UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

COMPILADORES

Documentación

MINI COMPILADOR PASCAL

DESARROLLADORAS

GARCÍA ESTRADA ARIADNA DENISSE

GONZÁLEZ ARIAS ALEXANDRA GALILEA

Fecha: Junio de 2025

Versión del sistema: 1.0

CONTENIDO

Capturas del funcionamiento.....2
- Quitar los comentarios (* Este es un comentario *).....2
- Marcar error cuando falta ";".....3
- Marca un error cuando una variable se redeclara.....3

Capturas extras del funcionamiento.....10
¿Cómo funciona el Compilador?.....11
Explicación del código.....12
- Clase Variable.....12
- Listas y Diccionarios Globales.....12
- Funciones de Gestión de Registros.....13
- Funciones de Gestión de la Tabla de Símbolos.....14
- Análisis Léxico (Tokenización).....15
- Procesamiento de Expresiones.....16
- Generación de Código Intermedio y Ensamblador.....16
- Generación de Código de Llamadas al Sistema RISC-V.....17
- Procesamiento y Simulación de Instrucciones.....18
- Flujo Principal de Compilación.....19

Capturas del funcionamiento

- **Quitar los comentarios** (* Este es un comentario *)

miniCompilador.py | main.pas | for.pas | expresiones.pas

main.pas

```
1   var x1 real;
2   var x2 real; (* coordenadas del 1er punto *)
3   var y1 real;
4   var y2 real; (* coordenadas del 2do punto *)
5   var m real; (* pendiente de la recta *)
6   write("Escriba el valor de x1: ");
7   read(x1);
8   writeln("Escriba el valor de x2: ");
9   read(x2);
10  write("Escriba el valor de y1: ");
11  read(y1);
12  writeln("Escriba el valor de y2: ");
13  read(y2);
14  m := (y2 - y1) / (x2 - x1);
15  writeln("La pendiente es: ", m);
16  end.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
-------------------------------- ANÁLISIS LÉXICO --------------------------------
----- Quita comentarios ----
var x1 real;
var x2 real;
var y1 real;
var y2 real;
var m real;
write("Escriba el valor de x1: ");
read(x1);
writeln("Escriba el valor de x2: ");
read(x2);
write("Escriba el valor de y1: ");
read(y1);
writeln("Escriba el valor de y2: ");
read(y2);
m := (y2 - y1) / (x2 - x1);
writeln("La pendiente es: ", m);
end.
----------------------
```

- **Marcar error cuando falta ";"**

compiladorPascal.py | expresiones.pas | for.pas | forDos.pas | main.pas

main.pas

```
1   var x1 real;
2   var x2 real; (* coordenadas del 1er punto *)
3   var y1 real
4   var y2 real; (* coordenadas del 2do punto *)
5   var m real; (* pendiente de la recta *)
6   writeln("Escriba el valor de x1: ");
7   read(x1);
8   write("Escriba el valor de x2: ");
9   read(x2);
10  writeln("Escriba el valor de y1: ")
11  read(y1);
12  write("Escriba el valor de y2: ");
13  read(y2);
14  m := (y2 - y1) / (x2 - x1);
15  writeln("La pendiente es: ", m);
16  end;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
----------------------
Error: Falta un ";" al final de la línea 3 -> var y1 real
Error: Falta un ";" al final de la línea 10 -> writeln("Escriba el valor de y1: ")

Tokens encontrados:
var -> Palabra reservada
x1 -> ID
real -> Tipo
; -> Simbolo especial
var -> Palabra reservada
```

- **Marca un error cuando una variable se redeclara**

Se intentó redeclarar la variable y1.

main.pas

```
1   var x1 real;
2   var x2 real; (* coordenadas del 1er punto *)
3   var y1 real;
4   var y2 real; (* coordenadas del 2do punto *)
5   var m real; (* pendiente de la recta *)
6   write("Escriba el valor de x1: ");
7   read(x1);
8   writeln("Escriba el valor de x2: ");
9   read(x2);
10  write("Escriba el valor de y1: ");
11  read(y1);
12  writeln("Escriba el valor de y2: ");
13  read(y2);
14  m := (y2 - y1) / (x2 - x1);
15  writeln("La pendiente es: ", m);
16  end.
```

Se muestra un error y no te permite asignarle un valor a la variable redeclarada.

