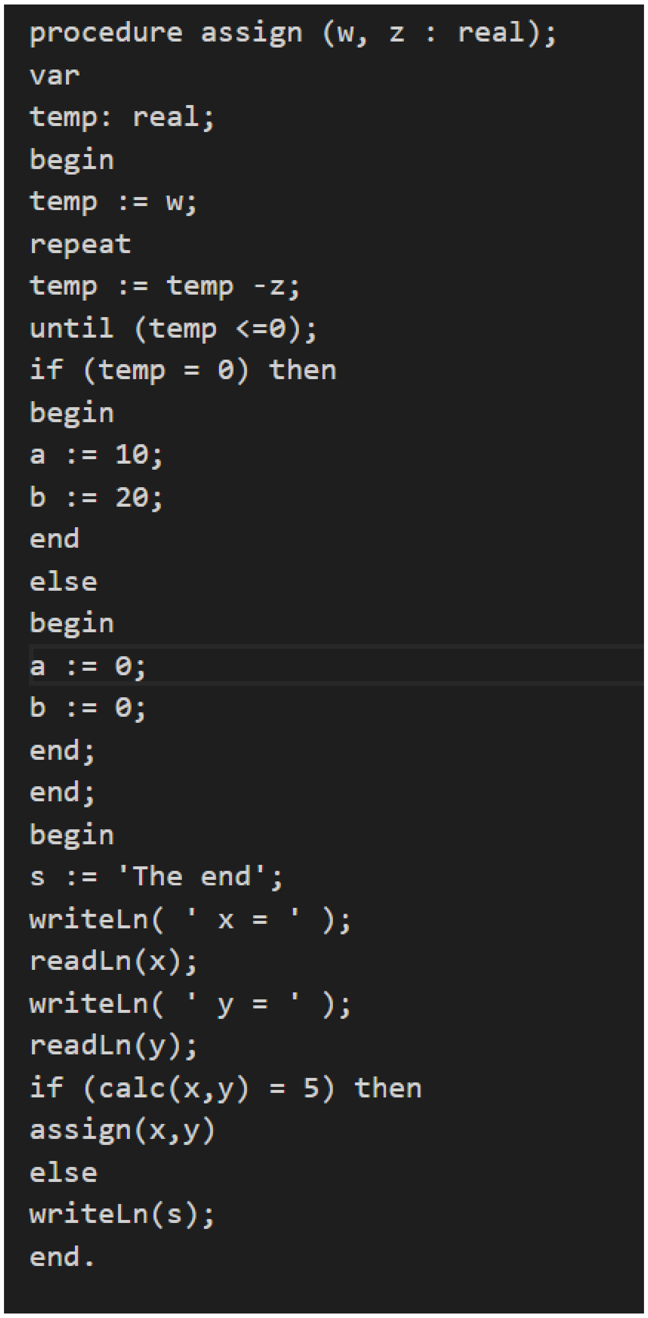


```
{ Example #3 }
program Ejemplo3;
(* Var declaration section*)
var
a, b : integer;
x, y : real;
n : array [1..10] of integer;
s: string;

function calc (w, z : real) : integer;
begin
if (w >= z) then
    calc := 5
else
    calc := 0;
end;

procedure arrayInit (w: integer; z: real);
begin
for i := 1 to 10 do
    begin
    n[i] := 1 * 5;
    writeLn( 'n[', i, '] =', n[i]);
    end;
end;

begin
end.
```

Figura 5.1.3.1

5.1.4. Test4.txt

Escenario probado: Programa erróneo que se usó para probar el reporte de errores del parser. Cuenta con sentencias válidas, pero con estructura general incorrecta que debe reportarse por el analizador.

