Manual de Usuario de BCL

Basic Command Language

Versión 1.5.1

Octubre 2025

Rafa

Un lenguaje de scripting inspirado en Tcl
con sintaxis estilo BASIC

Diseñado para sistemas embebidos y entornos PC

Este manual describe BCL (Basic Command Language) versión 1.5.1.

BCL está inspirado en Tcl 8.x pero presenta una sintaxis simplificada estilo BASIC, diseñada para ser amigable para principiantes y adecuada tanto para sistemas embebidos como para entornos PC.

Índice general

Índice de cuadros

Listings

Capítulo 1

Introducción

1.1. ¿Qué es BCL?

BCL (Basic Command Language) es un lenguaje de scripting interpretado y ligero diseñado para combinar el poder de Tcl con la legibilidad de BASIC. Fue creado con los siguientes objetivos en mente:

- **Simplicidad**: Fácil de aprender para principiantes
- **Portabilidad**: Se ejecuta tanto en PC como en sistemas embebidos
- **Flexibilidad**: Todo es una cadena, lo que hace que la manipulación de datos sea directa
- **Expresividad**: Estructura de comandos inspirada en Tcl con palabras clave al estilo BASIC
- **Insensibilidad a mayúsculas/minúsculas**: Los comandos y palabras clave no distinguen entre mayúsculas y minúsculas

Nota

BCL trata **todo como una cadena**. Números, listas e incluso código están representados como texto. Esto simplifica el modelo del lenguaje y facilita el trabajo con datos de diversas fuentes.

1.2. ¿Por qué BCL?

BCL llena un nicho único en el ecosistema de lenguajes de scripting:

- **Para principiantes**: La sintaxis al estilo BASIC (`IF...THEN...END`, `WHILE...DO...END`) es más intuitiva que las llaves
- **Para usuarios de Tcl**: Estructura de comandos y modelo de evaluación familiar
- **Para sistemas embebidos**: Intérprete compacto adecuado para entornos con recursos limitados
- **Para automatización**: E/S de archivos simple, procesamiento de cadenas e interacción con el sistema

1.3. Instalación y Primeros Pasos

1.3.1. Instalación

BCL puede instalarse en varias plataformas. La distribución estándar incluye:

- El ejecutable del intérprete BCL (`bcl`)
- Scripts de la biblioteca estándar
- Programas de ejemplo
- Documentación

Consulte el Apéndice A para obtener instrucciones detalladas de instalación para su plataforma.

1.3.2. Ejecución de Programas BCL

Los programas BCL pueden ejecutarse de dos formas:

1. **Modo script**: Ejecutar un archivo de script BCL

```
bcl myscript.bcl
```

2. **Modo interactivo (REPL)**: Iniciar el intérprete BCL sin argumentos

```
bcl
```

1.3.3. Tu Primer Programa BCL

Comencemos con el tradicional programa "¡Hola, Mundo!":

Hola Mundo

```
# This is a comment - my first BCL program
PUTS "Hello, World!"
```

Salida:

```
Hello, World!
```

Explicación: El comando `PUTS` imprime texto en la consola seguido de una nueva línea. El texto encerrado entre comillas dobles se trata como una cadena literal.

1.3.4. Usando el REPL

El REPL (Read-Eval-Print Loop) es el modo interactivo de BCL. Es perfecto para experimentar con comandos y probar pequeños fragmentos de código.

Sesión REPL

```
$ bcl
BCL> PUTS "Hello from REPL"
Hello from REPL
```

```
BCL> SET x 42
BCL> PUTS "The answer is $x"
The answer is 42
BCL> EXIT
```

Explicación: En modo REPL, escribes comandos y ves resultados inmediatos. El comando `EXIT` cierra el intérprete.

Consejo

¡Usa el REPL para probar comandos BCL mientras lees este manual. Es una excelente manera de aprender interactivamente!

1.4. Filosofía de BCL

Comprender la filosofía central de BCL te ayudará a escribir mejor código:

1. **Todo es una cadena**: Todos los valores son texto. Las operaciones determinan cómo se interpretan las cadenas (como números, listas, etc.)
2. **Los comandos devuelven valores**: Casi todos los comandos BCL devuelven un valor que puede ser usado por otros comandos
3. **Sustitución antes de evaluación**: Los valores de variables (usando $) y los resultados de comandos (usando []) se sustituyen antes de que el comando se ejecute
4. **Los bloques terminan con END**: Las estructuras de control usan palabras clave `END` explícitas en lugar de llaves o indentación
5. **Insensible a mayúsculas/minúsculas**: `PUTS`, `puts`, y `Puts` son todos el mismo comando

Capítulo 2

Fundamentos de Programación

Este capítulo introduce conceptos básicos de programación para principiantes absolutos usando BCL.