Error: La variable y1 ya sido declarada

```
Procesando instrucción 1: var x1 real ;

Procesando instrucción 2: var x2 real ;

Procesando instrucción 3: var y1 real ;

Procesando instrucción 4: var y1 integer ;
Error: la Variable "y1" ya ha sido declarada

Procesando instrucción 5: var y2 real ;

Procesando instrucción 6: var m real ;

Procesando instrucción 7: writeln ( "Escriba el valor de x1: " ) ;
Escriba el valor de x1:

Procesando instrucción 8: read ( x1 ) ;
Input para variable x1:
```

```
Procesando instrucción 9: write ( "Escriba el valor de x2: " ) ;
Escriba el valor de x2:
Procesando instrucción 10: read ( x2 ) ;
Input para variable x2: 10
Variable x2 asignada con valor "10.0"

Procesando instrucción 11: write ( "Escriba el valor de y1: " ) ;
Escriba el valor de y1:
Procesando instrucción 12: read ( y1 ) ;
Input para variable y1: 20
Variable y1 asignada con valor "20.0"

Procesando instrucción 13: write ( "Escriba el valor de y2: " ) ;
Escriba el valor de y2:
Procesando instrucción 14: read ( y2 ) ;
Input para variable y2: 30
Variable y2 asignada con valor "30.0"
```

- **Implementa write() y writeln()**

El compilador muestra el salto de línea con writeln

```
Procesando instrucción 7: write ( "Escriba el valor de x1: " ) ;
Escriba el valor de x1:
Procesando instrucción 8: read ( x1 ) ;
Input para variable x1: 5
Variable x1 asignada con valor "5.0"

Procesando instrucción 9: writeln ( "Escriba el valor de x2: " ) ;
Escriba el valor de x2:

Procesando instrucción 10: read ( x2 ) ;
Input para variable x2: 10
Variable x2 asignada con valor "10.0"
```

Se genera el código ensamblador para la arquitectura RiscV correspondiente para ambos casos.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

# write("Escriba el valor de x1: ")
  la a0, str_0  # Cargar dirección de la cadena
  li a7, 4      # Syscall para Write('texto')
  ecall         # Ejecutar syscall


# read(x1)
  li a7, 6      # Syscall para Read(real)
  ecall         # Ejecutar syscall
  fmv.s ft0, fa0  # Mover valor leído a x1


# writeln("Escriba el valor de x2: ")
  la a0, str_1  # Cargar dirección de la cadena
  li a7, 4      # Syscall para Write('texto')
  ecall         # Ejecutar syscall
  li a0, 10     # ASCII para newline
  li a7, 11     # Syscall para WriteLn(carácter)
  ecall         # Ejecutar syscall


# read(x2)
  li a7, 6      # Syscall para Read(real)
  ecall         # Ejecutar syscall
  fmv.s ft1, fa0  # Mover valor leído a x2
```

- **Implementa read()**

El compilador implementa read(), lee un valor desde la entrada y lo almacena en la variable indicada.

```
Procesando instrucción 6: var m real ;
Variable m declarada como real

Procesando instrucción 7: write ( "Escriba el valor de x1: " ) ;
Escriba el valor de x1:
Procesando instrucción 8: read ( x1 ) ;
Input para variable x1:
```

Lee los valores y los asigna a la variable

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Variable y2 declarada como real

Procesando instrucción 5: var m real ;
Variable m declarada como real

Procesando instrucción 6: write ( "Escriba el valor de x1: " ) ;
Escriba el valor de x1:
Procesando instrucción 7: read ( x1 ) ;
Input para variable x1: 56
Variable x1 asignada con valor "56.0"

Procesando instrucción 8: writeln ( "Escriba el valor de x2: " ) ;
Escriba el valor de x2:

Procesando instrucción 9: read ( x2 ) ;
Input para variable x2: 80
Variable x2 asignada con valor "80.0"

Procesando instrucción 10: write ( "Escriba el valor de y1: " ) ;
Escriba el valor de y1:
Procesando instrucción 11: read ( y1 ) ;
Input para variable y1: 144
Variable y1 asignada con valor "144.0"

Procesando instrucción 12: writeln ( "Escriba el valor de y2: " ) ;
Escriba el valor de y2:

Procesando instrucción 13: read ( y2 ) ;
Input para variable y2: 192
Variable y2 asignada con valor "192.0"

Procesando instrucción 14: m := ( y2 - y1 ) / ( x2 - x1 ) ;
Expresión: m := ['(', 'y2', '-', 'y1', ')', '/', '(', 'x2', '-', 'x1', ')']
Variable m asignada con valor 2.0

Procesando instrucción 15: writeln ( "La pendiente es: " , m ) ;
La pendiente es: 2.0
```

Comprobamos que el valor de la variable se almacenó.

```
--------------------------------- TABLA FINAL DE VARIABLES ---------------------------------

 Tabla de variables
nombre          tipo            registro asignado       valor
x1              real            ft0             56.0
x2              real            ft1             80.0
y1              real            ft2             144.0
y2              real            ft3             192.0
m               real            ft4             2.0
t1              real            ft5             48.0
t2              real            ft6             24.0
```

- **Ejecuta expresiones**

El compilador es capaz de manejar y evaluar expresiones. ***Ej:*** $m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

--------------------------------- TABLA FINAL DE VARIABLES ---------------------------------

 Tabla de variables
nombre          tipo            registro asignado       valor
x1              real            ft0             5.0
x2              real            ft1             10.0
y1              real            ft2             20.0
y2              real            ft3             30.0
m               real            ft4             2.0
t1              real            ft5             10.0
t2              real            ft6             5.0