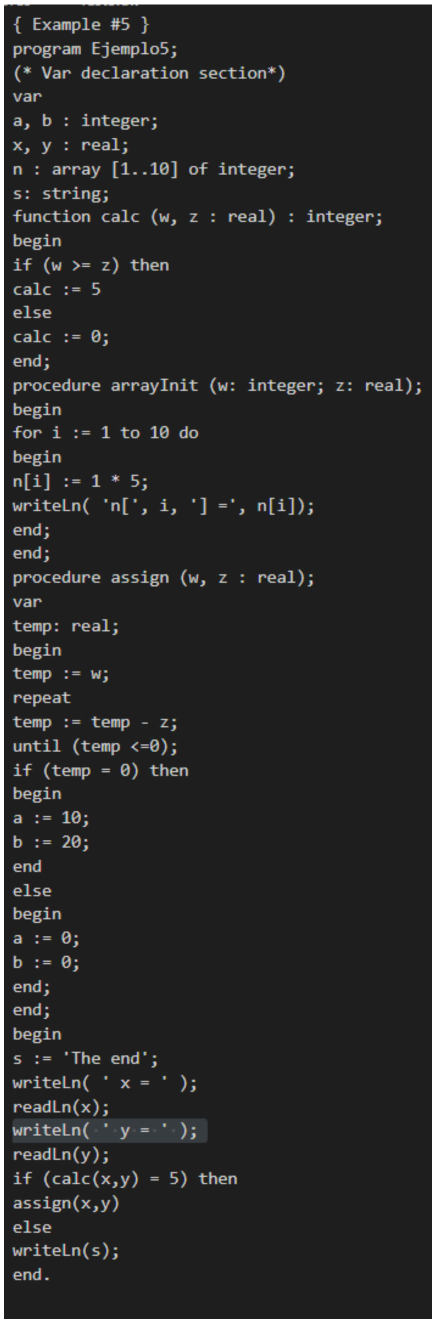


```
procedure assign (w, z : real);
var
temp: real;
begin
temp := w;
repeat
temp := temp -z;
until (temp <=0);
if (temp = 0) then
begin
a := 10;
b := 20;
end
else
begin
a := 0;
b := 0;
end;
end;
begin
s := 'The end';
writeLn( ' x = ' );
readLn(x);
writeLn( ' y = ' );
readLn(y);
if (calc(x,y) = 5) then
assign(x,y)
else
writeLn(s);
end.
```

Figura 5.1.4.1

5.1.5. Test5.txt

Escenario probado: Versión aún más compleja del escenario 3, ya que cuenta con las mismas sentencias, así como casos adicionales anidados. La estructura es válida pero el analizador semántico reporta múltiples variables no declaradas al concluir su actualización de la tabla de símbolos.