2.1. ¿Qué es un Programa?

Un programa es una secuencia de instrucciones que le dice a la computadora qué hacer. En BCL, los programas consisten en comandos, cada uno en su propia línea o separados por punto y coma.

Estructura de Programa Simple

```
# A program with three commands
PUTS "Starting program..."
SET name "Alice"
PUTS "Hello, $name!"
```

Salida:

```
Starting program...
Hello, Alice!
```

2.2. Variables: Almacenar Información

Las variables son contenedores con nombre que almacenan valores. Piensa en ellas como cajas etiquetadas donde puedes poner información y recuperarla más tarde.

2.2.1. Crear Variables

Usa el comando **SET** para crear y asignar valores a variables:

```
SET age 25
SET name "Bob"
SET price 19.99
```

2.2.2. Usar Variables

Para usar el valor de una variable, antepone su nombre con un signo de dólar ($):

Usar Variables

```
SET username "Alice"
SET score 100

PUTS "User: $username"
PUTS "Score: $score"
```

Salida:

```
User: Alice
Score: 100
```

2.3. Tipos de Datos: Todo es una Cadena

A diferencia de muchos lenguajes de programación, BCL no tiene tipos separados para números, texto, etc. Todo se almacena como una cadena (texto). BCL automáticamente interpreta las cadenas como números cuando es necesario para cálculos.

Cadenas como Números

```
SET a "10"
SET b "20"
SET sum [EXPR $a + $b]
PUTS "Sum: $sum"
```

Salida:

```
Sum: 30
```

Explicación: Aunque `$a` y `$b` son texto, `EXPR` los interpreta como números para aritmética.

2.4. Comentarios: Documentar tu Código

Los comentarios son notas para humanos que la computadora ignora. Ayudan a explicar qué hace tu código.

```
# This is a single-line comment

SET x 10   # Comments can also appear at line end
SET y 20   ;# Using semicolon before hash is also valid
```

Consejo

Escribe comentarios para explicar *por qué* tu código hace algo, no solo *qué* hace. ¡Tu yo futuro te lo agradecerá!

2.5. Entrada y Salida Básica

2.5.1. Salida: PUTS y PUTSN

El comando **PUTS** muestra texto en la pantalla:

```
PUTS "This appears on the screen"
PUTS "Each PUTS starts a new line"
```

Usa **PUTSN** para imprimir sin agregar una nueva línea:

PUTS vs PUTSN

```
PUTS "Line 1"
PUTS "Line 2"
PUTSN "Same "
PUTSN "line"
PUTS ""   # Empty line
```

Salida:

```
Line 1
Line 2
Same line
```

2.5.2. Entrada: GETS

El comando **GETS** lee la entrada del usuario desde el teclado:

Leer Entrada del Usuario

```
PUTS "What is your name?"
SET name [GETS stdin]
PUTS "Hello, $name!"
```

Interacción:

```
What is your name?
> Charlie
Hello, Charlie!
```

2.6. Tu Primer Programa Interactivo

Combinemos lo que hemos aprendido:

Programa de Saludo Interactivo

```
# Interactive greeting program
PUTS "=== Welcome to BCL ==="
PUTS ""

# Get user's name
PUTS "Please enter your name:"
SET username [GETS stdin]
```

```
8
9  # Get user's age
10 PUTS "Please enter your age:"
11 SET age [GETS stdin]
12
13 # Display personalized greeting
14 PUTS ""
15 PUTS "Hello, $username!"
16 PUTS "You are $age years old."
17 PUTS "Welcome to the world of BCL programming!"
```

Capítulo 3

Variables y Datos

Este capítulo explora en profundidad el sistema de variables de BCL, incluyendo creación, modificación, alcance y técnicas avanzadas.

3.1. Crear y Modificar Variables

3.1.1. SET: La Base

El comando **SET** se usa tanto para crear como para modificar variables.
Sintaxis:

```
SET variablename value          # Assign value
SET variablename                # Return current value
```

Ejemplos de SET

```
# Creating variables
SET counter 0
SET message "Hello"
SET pi 3.14159

# Modifying variables
SET counter 10
SET message "Goodbye"

# Reading values
PUTS [SET counter]   # Prints: 10
```

3.1.2. UNSET: Eliminar Variables

El comando **UNSET** elimina una variable de la memoria.

Ejemplo de UNSET

```
SET temp "temporary data"
PUTS $temp           # Prints: temporary data

UNSET temp
```

```
# PUTS $temp          # ERROR: variable doesn't exist
```

Advertencia

Acceder a una variable eliminada causa un error. Usa **INFO EXISTS** para verificar si una variable existe antes de usarla.

