Proyecto de Compiladores

Compilador de BASIC-R

Entrega final

Grupo: 21

Alumnos: Ricardo Catalinas Jiménez (M050028)
Mauricio César Togneri (L040194)
Jorge Mozos Arias (L040134)

Índice:

1. Analizador Léxico:

1.1 Lenguaje a reconocer:

El lenguaje que el analizador léxico deberá reconocer es un subconjunto de BASIC-R, el cual posee las siguientes características:

- La estructura general de un programa compuesto por procedimientos e instrucciones del programa principal.
- Procedimientos con paso de parámetros por valor.
- Tipos enteros y lógicos.
- Variables enteras y su declaración.
- Constantes enteras y cadenas de caracteres.
- Sentencias: asignación, condicional simple y llamada a procedimientos.
- Expresiones.
- Comentarios.
- Operaciones de entrada/salida por terminal:
    - PRINT
    - INPUT
- Operadores:
    - Aritméticos: +, -, *, /, ^
    - Relacionales: =, <>, <, >, <=, >=
    - Lógicos: AND, OR, NOT

Además, nuestro compilador implementa:

- Tipos de datos:
    - Cadenas
- Sentencias:
    - Sentencia repetitiva (WHILE-WEND)
- Procedimientos:
    - Subprogramas (SUB)

Donde las palabras reservadas son:

END	NOT	AND	OR	TRUE	FALSE
STATIC	PRINT	INPUT	LET	CALL	IF
THEN	ELSE	WHILE	WEND	SUB	

1.2 Implementación:

El analizador léxico es el módulo del compilador encargado de reconocer y devolver los token pertenecientes al fichero fuente. Para ello, lee dicho fichero carácter a carácter hasta encontrar un token. En el proceso de reconocimiento de los token, el analizador léxico debe ser capaz de detectar y filtrar la información innecesaria para el proceso de compilación (comentarios, espacios en blanco, saltos de línea, tabulaciones, etc). Además debe ser capaz de detectar e informar de los posibles errores que surjan al leer los tokens. El lenguaje que hemos utilizado para desarrollar el analizador léxico ha sido Python y la herramienta que hemos utilizando ha sido Lex.

1.3 Tipos de Tokens:

Para poder interactuar con el analizador sintáctico, se ha diseñado la siguiente codificación de los tokens que devolverá el analizador léxico:

1	<SUMA, '+'>	Operador aritmético de suma
2	<RESTA, '-'>	Operador aritmético de resta
3	<MULTIPLICACIÓN, '*'>	Operador aritmético de multiplicación
4	<DIVISIÓN, '/'>	Operador aritmético de división
5	<IGUAL, '='>	Operador de asignación
6	<POTENCIA, '^'>	Operador aritmético de potencia
7	<DOS_PUNTOS, ':'>	Dos puntos
8	<COMA, ','>	Coma
9	<PUNTO_COMA, ';'>	Punto y coma
10	<ABRE_PARÉNTESIS, '('>	Abre paréntesis
11	<CIERRA_PARÉNTESIS, ')'>	Cierra paréntesis
12	<MENOR, '<'>	Comparador menor
13	<MAYOR, '>'>	Comparador mayor
14	<MENOR_IGUAL, '<='>	Comparador menor o igual
15	<MAYOR_IGUAL, '>='>	Comparador mayor o igual
16	<DISTINTO, '<>'>	Comparador distinto
17	<CADENA, valor>	Cadena de caracteres
18	<NUMERO, valor>	Número entero
19	<IDENTIFICADOR, valor>	Identificador de entero/cadena
19	<PALABRA_RESERVADA, valor>	Palabra reservada

1.4 Gramática:

La gramática del analizador léxico en formato BNF es la siguiente:

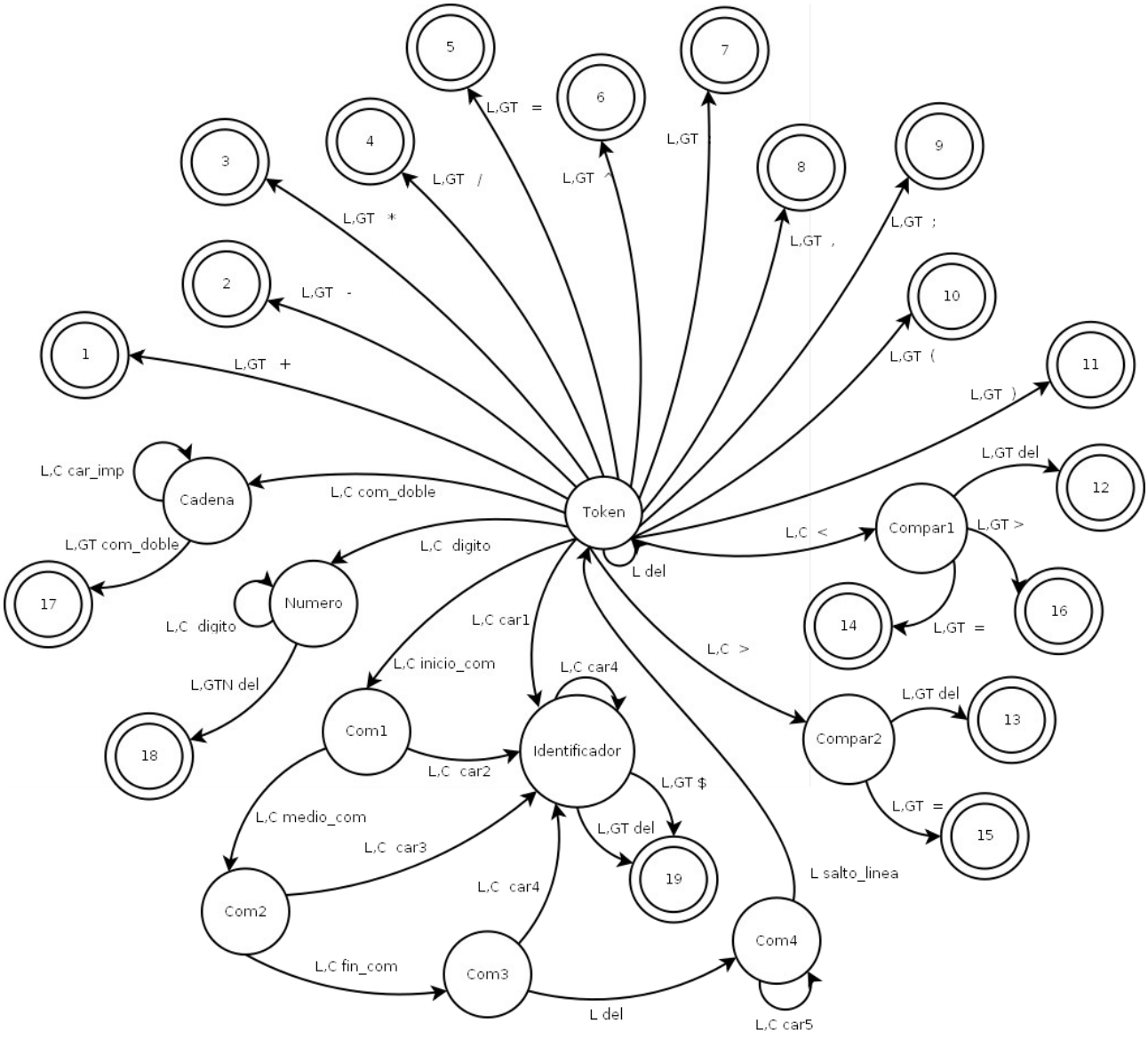
```
Token := inicio_com Com1 | car1 Identif | com_doble Cadena | digito Numero |
del Token | salto_linea | + | - | * | / | = | ^ | : | , | ; | ( | ) | <
Compar1 | > Compar2

Com1    := medio_com Com2 | car2 Identif
Com2    := fin_com Com3 | car3 Identif
Com3    := del Com4 | car4 Identif
Com4    := car5 Com4 | salto_linea Token
Identif := car4 Identif | $ | λ
Numero  := digito Numero | λ
Cadena  := car_imp Cadena | com_doble
Compar1 := > | = | λ
Compar2 := = | λ
```