--------------------------------- CÓDIGO INTERMEDIO ---------------------------------
# m := ( y2 - y1 ) / ( x2 - x1 )
  t1 = y2 - y1
  t2 = x2 - x1
  t3 = t1 / t2
```

Se genera el código ensamblador para la arquitectura RiscV.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

  li a7, 4        # Syscall para Write('texto')
  ecall           # Ejecutar syscall
  li a0, 10       # ASCII para newline
  li a7, 11       # Syscall para WriteLn(carácter)
  ecall           # Ejecutar syscall


# read(y2)
  li a7, 6        # Syscall para Read(real)
  ecall           # Ejecutar syscall
  fmv.s ft3, fa0  # Mover valor leído a y2


# m := ( y2 - y1 ) / ( x2 - x1 )
  fsub.s ft5, ft3, ft2  # ft5 = ft3 - ft2
  fsub.s ft6, ft1, ft0  # ft6 = ft1 - ft0
  fdiv.s ft7, ft5, ft6  # ft7 = ft5 / ft6
  fmv.s ft4, ft7  # m = resultado


# writeln("La pendiente es: ", m)
  la a0, str_4  # Cargar dirección de la cadena
  li a7, 4        # Syscall para Write('texto')
  ecall           # Ejecutar syscall
  fmv.s fa0, ft4  # Cargar valor de m
  li a7, 2        # Syscall para Write(real)
  ecall           # Ejecutar syscall
  li a0, 10       # ASCII para newline
  li a7, 11       # Syscall para WriteLn(carácter)
  ecall           # Ejecutar syscall


  # Exit
  li a7, 10       # Syscall para exit
  ecall           # Ejecutar syscall
```

- **Implementa ciclos FOR**

for.pas

```
1   var c integer;
2   var a integer;
3
4   begin
5     a := 10;
6     for c := 1 to 5 do
7     begin
8       a := a * 2;
9       writeln(a);
10    end;
11  end.
```

```
-------------------------------- ANÁLISIS SINTÁCTICO Y EJECUCIÓN --------------------------------
Instrucción 1: ['var', 'c', 'integer', ';']
Instrucción 2: ['var', 'a', 'integer', ';']
Instrucción 3: ['begin']
Instrucción 4: ['a', ':=', '10', ';']
Instrucción 5: ['for', 'c', ':=', '1', 'to', '5', 'do', 'begin', 'a', ':=', 'a', '*', '2', ';', 'writeln', '(', 'a', ')', ';', 'end']
Instrucción 6: [';']
Instrucción 7: ['end.']

Procesando instrucción 1: var c integer ;

Procesando instrucción 2: var a integer ;

Procesando instrucción 3: begin
```

El compilador procesa cada instrucción y muestra en la consola el resultado del bucle for.

```
Procesando instrucción 3: begin

Procesando instrucción 4: a := 10 ;
Expresión: a := ['10']
Variable a asignada con valor 10

Procesando instrucción 5: for c := 1 to 5 do begin a := a * 2 ; writeln ( a ) ; end

Simulando for c desde 1 hasta 5:
20
40
80
160
320

Procesando instrucción 6: ;

Procesando instrucción 7: end.
Programa terminado.
```

Se almacenan en la tabla de variables los resultados y se genera el respectivo código ensamblador.

```
--------------------------------- TABLA FINAL DE VARIABLES ---------------------------------

 Tabla de variables
nombre          tipo            registro asignado       valor
c               integer         t0              5
a               integer         t1              320

--------------------------------- CÓDIGO ENSAMBLADOR RISC-V ---------------------------------
.text
.globl main
main:

# for c := 1 to 5 do
  li t0, 1  # c := 1
  loop_start_0:
  li t7, 5
  bgt t0, t7, loop_end_0  # if c > 5 goto end

# a := a * 2 (dentro del FOR)
  li t6, 2
  mul t2, t1, t6  # t2 = t1 * 2
  mv t1, t2  # a = resultado

# for(a) (dentro del FOR)
  mv a0, t1  # Cargar valor de a
  li a7, 1        # Syscall para Write(integer)
  ecall           # Ejecutar syscall
  li a0, 10       # ASCII para newline
  li a7, 11       # Syscall para WriteLn(carácter)
  ecall           # Ejecutar syscall
  addi t0, t0, 1  # c++
  j loop_start_0  # jump to loop start
  loop_end_0:


  # Exit
  li a7, 10       # Syscall para exit
  ecall           # Ejecutar syscall
```