3.2. Expansión de Variables

La expansión de variables significa reemplazar `$varname` con el valor de la variable.

3.2.1. Expansión Básica

Expansión de Variables

```
SET fruit "apple"
SET count 5

PUTS "I have $count ${fruit}s"  # Note: BCL uses only $var
PUTS "I have $count apples"
```

Salida:

```
I have 5 apples
I have 5 apples
```

Nota

BCL usa solo la forma `$var` para la expansión de variables. A diferencia de Tcl, `${var}` no está soportado.

3.2.2. Concatenación de Cadenas

Puedes concatenar cadenas colocando expansiones de variables una al lado de la otra:

Concatenación

```
SET first "John"
SET last "Doe"
SET fullname $first" "$last
PUTS $fullname  # Prints: John Doe

# Alternative with variables only
SET a "Hello"
SET b "World"
SET c $a$b
PUTS $c  # Prints: HelloWorld
```

3.3. INCR: Incrementar Números

INCR es un comando especializado para incrementar (o decrementar) variables numéricas.

Sintaxis:

```
INCR varname           # Increment by 1
INCR varname amount    # Increment by amount
```

Ejemplos de INCR

```
SET counter 10

INCR counter        # counter is now 11
INCR counter 5      # counter is now 16
INCR counter -3     # counter is now 13 (decrement)

PUTS "Counter: $counter"
```

Salida:

```
Counter: 13
```

3.4. APPEND: Construir Cadenas

APPEND agrega texto al final de una variable, modificándola en su lugar.

Sintaxis:

```
APPEND varname value1 value2 ...
```

Ejemplos de APPEND

```
SET message "Hello"
APPEND message " " "World" "!"
PUTS $message  # Prints: Hello World!

# Building strings in a loop
SET result ""
SET i 1
WHILE $i <= 5 DO
  APPEND result $i " "
  INCR i
END
PUTS $result  # Prints: 1 2 3 4 5
```

Consejo

APPEND es más eficiente que operaciones **SET** repetidas cuando se construyen cadenas grandes en bucles.

3.5. Alcance de Variables

3.5.1. Variables Globales

Las variables creadas fuera de procedimientos son globales—pueden accederse desde cualquier parte de tu programa.

```
SET global_var "I am global"

PROC test DO
  # Cannot access global_var here without GLOBAL declaration
END
```

3.5.2. Variables Locales

Las variables creadas con **SET** dentro de un procedimiento son locales—solo existen dentro de ese procedimiento.

Local vs Global

```
SET outside "global"

PROC demo DO
  SET inside "local"
  PUTS "Inside proc: $inside"
END

demo
PUTS "Outside proc: $outside"
# PUTS $inside  # ERROR: inside doesn't exist here
```

Salida:

```
Inside proc: local
Outside proc: global
```

3.5.3. El Comando GLOBAL

Para acceder o modificar variables globales desde dentro de un procedimiento, usa **GLOBAL**.

Usando GLOBAL

```
SET score 0

PROC add_points WITH points DO
  GLOBAL score
  INCR score $points
END

add_points 10
add_points 5
PUTS "Total score: $score"  # Prints: Total score: 15
```

3.6. Ejemplos Prácticos

3.6.1. Ejemplo de Contador

Contador de Visitas

```
SET hits 0

PROC record_hit DO
  GLOBAL hits
  INCR hits
  PUTS "Hit number $hits recorded"
END

record_hit
record_hit
record_hit
PUTS "Total hits: $hits"
```

Salida:

```
Hit number 1 recorded
Hit number 2 recorded
Hit number 3 recorded
Total hits: 3
```

3.6.2. Ejemplo de Constructor de Texto

Construir un Reporte

```
SET report ""
APPEND report "=== SYSTEM REPORT ===\n"
APPEND report "Date: " [CLOCK FORMAT [CLOCK SECONDS]] "\n"
APPEND report "Status: OK\n"
APPEND report "====================="

PUTS $report
```

Capítulo 4

Expresiones y Matemáticas

BCL proporciona evaluación de expresiones matemáticas y lógicas potente a través del comando **EXPR**.