Donde:

```
com_doble   => "
car_imp     => cualquier caracter - salto_linea - com_doble
del         => espacio, \t, \r
salto_linea => \n
inicio_com  => {r, R}
medio_com   => {e, E}
fin_com     => {m, M}
letra       => cualquier letra del alfabeto
digito      => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
car1        => letra - inicio_com
car2        => (letra - medio_com) U digito
car3        => (letra - fin_com) U digito
car4        => letra U digito
car5        => cualquier_caracter - salto_linea
```

1.5 Autómata finito determinista:



1.6 Errores detectados:

El analizador léxico detecta los siguientes errores:

Error	Ejemplo
ERROR_IDENTIFICADOR	123foo
CARÁCTER_INVALIDO	_ ! #
CADENA_MAL_FORMADA	cad$ = “hola
OVERFLOW	num = 45678

1.7 Acciones semánticas:

L	Leer carácter
C	Concatenar carácter
GTN	Genera token de número Int(lexema)

GT	if (lexema == PAL_RES)
	  GENERA_TOKEN (PAL_RES, lexema)
	else
	  GENERA_TOKEN (ID, valor)

2. Tabla de símbolos:

2.1 Diseño:

La tabla de símbolos es un componente del compilador encargado de almacenar todos los símbolos presentes en el programa fuente y toda la información relativa a ellos. La tabla de símbolos será utilizada por el analizador sintáctico, el analizador semántico y el generador de código intermedio. Puesto que el compilador está implementado en Python, tenemos un tipado dinámico y tablas hash integradas en el propio lenguaje, por lo que se ha optado por una implementación propia de la tabla de símbolos.

La tabla de símbolos está compuesta por una tabla hash principal en donde se encuentran las entradas correspondientes a las variables globales del programa, las variables temporales del programa principal y todos los nombres de subprogramas presentes en el fichero fuente. Por cada variable se almacena su lexema, su tipo (entero o cadena) y su desplazamiento dentro de la tabla. Por cada subprograma se almacena sus argumentos (cantidad y tipos) y una referencia a una tabla de ámbito local la cual almacenará más información vinculada al subprograma.

Cada tabla de símbolos local contendrá la información de las variables locales, temporales y los parámetros de dicho subprograma. Al igual que en la tabla de símbolos global, para cada una de estas variables se almacenará su lexema, su tipo y su desplazamiento dentro de la tabla.

Con todo esto, el compilador es capaz de saber con que variables cuenta en cada momento, dependiendo del ámbito en el que se encuentre, el tipo de estas y sus correspondientes desplazamientos para poder llevar a cabo el análisis sintáctico y semántico del programa y la posterior generación de código.

3. Analizador sintáctico:

El analizador sintáctico se encarga de comprobar que la sintaxis del código fuente es correcta. El análisis sintáctico se realizará mediante un análisis ascendente de tipo LR.

El analizador sintáctico, creado en base a la herramienta *Yacc*, le irá pidiendo *tokens* al analizador léxico y comprobará si van coincidiendo con las reglas sintácticas previamente definidas.