- **Implementa las funciones sin(), cos(), tan()**

miniCompilador.py | main.pas | for.pas | expresiones.pas

expresiones.pas

```
1   var x1 real;
2   var x2 real;
3   var r_cos real;
4   var r_sen real;
5   var r_tan real;
6
7   begin
8   x1 := 0.5;
9   x2 := 0.3;
10
11  r_cos := cos(x1 + x2); (*Calcula el coseno de la suma de x1 y x2*)
12  r_sen := sin(x1 + x2);  (*Calcula el seno de la suma de x1 y x2*)
13  r_tan := tan(x1 + x2); (*Calcula la tangente de la suma de x1 y x2*)
14  end.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
------------------------------ ANÁLISIS SINTÁCTICO Y EJECUCIÓN ------------------------------
Instrucción 1: ['var', 'x1', 'real', ';']
Instrucción 2: ['var', 'x2', 'real', ';']
Instrucción 3: ['var', 'r_cos', 'real', ';']
Instrucción 4: ['var', 'r_sen', 'real', ';']
Instrucción 5: ['var', 'r_tan', 'real', ';']
Instrucción 6: ['begin']
Instrucción 7: ['x1', ':=', '0.5', ';']
Instrucción 8: ['x2', ':=', '0.3', ';']
Instrucción 9: ['r_cos', ':=', 'cos', '(', 'x1', '+', 'x2', ')', ';']
Instrucción 10: ['r_sen', ':=', 'sin', '(', 'x1', '+', 'x2', ')', ';']
Instrucción 11: ['r_tan', ':=', 'tan', '(', 'x1', '+', 'x2', ')', ';']
Instrucción 12: ['end.']

Procesando instrucción 1: var x1 real ;

Procesando instrucción 2: var x2 real ;

Procesando instrucción 3: var r_cos real ;

Procesando instrucción 4: var r_sen real ;

Procesando instrucción 5: var r_tan real ;

Procesando instrucción 6: begin

Procesando instrucción 7: x1 := 0.5 ;
Expresión: x1 := ['0.5']
Variable x1 asignada con valor 0.5

Procesando instrucción 8: x2 := 0.3 ;
Expresión: x2 := ['0.3']
Variable x2 asignada con valor 0.3

Procesando instrucción 9: r_cos := cos ( x1 + x2 ) ;
Variable r_cos asignada con valor 0.6967067093471654

Procesando instrucción 10: r_sen := sin ( x1 + x2 ) ;
Variable r_sen asignada con valor 0.7173560908995228

Procesando instrucción 11: r_tan := tan ( x1 + x2 ) ;
Variable r_tan asignada con valor 1.0296385570503641

Procesando instrucción 12: end.
Programa terminado.
```

Se almacenan las variables con sus respectivos valores, y se muestra el código.

```
-------------------------------- TABLA FINAL DE VARIABLES --------------------------------

 Tabla de variables
nombre          tipo            registro asignado       valor
x1              real            ft0             0.5
x2              real            ft1             0.3
r_cos           real            ft2             0.6967067093471654
r_sen           real            ft3             0.7173560908995228
r_tan           real            ft4             1.0296385570503641
```

```
-------------------------------- CÓDIGO ENSAMBLADOR RISC-V --------------------------------
.text
.globl main
main:

# r_cos := cos(x1+x2)
  fadd.s ft5, ft0, ft1  # ft5 = ft0 + ft1
  fmv.s fa0, ft5  # pasar arg a fa0
  call cos              # llamar a cos
  fmv.s ft2, fa0     # guardar resultado en r_cos


# r_sen := sin(x1+x2)
  fadd.s ft6, ft0, ft1  # ft6 = ft0 + ft1
  fmv.s fa0, ft6  # pasar arg a fa0
  call sin              # llamar a sin
  fmv.s ft3, fa0     # guardar resultado en r_sen
```

```
# r_tan := tan(x1+x2)
  fadd.s ft7, ft0, ft1  # ft7 = ft0 + ft1
  fmv.s fa0, ft7  # pasar arg a fa0
  call tan              # llamar a tan
  fmv.s ft4, fa0     # guardar resultado en r_tan


  # Exit
  li a7, 10       # Syscall para exit
  ecall           # Ejecutar syscall
```

Capturas extras del funcionamiento

- **Separa en tokens**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

El programa es correcto, no faltan ";"

Tokens encontrados:
var -> Palabra reservada
x1 -> ID
real -> Tipo
; -> Simbolo especial
var -> Palabra reservada
x2 -> ID
real -> Tipo
; -> Simbolo especial
var -> Palabra reservada
y1 -> ID
real -> Tipo
; -> Simbolo especial
var -> Palabra reservada
y2 -> ID
real -> Tipo
; -> Simbolo especial
var -> Palabra reservada
m -> ID
real -> Tipo
; -> Simbolo especial
write -> Procedimiento estandar
( -> Simbolo especial
"Escriba el valor de x1: " -> Cadena
) -> Simbolo especial
; -> Simbolo especial
read -> Procedimiento estandar
( -> Simbolo especial
x1 -> ID
) -> Simbolo especial
; -> Simbolo especial
writeln -> Procedimiento estandar
( -> Simbolo especial
"Escriba el valor de x2: " -> Cadena
) -> Simbolo especial
; -> Simbolo especial
read -> Procedimiento estandar
( -> Simbolo especial
x2 -> ID
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