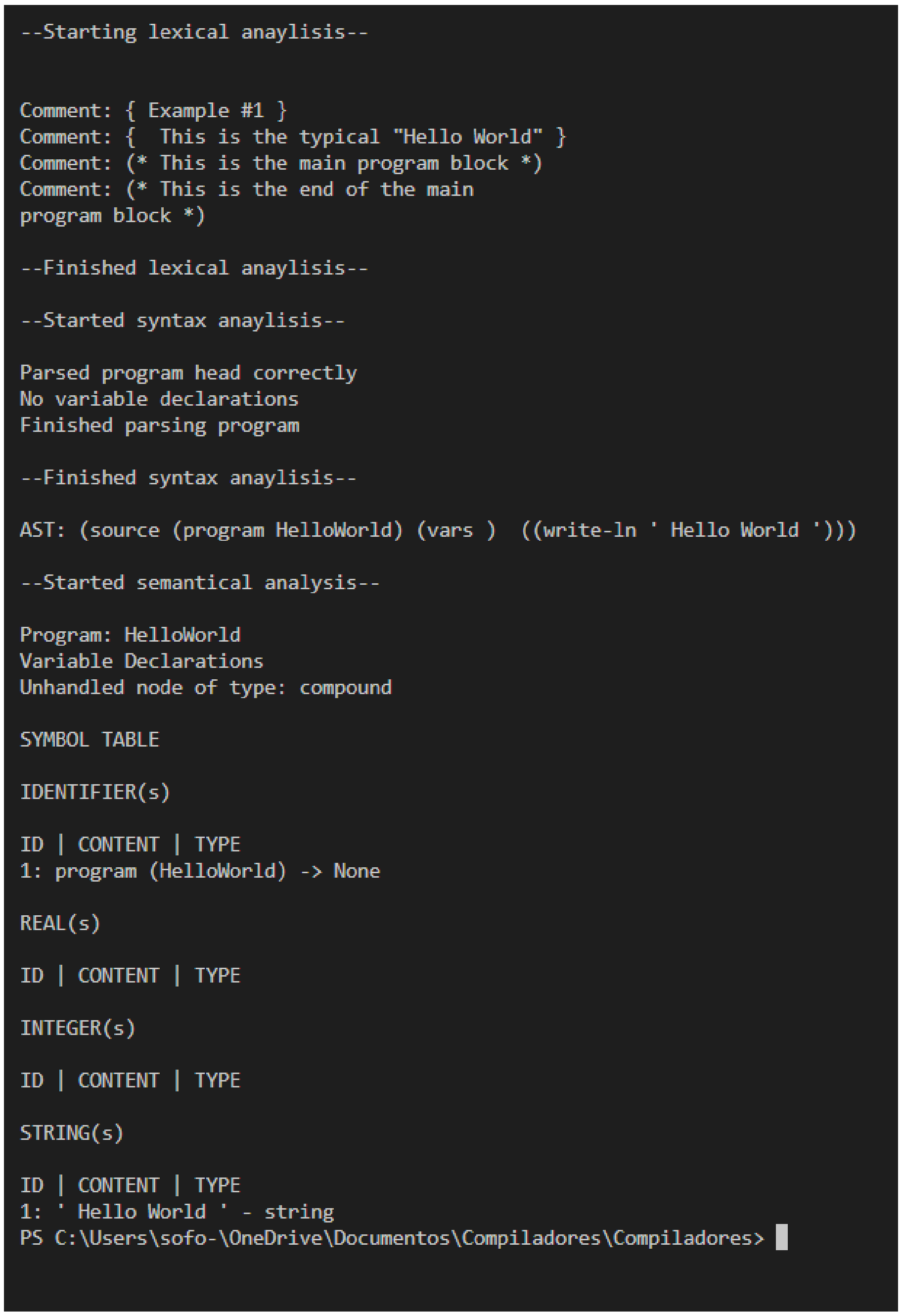


```
{ Example #5 }
program Ejemplo5;
(* Var declaration section*)
var
a, b : integer;
x, y : real;
n : array [1..10] of integer;
s: string;
function calc (w, z : real) : integer;
begin
if (w >= z) then
calc := 5
else
calc := 0;
end;
procedure arrayInit (w: integer; z: real);
begin
for i := 1 to 10 do
begin
n[i] := 1 * 5;
writeLn( 'n[', i, '] =', n[i]);
end;
end;
procedure assign (w, z : real);
var
temp: real;
begin
temp := w;
repeat
temp := temp - z;
until (temp <=0);
if (temp = 0) then
begin
a := 10;
b := 20;
end
else
begin
a := 0;
b := 0;
end;
end;
begin
s := 'The end';
writeLn( ' x = ' );
readLn(x);
writeLn( ' y = ' );
readLn(y);
if (calc(x,y) = 5) then
assign(x,y)
else
writeLn(s);
end.
```