3.1 Gramática del lenguaje:

```
axioma ::= separador_opcional programa

programa ::= subprograma programa | cuerpo END separador_opcional

subprograma ::= SUB IDENTIFICADOR_NUMERO lista_argumentos
separador_obligatorio cuerpo END SUB separador_obligatorio

lista_argumentos ::= "(" identificador siguiente_argumento ")" | lambda

siguiente_argumento ::= "," identificador siguiente_argumento | lambda

cuerpo ::= declaracion_local cuerpo | sentencia cuerpo |
declaracion_local FIN_CUERPO | sentencia FIN_CUERPO

declaracion_local ::= STATIC identificador siguiente_declaracion
separador_obligatorio

siguiente_declaracion ::= "," identificador siguiente_declaracion |
lambda

identificador ::= IDENTIFICADOR_NUMERO | IDENTIFICADOR_CADENA

sentencia ::= sentencia_simple separador_obligatorio | sentencia_compleja
| sentencias_simples_misma_linea separador_obligatorio

sentencias_simples_misma_linea ::= sentencia_simple
sentencias_simples_misma_linea2

sentencias_simples_misma_linea2 ::= ":" sentencia_simple
sentencias_simples_misma_linea2 | ":" sentencia_simple

asignacion ::= LET identificador "=" expresion

llamada_subprograma ::= CALL IDENTIFICADOR_NUMERO lista_parametros

lista_parametros ::= "(" expresion siguiente_parametro ")" | lambda

siguiente_parametro ::= "," expresion siguiente_parametro | lambda
```

```
separador_obligatorio ::= SALTO_LINEA separador_obligatorio
MAS_SALTOS_LINEA | SALTO_LINEA NO_MAS_SALTOS_LINEA

separador_opcional ::= SALTO_LINEA separador_opcional | lambda

condicional ::= IF expresion THEN separador_obligatorio sentencia_simple
separador_obligatorio si_no

sentencia_simple ::= asignacion | llamada_subprograma | entrada | salida

sentencia_compleja ::= condicional | mientras separador_obligatorio

si_no ::= ELSE separador_obligatorio sentencia_simple
separador_obligatorio | lambda

entrada ::= INPUT identificador

salida ::= PRINT expresion siguiente_salida

siguiente_salida ::= ";" expresion siguiente_salida | lambda

mientras ::= WHILE expresion separador_obligatorio cuerpo WEND

expresion ::= NOT expresion
                | expresion AND expresion
                | expresion OR expresion
                | expresion ">" expresion
                | expresion MAYOR_IGUAL expresion
                | expresion "<" expresion
                | expresion MENOR_IGUAL expresion
                | expresion "=" expresion
                | expresion DISTINTO expresion
                | expresion "+" expresion
                | expresion "-" expresion
                | expresion "*" expresion
                | expresion "/" expresion
                | expresion "^" expresion
                | "-" expresion MENOS_UNARIO
                | "+" expresion MAS_UNARIO
                | "(" expresion ")"
                | identificador
                | constante

constante ::= NUMERO | CADENA | TRUE | FALSE



Notas: El texto entre comillas ("") significa que es un token que
devuelve el analizador léxico.
```

La gramática consta de 40 símbolos terminales y 28 símbolos no terminales.

3.2 Errores detectados:

Programa inválido, no puede haber nada después del END
Sentencia no reconocida
Expresión inválida
Se esperaba una sentencia no compuesta después del THEN
Se esperaba un salto de línea después del THEN
Se esperaba una sentencia no compuesta después del ELSE
Se esperaba un salto de línea después del ELSE
Se esperaba un WEND
Sentencia INPUT inválida
Sentencia PRINT inválida
Se esperaba un identificador para la asignación
Se esperaba una expresión para la asignación
Se esperaba un identificador para la declaración
Se esperaba un nombre de subprograma
Lista de parámetros inválida
Parámetro inválido

4. Analizador semántico:

4.1 Diseño:

Al mismo tiempo que se realiza la comprobación de la sintaxis, se comprueba la validez semántica del código fuente.

El analizador semántico se encarga, entre otras cosas, de comprobar la validez de los tipos en las expresiones, posibles conflictos entre las declaraciones de las variables y sus usos, control del flujo del programa, etc. Para realizar estas y otras tareas, el analizador semántico utiliza la información contenida en la tabla de símbolos e informará al usuario de los errores encontrados.

4.2 Errores detectados:

No se pueden definir variables locales con STATIC en el cuerpo principal del programa
No se puede definir el subprograma porque el símbolo global %s ha sido definido con anterioridad
No se puede definir la variable global %s porque existe un subprograma con el mismo nombre
No se puede definir la variable local %s porque existe un argumento del subprograma con mismo nombre
No se puede pasar como argumento a un subprograma una expresión de tipo booleano
No está definido el subprograma %s
La condición de un IF debe ser una expresión del tipo entero o booleano
La expresión de un PRINT debe ser del tipo entero o cadena
La condición de un WHILE debe ser una expresión del tipo entero o booleano
La variable %s ha sido definida previamente como local
El tipo de la expresión no coincide con el de la variable en la asignación
El número de argumentos en la llamada a %s es incorrecto
El argumento %d indice de la llamada al subprograma %s debe de ser de tipo entero
El argumento %d indice de la llamada al subprograma %s debe de ser de tipo cadena
El operador NOT se debe aplicar sobre una expresión tipo booleano
El operador unario %s se debe aplicar sobre una expresión de tipo entero
El operador %s se debe aplicar sobre expresiones de tipo booleano
El operador + se debe aplicar sobre expresiones de tipo entero o para concatenar cadenas
El operador %s se debe aplicar sobre expresiones de tipo entero
Una constante de tipo cadena debe tener como máximo una longitud de %s caracteres