) -> Simbolo especial
; -> Simbolo especial
write -> Procedimiento estandar
( -> Simbolo especial
"Escriba el valor de y1: " -> Cadena
) -> Simbolo especial
; -> Simbolo especial
read -> Procedimiento estandar
( -> Simbolo especial
y1 -> ID
) -> Simbolo especial
; -> Simbolo especial
writeln -> Procedimiento estandar
( -> Simbolo especial
"Escriba el valor de y2: " -> Cadena
) -> Simbolo especial
; -> Simbolo especial
read -> Procedimiento estandar
( -> Simbolo especial
y2 -> ID
) -> Simbolo especial
; -> Simbolo especial
m -> ID
:= -> Operador
( -> Simbolo especial
y2 -> ID
- -> Operador
y1 -> ID
) -> Simbolo especial
/ -> Operador
( -> Simbolo especial
x2 -> ID
- -> Operador
x1 -> ID
) -> Simbolo especial
; -> Simbolo especial
writeln -> Procedimiento estandar
( -> Simbolo especial
"La pendiente es: " -> Cadena
, -> Simbolo especial
m -> ID
```

```
m -> ID
) -> Simbolo especial
; -> Simbolo especial
end -> Palabra reservada
; -> Simbolo especial
```

- **Genera código intermedio y es capaz de manejar variables temporales**

```
-------------------------------- CÓDIGO INTERMEDIO --------------------------------
# m := ( y2 - y1 ) / ( x2 - x1 )
  t1 = y2 - y1
  t2 = x2 - x1
  t3 = t1 / t2
```

```
-------------------------------- TABLA FINAL DE VARIABLES --------------------------------

 Tabla de variables
nombre          tipo            registro asignado       valor
x1              real            ft0             2.0
x2              real            ft1             5.0
y1              real            ft2             3.0
y2              real            ft3             7.0
m               real            ft4             1.3333333333333333
t1              real            ft5             4.0
t2              real            ft6             3.0
```

¿Cómo funciona el Compilador?

Análisis Léxico: La función tokenizar lee el código fuente y lo divide en un flujo de unidades significativas llamadas "tokens". También realiza una comprobación básica de punto y coma faltantes.

Análisis Sintáctico y Semántico (y Generación de Código): Las funciones dividir_en_instrucciones y procesar_instruccion trabajan juntas para analizar el flujo de tokens, identificar el tipo de cada instrucción (declaración, asignación, read, write, for) y realizar comprobaciones semánticas (por ejemplo, existencia de variables, compatibilidad de tipos durante la evaluación). De manera simultánea:

- **Simula la Ejecución:** Para los bucles read, write, writeln y for, el código ejecuta directamente las operaciones y actualiza tabla_var para reflejar los cambios en los valores de las variables. Esto proporciona una "retroalimentación" inmediata en tiempo de ejecución.
- **Genera Código Intermedio:** Para las expresiones, las convierte a notación postfija y genera código de tres direcciones.
- **Genera Código Ensamblador RISC-V:** Para cada operación soportada, produce las instrucciones RISC-V correspondientes, incluyendo el manejo de la asignación de registros.

Salida: Finalmente, el compilador imprime la tabla de símbolos, el código intermedio generado y el código ensamblador RISC-V completo, listo para ser ensamblado y ejecutado por un simulador RISC-V.

Explicación del código

Clase Variable

Inicializa un objeto Variable con nombre, tipo, valor (valor actual) y registro. Esta clase actúa como la estructura para las entradas en la tabla de símbolos.

```
class Variable:
    def __init__(self, nombre, tipo, valor, registro):
        self.nombre = nombre
        self.tipo = tipo
        self.valor = valor
        self.registro = registro
```

Listas y Diccionarios Globales

registros_real, registros_int, registros_char, registros_string: Listas de registros RISC-V disponibles para diferentes tipos de datos. Se utilizan para la asignación de registros.

```
registros_real = [f"ft{i}" for i in range(32)]
registros_int = ["t0", "t1", "t2", "t3", "t4", "t5", "t6", "a0", "a1", "a2",
"a3", "a4", "a5", "a6", "a7"]
registros_char = ["x0", "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8", "x9"]
registros_string = ["s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7"]
```

memoria_spill: Lista para mantener las variables a las que no se les pudo asignar un registro (desbordadas a memoria). Esto es un marcador de posición para una asignación de registros más avanzada.

```
memoria_spill = []
```

tabla_var: Tabla de símbolos que almacena información sobre las variables declaradas.

```
tabla_var = []
```

codigo_intermedio_total: Acumula el código intermedio generado.

codigo_ensamblador_total: Acumula el código ensamblador RISC-V generado.

seccion_data: Almacena directivas de la sección de datos para el código ensamblador, principalmente para cadenas.