Figura 5.1.5.1

5.2. Output del Parser

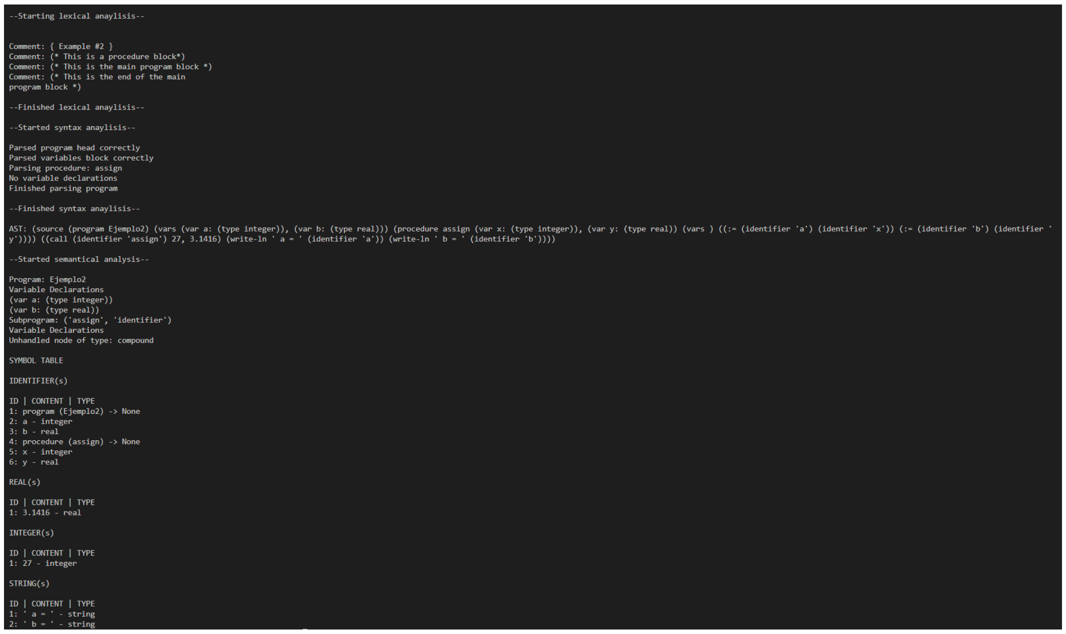
5.2.1. Output Test1.txt



```
--Starting lexical anaylisis--


Comment: { Example #1 }
Comment: {  This is the typical "Hello World" }
Comment: (* This is the main program block *)
Comment: (* This is the end of the main
program block *)

--Finished lexical anaylisis--

--Started syntax anaylisis--

Parsed program head correctly
No variable declarations
Finished parsing program

--Finished syntax anaylisis--

AST: (source (program HelloWorld) (vars )  ((write-ln ' Hello World ')))

--Started semantical analysis--

Program: HelloWorld
Variable Declarations
Unhandled node of type: compound

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
1: program (HelloWorld) -> None

REAL(s)

ID | CONTENT | TYPE

INTEGER(s)

ID | CONTENT | TYPE

STRING(s)

ID | CONTENT | TYPE
1: ' Hello World ' - string
PS C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores>
```

Figura 5.2.1.1

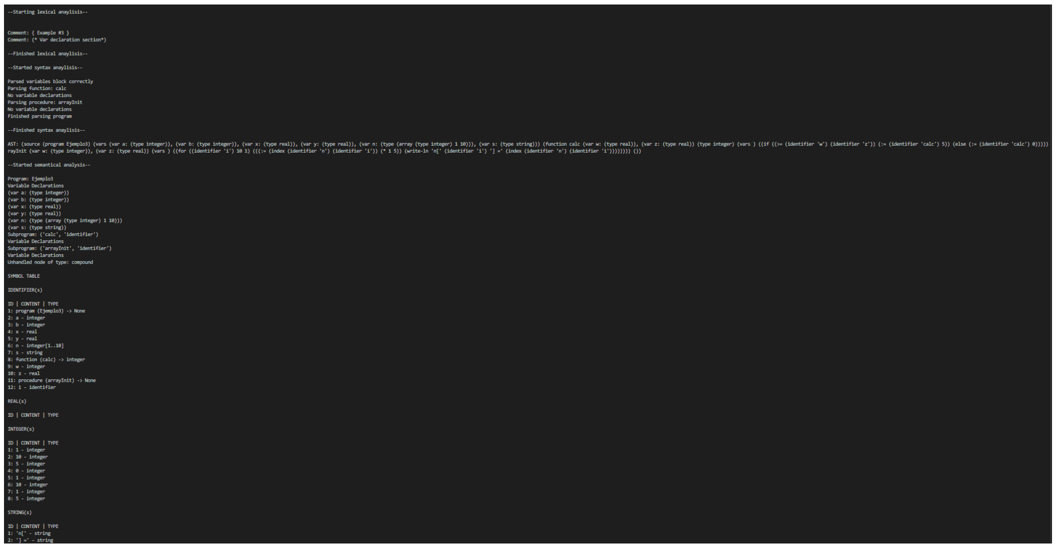
5.2.2. Output Test2.txt



```
--Starting lexical anaylisis--


Comment: { Example #2 }
Comment: (* This is a procedure block*)
Comment: (* This is the main program block *)
Comment: (* This is the end of the main
program block *)

--Finished lexical anaylisis--

--Started syntax anaylisis--

Parsed program head correctly
Parsed variables block correctly
Parsing procedure: assign
No variable declarations
Finished parsing program

--Finished syntax anaylisis--

AST: (source (program Ejemplo2) (vars (var a: (type integer)), (var b: (type real))) (procedure assign (var x: (type integer)), (var y: (type real)) (vars ) ((:= (identifier 'a') (identifier 'x')) (:= (identifier 'b') (identifier '
y')))) ((call (identifier 'assign') 27, 3.1416) (write-ln ' a = ' (identifier 'a')) (write-ln ' b = ' (identifier 'b'))))

--Started semantical analysis--

Program: Ejemplo2
Variable Declarations
(var a: (type integer))
(var b: (type real))
Subprogram: ('assign', 'identifier')
Variable Declarations
Unhandled node of type: compound

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
1: program (Ejemplo2) -> None
2: a - integer
3: b - real
4: procedure (assign) -> None
5: x - integer
6: y - real

REAL(s)

ID | CONTENT | TYPE
1: 3.1416 - real

INTEGER(s)

ID | CONTENT | TYPE
1: 27 - integer

STRING(s)

ID | CONTENT | TYPE
1: ' a = ' - string
2: ' b = ' - string
```