Donde %s es el token que ha provoca el error semántico y %d el indice del argumento.

5. Generación de código:

5.1 Generación de código intermedio:

El código intermedio se realiza mediante códigos de 3 direcciones utilizando tercetos. Este paso a código intermedio se debe realizar una vez se ha comprobado que el código fuente es correcto léxica, sintáctica y semánticamente.

5.2 Generación de código objeto:

Una vez generado todo el código intermedio, se "traduce" cada uno de estos a su equivalente en código objeto. El lenguaje ensamblador que se ha generado es el utilizado por el ensamblador ASS 1.3.
La memoria está dividida en 3 zonas: el código ejecutable generado se sitúa en las primeras posiciones de la memoria, seguido de las variables globales del programa (enteros y cadenas) mientas que la pila se se sitúa en las últimas posiciones de la memoria.
El registro de activación esta compuesto por la dirección de retorno, el antiguo marco de pila, los parámetros del subprograma, sus variables locales y sus variables temporales.

5.3 Acceso a las variables:

Para acceder a una variable global, se utiliza la etiqueta asignada a la misma. Para acceder a una variable local, temporal o un parámetro de un subprograma se utiliza un desplazamiento en base al registro indice (marco de pila).

5.4 Uso del compilador:

El compilador se divide en los siguientes ficheros:

- compilador.py
- lexico.py
- sintactico_error.py
- sintactico_semantico.py
- codigo_objeto.py

Uso del compilador:

```
python compilador.py <fichero_de_entrada.bas>
```

Una vez compilado el fichero fuente, se genera un fichero con extensión *ens*, el cual deberá pasársele como argumento al programa que emula el ensamblador:

```
ass <fichero_de_entrada.ens>
```

Para realizar estas dos tareas a la vez se puede activar un flag (*-e*) en el compilador para llamar automáticamente al programa que emula el ensamblador:

```
python compilador.py -e <fichero_de_entrada.bas>
```

6. Casos de prueba:

6.1 Pruebas incorrectas:

Prueba 1:

```
rem prueba1.bas (incorrecta)

sub mostrar(numero)
      rem cadena mal formada
      print "El numero es:  ; numero
end sub

print "Ingrese un numero:"
input valor
call  mostrar(valor)

end
```

Prueba 2:

```
rem prueba2.bas (incorrecta)

sub sumar(a, b)
      rem caracter ilegal ?
      let resultado = a ? b
end sub

print "Ingrese un numero:"
input a

print "Ingrese un numero:"
input b

call sumar(a, b)
print "La suma es: " ; resultado

end
```

Prueba 3:

```
rem prueba3.bas (incorrecta)

sub restar(a, b)
      let resultado = a - b

      rem mala terminacion del subprograma
end su

print "Ingrese un numero:"
input a

print "Ingrese un numero:"
input b

call restar(a, b)
print "La resta es: " ; resultado

end
```

Prueba 4:

```
rem prueba4.bas (incorrecta)

sub multiplicar(a, b)
      let resultado = a * b
end sub

print "Ingrese un numero:"
input a

print "Ingrese un numero:"
rem error al escribir input
inpu b

call multiplicar(a, b)
print "La multiplicacion es: " ; resultado

end
```

Prueba 5:

```
rem prueba5.bas (incorrecta)

sub dividir(a, b)
      let resultado = a / b
end sub

print "Ingrese un numero:"
input a

print "Ingrese un numero:"
input b

rem numero de argumentos erroneo
call dividir(a)
print "La division es: " ; resultado

end
```