```
# variables para acumular el código
codigo_intermedio_total = []
codigo_ensamblador_total = []
seccion_data = []  # para las cadenas
```

estado_for: Un diccionario para gestionar el estado de los bucles FOR durante el análisis sintáctico y la generación de código, incluyendo si un bucle FOR está activo, su variable de bucle, valores iniciales/finales y las instrucciones acumuladas dentro de su cuerpo.

```
estado_for = {'activo': False} # variable para el estado del FOR
```

Funciones de Gestión de Registros

reset_registros(): Reinicia todas las listas de registros y tabla_var, memoria_spill y estado_for a sus estados iniciales. Esto es crucial para compilar varios archivos o reiniciar el estado del compilador.

```
def reset_registros(): #reiniciamos los registros
    global registros_real, registros_int, registros_char, registros_string,
memoria_spill, tabla_var, estado_for
    registros_real = [f"ft{i}" for i in range(32)]
    registros_int = ["t0", "t1", "t2", "t3", "t4", "t5", "t6", "a0", "a1",
"a2", "a3", "a4", "a5", "a6", "a7"]
    registros_char = ["x0", "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8",
"x9"]
    registros_string = ["s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7"]
    memoria_spill = []
    tabla_var = []
    estado_for = {'activo': False}
```

asignar_registro(tipo): Asigna un registro disponible según el tipo dado. Si no hay ningún registro disponible, devuelve una ubicación de memoria "SPILL" (un mecanismo de desbordamiento simplista).

```
def asignar_registro(tipo):
    if tipo == "real":  # float en Pascal
```

asignar_registro_temporal(tipo): Similar a asignar_registro pero específicamente para registros temporales utilizados durante la evaluación de expresiones.

```
def asignar_registro_temporal(tipo): #Asigna un registro temporal
    global registros_real, registros_int
    if tipo == "real":
```

liberar_registro_temporal(registro, tipo): Vuelve a añadir un registro temporal a su lista de registros disponibles respectiva si no es una ubicación de memoria desbordada.

```
def liberar_registro_temporal(registro, tipo):
    global registros_real, registros_int
    if "SPILL" not in registro:
        if tipo == "real" and registro.startswith("ft"):
            registros_real.append(registro)
        elif registro in ["t0", "t1", "t2", "t3", "t4", "t5", "t6", "a0",
"a1", "a2", "a3", "a4", "a5", "a6", "a7"]:
            registros_int.append(registro)
```

Funciones de Gestión de la Tabla de Símbolos

agregar_var(tabla_var, nombre, tipo): Añade una nueva variable a tabla_var. Comprueba si hay errores de redeclaración y asigna un registro usando asignar_registro.

```
#Agregar la nueva variable a la tabla
def agregar_var(tabla_var, nombre, tipo):
    if existe_var(tabla_var, nombre):
        print(f'Error: la Variable "{nombre}" ya ha sido declarada')
```

existe_var(tabla_var, nombre): Comprueba si una variable con el nombre dado ya existe en tabla_var.

```
#checa si la variable existe en la tabla
def existe_var(tabla_var, nombre):
    for v in tabla_var:
```

set_var(tabla_var, nombre, valor): Establece el valor de una variable existente en tabla_var.

```
#si existe la variable en la tabla, asigna el valor al campo "valor"
def set_var(tabla_var, nombre, valor):
    if existe_var(tabla_var, nombre):
```

get_valor(tabla_var, varNombre): Recupera el valor de una variable de tabla_var.

```
# recupera el valor de la variable por medio de su nombre
def get_valor(tabla_var, varNombre):
    for v in tabla_var:
        if (v.nombre == varNombre):
            return v.valor
```

get_registro_var(tabla_var, nombre): Recupera el registro asignado para una variable de tabla_var.

```
# obtiene el registro que se le asigno a una variable
def get_registro_var(tabla_var, nombre):
    for v in tabla_var:
        if v.nombre == nombre:
            return v.registro
    return None
```

get_tipo_var(tabla_var, nombre): Recupera el tipo de una variable de tabla_var.

```
def get_tipo_var(tabla_var, nombre): #obtener el tipo de la variable
    for v in tabla_var:
        if v.nombre == nombre:
            return v.tipo
    return None
```

imprime_tabla_var(tabla_var): Imprime el contenido actual de la tabla de símbolos de forma formateada.

```
# imprime el contenido de la tabla
def imprime_tabla_var(tabla_var):
    print()
    print(' Tabla de variables')
    print('nombre\t\ttipo\t\tregistro asignado\tvalor')
    for v in tabla_var:
        print(f'{v.nombre}\t\t{v.tipo}\t\t{v.registro}\t\t{v.valor}')
    return None
```

Análisis Léxico (Tokenización)

- Elimina los comentarios de estilo Pascal (* … *)
- **Comprobación de Errores:** Incluye una comprobación básica de punto y coma faltantes al final de las líneas.
- Divide el texto de entrada en una lista de tokens, manejando identificadores, palabras clave, números (incluidos los decimales), cadenas y símbolos especiales como :=.