Figura 5.2.2.1

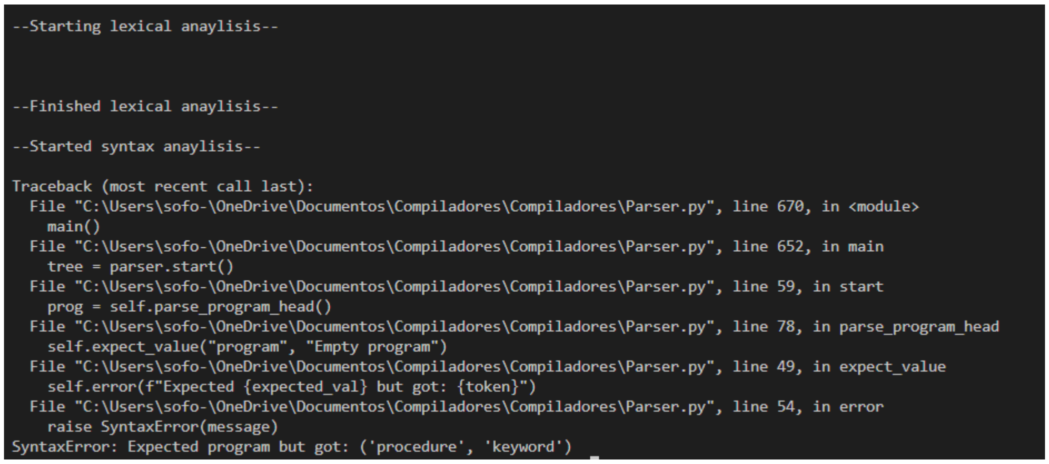
5.2.3. Output Test3.txt



```
--Starting lexical anaylisis--


Comment: { Example #3 }
Comment: (* Var declaration section*)

--Finished lexical anaylisis--

--Started syntax anaylisis--

Parsed variables block correctly
Parsing function: calc
No variable declarations
Parsing procedure: arrayInit
No variable declarations
Finished parsing program

--Finished syntax anaylisis--

AST: (source (program Ejemplo3) (vars (var a: (type integer)), (var b: (type integer)), (var x: (type real)), (var y: (type real)), (var n: (type (array (type integer) 1 10))), (var s: (type string))) (function calc (var w: (type real)), (var z: (type real)) (type integer) (vars ) ((if ((>= (identifier 'w') (identifier 'z')) (:= (identifier 'calc') 5)) (else (:= (identifier 'calc') 0)))))
rayInit (var w: (type integer)), (var z: (type real)) (vars ) ((for ((identifier 'i') 10 1) (((:= (index (identifier 'n') (identifier 'i')) (* 1 5)) (write-ln 'n[' (identifier 'i') '] =' (index (identifier 'n') (identifier 'i')))))))) ())

--Started semantical analysis--

Program: Ejemplo3
Variable Declarations
(var a: (type integer))
(var b: (type integer))
(var x: (type real))
(var y: (type real))
(var n: (type (array (type integer) 1 10)))
(var s: (type string))
Subprogram: ('calc', 'identifier')
Variable Declarations
Subprogram: ('arrayInit', 'identifier')
Variable Declarations
Unhandled node of type: compound

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
1: program (Ejemplo3) -> None
2: a - integer
3: b - integer
4: x - real
5: y - real
6: n - integer[1..10]
7: s - string
8: function (calc) -> integer
9: w - integer
10: z - real
11: procedure (arrayInit) -> None
12: i - identifier

REAL(s)

ID | CONTENT | TYPE

INTEGER(s)

ID | CONTENT | TYPE
1: 1 - integer
2: 10 - integer
3: 5 - integer
4: 0 - integer
5: 1 - integer
6: 10 - integer
7: 1 - integer
8: 5 - integer

STRING(s)

ID | CONTENT | TYPE
1: 'n[' - string
2: '] =' - string
```