Prueba 6:

```
rem prueba6.bas (incorrecta)

sub elevar(a, b)
      rem error al operar con una cadena
      let resultado = a ^ b$
end sub

print "Ingrese un numero:"
input a

print "Ingrese un numero:"
input b

call elevar(a, b)
print "La elevar es: " ; resultado

end
```

6.2 Pruebas correctas:

Prueba 7:

```
rem prueba7.bas (correcta)

sub fibonacci(veces)
      static primero
      static segundo
      static auxiliar

      let primero = 0
      let segundo = 1

      if (veces > 0) then
            print "0"

      if (veces > 1) then
            print "1"

      while (veces > 2)
            print (primero + segundo)

            let auxiliar = primero
            let primero  = segundo
            let segundo  = auxiliar + segundo

            let veces = veces - 1
      wend

end sub

print "Ingrese un numero:"
input veces

print "Fibonacci:"
call fibonacci(veces)

end
```

Prueba 8:

```
rem prueba8.bas (correcta)

sub factorial(numero)
      if (numero > 1) then
            let resultado = resultado * numero

      if (numero > 1) then
            call factorial(numero - 1)
end sub

print "Ingrese un numero:"
input valor

let resultado = 1
call factorial(valor)
print "El factorial de " ; valor ; " es: " ; resultado

end
```

Prueba 9:

```
rem prueba9.bas (correcta)

sub resto(a, b)
      static resto
      let resto = a

      while (resto >= b)
            let resto = resto - b
      wend

      let resultado = resto
end sub

print "Ingrese un numero:"
input x

print "Ingrese un numero:"
input y

let resultado = 0
call resto(x, y)
print "El resto de " ; x ; "/" ; y ; " es: " ; resultado

end
```

Prueba 10:

```
rem prueba10.bas (correcta)

sub resto(a, b)
      static resto
      let resto = a

      while (resto >= b)
            let resto = resto - b
      wend

      let temporal = resto
end sub

sub primo(numero)
      static divide
      let divide = 0

      static actual
      let actual = numero - 1

      while (actual > 1)
            call resto(numero, actual)

            if (temporal = 0) then
                  let divide = divide + 1

            let actual = actual - 1
      wend

      if (divide > 0) then
            print "El numero " ; numero ; " no es primo"
      else
            print "El numero " ; numero ; " es primo"

end sub

print "Ingrese un numero:"
input valor

let temporal = 0
call primo(valor)

end
```

Prueba 11:

```
rem prueba11.bas (correcta)

sub resto(a, b)
      static resto
      let resto = a

      while (resto >= b)
            let resto = resto - b
      wend

      let temporal = resto
end sub

sub binario(numero)
      if (numero > 1) then
            call binario(numero/2)

      call resto(numero, 2)
      print temporal
end sub

print "Ingrese un numero:"
input valor
call binario(valor)

end
```