```
def get_etiqueta(token):
    if es_operador(token):
        return 'Operador'
    elif es_simbolo_especial(token):
        return 'Simbolo especial'
    elif es_palabra_reservada(token):
        return 'Palabra reservada'
    elif es_procedimiento(token):
        return 'Procedimiento estandar'
    elif es_tipo(token):
        return 'Tipo'
    elif es_cadena(token):
        return 'Cadena'
```

```
    elif es_constante(token):
        return 'Constante'
    elif es_id(token):
        return 'ID'
    else:
        return 'Desconocido'
```

```
# Eliminar comentarios tipo (* ... *)
def tokenizar(texto):
    estado = 'Z'
    texto_sin_comentarios = ''
    i = 0
```

Procesamiento de Expresiones

precedencia: Un diccionario que define la precedencia de los operadores aritméticos para la conversión de infijo a postfijo.

```
precedencia = {'+': 1, '-': 1,'*': 2,'/': 2,'(': 0}
```

infija_a_postfija(tokens): Convierte una expresión infija (lista de tokens) a postfijo (notación polaca inversa) utilizando un algoritmo de shunting-yard.

```
def infija_a_postfija(tokens):
    salida = []
    pila = []
    for token in tokens:
```

infija_a_prefija(tokens): Convierte una expresión infija a notación prefija (invirtiendo, convirtiendo a postfijo y volviendo a invertir).

```
def infija_a_prefija(tokens):
    tokens = tokens[::-1]
    for i in range(len(tokens)):
```

evaluar_expresion(expresion, tabla_var): Evalúa una expresión aritmética dada en notación infija. Reemplaza los nombres de las variables por sus valores actuales de tabla_var y luego usa eval() de Python para el cálculo. Incluye manejo de errores para variables no declaradas o problemas de evaluación.

```
def evaluar_expresion(expresion, tabla_var): #procesa una expresión aritmética
    try:
        expr_evaluada = []
```

Generación de Código Intermedio y Ensamblador

codigoInterEnsambla(posfija, tabla_var): Utiliza un enfoque basado en pilas, procesando operandos y operadores.o Para cada operación, genera una línea de código intermedio *(por ejemplo, t1 = op1 + op2)*. Luego, genera las instrucciones RISC-V correspondientes. Maneja valores inmediatos (constantes) y registros de variables. Gestiona los registros temporales y actualiza mapeo temporales para realizar un

seguimiento de sus registros asignados. E intenta asignar valores a variables temporales dentro de tabla_var para la simulación.

```
#Generamos el código intermedio y ensamblador RISC-V desde notación postfija
def codigoInterEnsambla(posfija, tabla_var):
    codigo_intermedio = []
    codigo_ensamblador = []
    pila = []
    cont = 1
    mapeo_temporales = {}  # Para checar qué registro tiene cada temporal
    num_operadores = sum(1 for t in posfija if es_operador(t) and t != ':=')
    operador_actual = 0
```

mostrar_codigo_con_correspondencia(codigo_intermedio_total, codigo_ensamblador_total): Muestra el código intermedio generado y su traducción correspondiente en ensamblador RISC-V uno al lado del otro, mejorando la legibilidad para la depuración y la comprensión.

```
def mostrar_codigo_con_correspondencia(codigo_intermedio_total,
codigo_ensamblador_total):
    i_int = 0
    i_asm = 0
```

Generación de Código de Llamadas al Sistema RISC-V

generar_codigo_read(variable, tabla_var): Genera código ensamblador RISC-V para el procedimiento read. Determina el tipo de la variable y utiliza la syscall apropiada (li a7, X) y el registro (a0 para enteros/caracteres, fa0 para reales) para la entrada.

```
#Generamos código RISC-V para la instrucción read
def generar_codigo_read(variable, tabla_var):
    codigo = []
    tipo = get_tipo_var(tabla_var, variable)
    registro = get_registro_var(tabla_var, variable)
```

generar_codigo_write_mejorado(argumentos, tabla_var): Genera código ensamblador RISC-V para el procedimiento write. Maneja literales de cadena añadiéndolos a la sección .data y luego cargando sus direcciones. Para las variables, determina el tipo y utiliza la syscall correcta para imprimir enteros, reales, caracteres o cadenas.

```
#Generamos código RISC-V para write
def generar_codigo_write_mejorado(argumentos, tabla_var):
    global seccion_data
    codigo = []
```

generar_codigo_writeln_mejorado(argumentos, tabla_var): Genera código ensamblador RISC-V para el procedimiento writeln. Llama a generar_codigo_write_mejorado y luego añade instrucciones para imprimir un carácter de nueva línea.