Figura 5.2.3.1

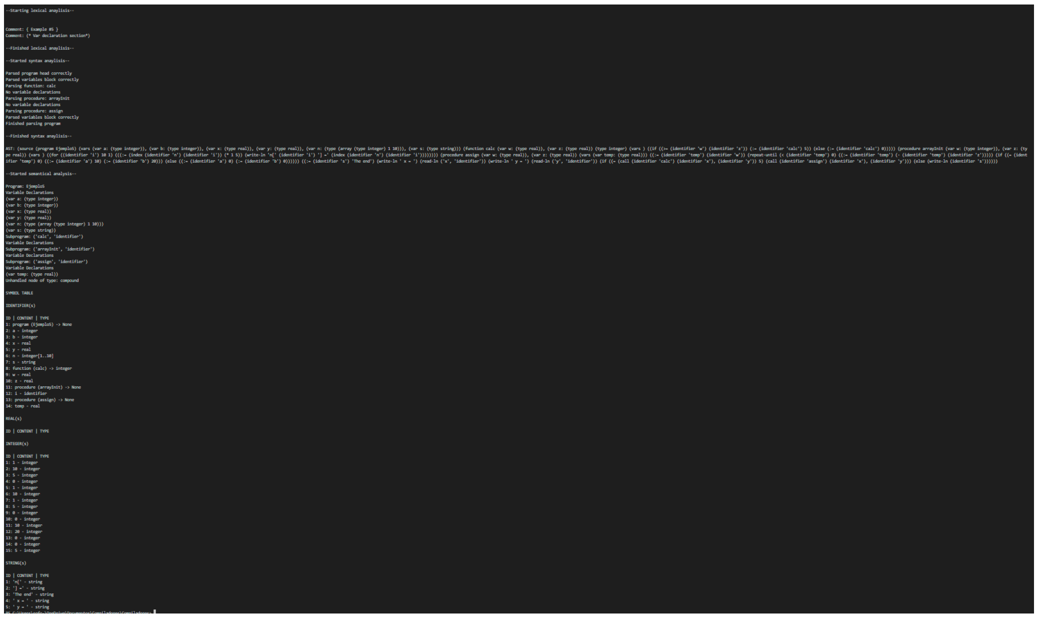
5.2.4. Output Test4.txt



```
--Starting lexical anaylisis--



--Finished lexical anaylisis--

--Started syntax anaylisis--

Traceback (most recent call last):
  File "C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores\Parser.py", line 670, in <module>
    main()
  File "C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores\Parser.py", line 652, in main
    tree = parser.start()
  File "C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores\Parser.py", line 59, in start
    prog = self.parse_program_head()
  File "C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores\Parser.py", line 78, in parse_program_head
    self.expect_value("program", "Empty program")
  File "C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores\Parser.py", line 49, in expect_value
    self.error(f"Expected {expected_val} but got: {token}")
  File "C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores\Parser.py", line 54, in error
    raise SyntaxError(message)
SyntaxError: Expected program but got: ('procedure', 'keyword')
```