Prueba 12:

```
rem prueba12.bas (correcta)

sub factorial
      print "Ingrese un numero:"

      static numero
      input numero

      static original
      let original = numero

      static resultado
      let resultado = 1

      while (numero > 0)
            let resultado = resultado * numero
            let numero = numero - 1
      wend

      print "El factorial de " ; original ; " es " ; resultado
end sub

sub suma
      static izquierda
      static derecha

      print "Ingrese un numero:"
      input izquierda

      print "Ingrese un numero:"
      input derecha

      print "La suma de " ; izquierda ; " + " ; derecha ; " es igual a: " ;
(izquierda + derecha)
end sub

sub potencia
      static base
      static exponente

      print "Ingrese la base:"
      input base

      print "Ingrese el exponente:"
      input exponente

      print base ; " elevado a " ; exponente ; " es igual a: " ; (base ^
exponente)
end sub

sub adivinar
      static secreto
```

```
	print "Ingrese el numero secreto:"
	input secreto

	static intento
	print "Intente adivinar el numero: "
	input intento

	while (intento <> secreto)

		if (secreto > intento) then
			print "El numero ingresado en menor que el secreto"

		if (secreto < intento) then
			print "El numero ingresado es mayor que el secreto"

		print "Intente adivinar el numero: "
		input intento

	wend

	print "Enhorabuena, ha acertado!"

end sub

sub comparaciones
	print "Ingrese un numero: "
	input a
	print "Ingrese un numero: "
	input b

	if (a < b) then
		print "Comparacion (a < b):  " ; a ; " es menor que " ; b
	else
		print "Comparacion (a < b):  " ; a ; " es mayor o igual que " ; b

	if (a >= b) then
		print "Comparacion (a >= b): " ; a ; " es mayor o igual que " ; b
	else
		print "Comparacion (a >= b): " ; a ; " es menor que " ; b

	if (a > b) then
		print "Comparacion (a > b):  " ; a ; " es mayor que " ; b
	else
		print "Comparacion (a > b):  " ; a ; " es menor o igual que " ; b

	if (a <= b) then
		print "Comparacion (a <= b): " ; a ; " es menor o igual que " ; b
	else
		print "Comparacion (a <= b): " ; a ; " es mayor que " ; b

	if (a = b) then
		print "Comparacion (a = b):  " ; a ; " es igual que " ; b
	else
		print "Comparacion (a = b):  " ; a ; " es distinto que " ; b
```

```
      if (a <> b) then
            print "Comparacion (a <> b): " ; a ; " es distinto que " ; b
      else
            print "Comparacion (a <> b): " ; a ; " es igual que " ; b

end sub

sub opcionesNumero
      print ""
      print "Que operacion desea realizar?:"
      print "   1) Factorial"
      print "   2) Suma"
      print "   3) Potencia"
      print "   4) Adivinar numero"
      print "   5) Comparaciones"
      print "   6) Salir"
      print ""
end sub

sub operacionesNumero
      static operacion
      let operacion = 0

      call opcionesNumero
      input operacion

      while (((operacion < 1) OR (operacion > 6)) OR (operacion <> 6))

            if (operacion = 1) then
                  call factorial

            if (operacion = 2) then
                  call suma

            if (operacion = 3) then
                  call potencia

            if (operacion = 4) then
                  call adivinar

            if (operacion = 5) then
                  call comparaciones

            call opcionesNumero
            input operacion

      wend

end sub

sub opcionesCadena
      print ""
      print "Que operacion desea realizar?:"
      print "   1) Concatenar"
      print "   2) Concatenar repetitivo"
```

```
      print "   3) Salir"
      print ""
end sub

sub concatenar
      static cadena1$
      print  "Ingrese una cadena: "
      input  cadena1$

      static cadena2$
      print  "Ingrese otra cadena: "
      input  cadena2$

      static resultado$
      let resultado$ = cadena1$ + cadena2$

      print "Cadenas concatenadas: " ; resultado$

end sub

sub concatenarRep
      static cadena$
      print  "Ingrese una cadena:"
      input  cadena$

      static veces
      print  "Ingrese un numero:"
      input  veces

      static resultado$

      while (veces > 0)
            let resultado$ = resultado$ + cadena$
            let veces = veces - 1
      wend

      print "La cadena resultante es: " ; resultado$

end sub

sub operacionesCadena
      static operacion
      let operacion = 0

      call opcionesCadena
      input operacion

      while (((operacion < 1) OR (operacion > 3)) OR (operacion <> 3))

            if (operacion = 1) then
                  call concatenar

            if (operacion = 2) then
                  call concatenarRep
```

```
            call opcionesCadena
            input operacion

      wend
end sub

sub tiposPruebas
      print ""
      print "Que prueba desea realizar?: "
      print "   1) Con numeros"
      print "   2) Con cadenas"
      print "   3) Salir"
      print ""
end sub

print "Cual es su nombre?: "
input nombre$

print ""
print "Hola, " ; nombre$ ; ". Bienvenido al programa de prueba!"

let tipo = 0
call tiposPruebas
input tipo

while (((tipo < 1) OR (tipo > 3)) OR (tipo <> 3))

      if (tipo = 1) then
            call operacionesNumero

      if (tipo = 2) then
            call operacionesCadena

      call tiposPruebas
      input tipo

wend

print "Hasta luego, " ; nombre$ ; "!!!"

end
```