```
#Generamos código RISC-V para writeln
def generar_codigo_writeln_mejorado(argumentos, tabla_var):
    codigo = generar_codigo_write_mejorado(argumentos, tabla_var)

    # Agregar salto de línea
    codigo.append("li a0, 10        # ASCII para newline")
    codigo.append("li a7, 11        # Syscall para WriteLn(carácter)")
    codigo.append("ecall            # Ejecutar syscall")
    return codigo
```

Procesamiento y Simulación de Instrucciones

extraer_instrucciones_begin_end(tokens): Extrae instrucciones individuales de un bloque begin-end, respetando las estructuras anidadas.

```
def extraer_instrucciones_begin_end(tokens): #Extrae todas las instrucciones
individuales entre begin y end

    if not tokens or tokens[0].lower() != 'begin':

        return []
```

procesar_instruccion(tokens, tabla_var): Es la función principal del análisis sintáctico y semántico.

- **Declaración de Variables (var)**: Maneja las declaraciones var (id) (type) , añadiendo variables a tabla_var.
- **Read (read)**: Procesa las sentencias read(var), genera código RISC-V y solicita al usuario una entrada.
- **Bucle For (for)**: Analiza los bucles for <var> := <start> to <end> do, genera el código ensamblador RISC-V completo para el bucle:, y *simula* la ejecución del bucle for iterando desde start hasta end. Maneja tanto bucles for de una sola sentencia como bloques for...begin...end.
- **Asignación (:=)**: Analiza var := expresion, convierte la expresión a postfija usando infija_a_postfija, llama a codigoInterEnsambla para generar código intermedio y ensamblador, genera código RISC-V, evalúa la expresión durante la simulación y actualiza el valor de la variable en tabla_var.
- **Funciones Matemáticas (sin, cos, tan)**: Maneja estas tres funciones, generando llamadas RISC-V y realizando el cálculo durante la simulación.
- **Write/Writeln (write, writeln)**: Analiza las sentencias write y writeln, extrae los cadenas o nombres de variables, genera código RISC-V usando generar_codigo_write_mejorado o generar_codigo_writeln_mejorado, e imprime la salida en la consola durante la simulación.

- **Begin/End (begin, end)**: Gestiona los niveles de anidamiento de los bloques begin…end para los bucles for, devuelve True para continuar el procesamiento, False para end; o end. para indicar la terminación del programa.

```
def procesar_instruccion(tokens, tabla_var): #Manejo del for y de todas las
funcionalidades del compilador

    global codigo_intermedio_total, codigo_ensamblador_total, seccion_data,
estado_for
```

procesar_instruccion_sin_for(tokens, tabla_var): Se usa *solo* durante la simulación de los cuerpos de los bucles FOR. Realiza las acciones semánticas (como la asignación de variables o la impresión) pero *no* genera código intermedio o ensamblador adicional, ya que ese código ya se genera una vez cuando se procesa el propio bucle FOR. Esto evita la generación de código duplicado para las iteraciones de bucle simuladas.

```
# procesamos instrucciones durante la simulación del FOR

def procesar_instruccion_sin_for(tokens, tabla_var):

    if not tokens:

        return True
```

dividir_en_instrucciones(tokens):Analiza la lista de tokens en una lista de instrucciones individuales. Esta función es fundamental para un procesamiento secuencial, agrupa correctamente los bucles for (incluidos sus bloques begin…end).

```
#Divide tokens en instrucciones

def dividir_en_instrucciones(tokens):

    instrucciones = []

    instruccion_actual = []

    i = 0
```

simular_for(var_loop, inicio, final, instrucciones_cuerpo, tabla_var):simula la ejecución de un bucle FOR. Itera a través del rango especificado, actualiza la variable del bucle y luego llama a procesar_instruccion_sin_for para cada instrucción en el cuerpo del bucle.

```
def simular_for(var_loop, inicio, final, instrucciones_cuerpo, tabla_var):
    for i in range(inicio, final + 1):
        set_var(tabla_var, var_loop, i)
        print(f"  Iteración {i}: {var_loop} = {i}")
```

Flujo Principal de Compilación

compilar_archivo(ruta_archivo): Es el punto de entrada para el compilador. Reinicia todos los estados globales usando reset_registros(), lee el contenido del archivo Pascal especificado, llama a tokenizar para el análisis léxico y divide los tokens en instrucciones lógicas. Itera a través de cada instrucción, llamando a procesar_instruccion para realizar el análisis sintáctico, el análisis semántico y la generación de código.
Después de procesar todas las instrucciones, imprime:

- El estado final de la tabla de símbolos (imprime_tabla_var).
- El código intermedio acumulado.
- El código ensamblador RISC-V acumulado, incluyendo la sección .data (para cadenas) y la sección .text con el punto de entrada main y la syscall de salida del programa.

```
def compilar_archivo(ruta_archivo):
    global codigo_intermedio_total, codigo_ensamblador_total, seccion_data
    # Reiniciar variables
    codigo_intermedio_total = []
    codigo_ensamblador_total = []
    seccion_data = []
```