Figura 5.2.4.1

5.2.5. Output Test5.txt



```
--Starting lexical anaylisis--


Comment: { Example #5 }
Comment: (* Var declaration section*)

--Finished lexical anaylisis--

--Started syntax anaylisis--

Parsed program head correctly
Parsed variables block correctly
Parsing function: calc
No variable declarations
Parsing procedure: arrayInit
No variable declarations
Parsing procedure: assign
Parsed variables block correctly
Finished parsing program

--Finished syntax anaylisis--

AST: (source (program Ejemplo5) (vars (var a: (type integer)), (var b: (type integer)), (var x: (type real)), (var y: (type real)), (var n: (type (array (type integer) 1 10))), (var s: (type string))) (function calc (var w: (type real)), (var z: (type real)) (type integer) (vars ) ((if ((>= (identifier 'w') (identifier 'z')) (:= (identifier 'calc') 5)) (else (:= (identifier 'calc') 0))))) (procedure arrayInit (var w: (type integer)), (var z: (ty
pe real)) (vars ) ((for ((identifier 'i') 10 1) (((:= (index (identifier 'n') (identifier 'i')) (* 1 5)) (write-ln 'n[' (identifier 'i') '] =' (index (identifier 'n') (identifier 'i')))))))) (procedure assign (var w: (type real)), (var z: (type real)) (vars (var temp: (type real))) ((:= (identifier 'temp') (identifier 'w')) (repeat-until (< (identifier 'temp') 0) ((:= (identifier 'temp') (- (identifier 'temp') (identifier 'z'))))) (if ((= (ident
ifier 'temp') 0) ((:= (identifier 'a') 10) (:= (identifier 'b') 20))) (else ((:= (identifier 'a') 0) (:= (identifier 'b') 0)))))) ((:= (identifier 's') 'The end') (write-ln ' x = ') (read-ln ('x', 'identifier')) (write-ln ' y = ') (read-ln ('y', 'identifier')) (if ((= (call (identifier 'calc') (identifier 'x'), (identifier 'y')) 5) (call (identifier 'assign') (identifier 'x'), (identifier 'y'))) (else (write-ln (identifier 's'))))))

--Started semantical analysis--

Program: Ejemplo5
Variable Declarations
(var a: (type integer))
(var b: (type integer))
(var x: (type real))
(var y: (type real))
(var n: (type (array (type integer) 1 10)))
(var s: (type string))
Subprogram: ('calc', 'identifier')
Variable Declarations
Subprogram: ('arrayInit', 'identifier')
Variable Declarations
Subprogram: ('assign', 'identifier')
Variable Declarations
(var temp: (type real))
Unhandled node of type: compound

SYMBOL TABLE

IDENTIFIER(s)

ID | CONTENT | TYPE
1: program (Ejemplo5) -> None
2: a - integer
3: b - integer
4: x - real
5: y - real
6: n - integer[1..10]
7: s - string
8: function (calc) -> integer
9: w - real
10: z - real
11: procedure (arrayInit) -> None
12: i - identifier
13: procedure (assign) -> None
14: temp - real

REAL(s)

ID | CONTENT | TYPE

INTEGER(s)

ID | CONTENT | TYPE
1: 1 - integer
2: 10 - integer
3: 5 - integer
4: 0 - integer
5: 1 - integer
6: 10 - integer
7: 1 - integer
8: 5 - integer
9: 0 - integer
10: 0 - integer
11: 10 - integer
12: 20 - integer
13: 0 - integer
14: 0 - integer
15: 5 - integer

STRING(s)

ID | CONTENT | TYPE
1: 'n[' - string
2: '] =' - string
3: 'The end' - string
4: ' x = ' - string
5: ' y = ' - string
PS C:\Users\sofo-\OneDrive\Documentos\Compiladores\Compiladores>
```

Figura 5.2.5.1

6. Referencias

1. R. Castelló, Class Lecture, Topic: “TC3048_Chapter_4_Syntax_Analysis_Part_II.pdf, School of Engineering and Science,ITESM, Chihuahua, Chih, April, 2024. 14