4.1. El Comando EXPR

EXPR evalúa expresiones matemáticas y lógicas.
Sintaxis:

```
EXPR expression
```

Aritmética Básica

```
SET result [EXPR 2 + 3]
PUTS $result  # Prints: 5

SET x 10
SET y 3
SET sum [EXPR $x + $y]
SET product [EXPR $x * $y]
SET quotient [EXPR $x / $y]

PUTS "Sum: $sum"          # Prints: Sum: 13
PUTS "Product: $product"  # Prints: Product: 30
PUTS "Quotient: $quotient"  # Prints: Quotient: 3.333...
```

4.2. Operadores Aritméticos

Usando Operadores

```
# Complex expression with parentheses
SET result [EXPR (10 + 5) * 2 - 3]
PUTS $result  # Prints: 27

# Power operator
SET squared [EXPR 5 ^ 2]
PUTS "5 squared is $squared"  # Prints: 5 squared is 25
```

Cuadro 4.1: Operadores Aritméticos en BCL

Operador	Operación	Ejemplo
`+`	Suma	`EXPR 5 + 3` → 8
`-`	Resta	`EXPR 5 - 3` → 2
`*`	Multiplicación	`EXPR 5 * 3` → 15
`/`	División	`EXPR 5 / 2` → 2.5
`%`	Módulo (resto)	`EXPR 5 % 3` → 2
`^`	Potencia	`EXPR 2 ^ 8` → 256

```
8
9  # Modulo for even/odd check
10 SET num 17
11 SET remainder [EXPR $num % 2]
12 IF $remainder == 0 THEN
13   PUTS "$num is even"
14 ELSE
15   PUTS "$num is odd"
16 END
```

4.3. Operadores de Comparación

Cuadro 4.2: Operadores de Comparación

Operador	Significado	Ejemplo
`=`	Igual a	`$a == 5`
`!=`	No igual a	`$a != 5`
`<`	Menor que	`$a <10`
`<=`	Menor o igual que	`$a <= 10`
`>`	Mayor que	`$a >5`
`>=`	Mayor o igual que	`$a >= 5`

Comparaciones

```
1  SET age 25
2
3  IF $age >= 18 THEN
4    PUTS "Adult"
5  ELSE
6    PUTS "Minor"
7  END
8
9  SET score 85
10 IF $score >= 90 THEN
11   PUTS "Grade: A"
12 ELSEIF $score >= 80 THEN
```

```
  PUTS "Grade: B"
ELSEIF $score >= 70 THEN
  PUTS "Grade: C"
ELSE
  PUTS "Grade: F"
END
```

4.4. Operadores Lógicos

Cuadro 4.3: Operadores Lógicos

Operador	Símbolo	Ejemplo
AND	&&	$a >0 AND $a <10
OR	||	$a == 5 OR $a == 10
NOT	!	NOT $flag

Operaciones Lógicas

```
SET age 25
SET has_license 1

# AND operation
IF $age >= 18 AND $has_license THEN
  PUTS "Can drive"
END

# OR operation
SET day "Saturday"
IF $day == "Saturday" OR $day == "Sunday" THEN
  PUTS "It's the weekend!"
END

# NOT operation
SET raining 0
IF NOT $raining THEN
  PUTS "No umbrella needed"
END
```

4.5. Funciones Matemáticas

BCL incluye muchas funciones matemáticas:

Trigonometría

```
SET pi 3.14159265359

# Sine of 90 degrees (pi/2 radians)
```

Cuadro 4.4: Funciones Matemáticas

Función	Descripción
`abs(x)`	Valor absoluto
`sqrt(x)`	Raíz cuadrada
`pow(x,y)`	x elevado a la potencia y
`exp(x)`	e elevado a la potencia x
`log(x)`	Logaritmo base 10
`ln(x)`	Logaritmo natural (base e)
`sin(x)`	Seno (radianes)
`cos(x)`	Coseno (radianes)
`tan(x)`	Tangente (radianes)
`asin(x)`	Arcoseno
`acos(x)`	Arcocoseno
`atan(x)`	Arcotangente
`hypo(x,y)`	Hipotenusa: sqrt($x^2 + y^2$)
`ceil(x)`	Redondear hacia arriba
`floor(x)`	Redondear hacia abajo
`round(x)`	Redondear al entero más cercano
`int(x)`	Convertir a entero
`double(x)`	Convertir a punto flotante
`rand()`	Número aleatorio 0.0 a 1.0
`srand(seed)`	Establecer semilla aleatoria

```
SET angle [EXPR $pi / 2]
SET sine [EXPR sin($angle)]
PUTS [FORMAT "sin(90\textdegree) = %.4f" $sine]

# Calculate hypotenuse
SET a 3
SET b 4
SET c [EXPR hypo($a, $b)]
PUTS [FORMAT "Hypotenuse of %d and %d = %.2f" $a $b $c]
```

Salida:

```
sin(90\textdegree) = 1.0000
Hypotenuse of 3 and 4 = 5.00
```

Números Aleatorios

```
# Generate random integer from 1 to 10
SET random [EXPR int(rand() * 10) + 1]
PUTS "Random number: $random"

# Dice roll simulator
PROC roll_dice DO
  SET roll [EXPR int(rand() * 6) + 1]
  RETURN $roll
END
```

```
10
11 SET roll1 [roll_dice]
12 SET roll2 [roll_dice]
13 PUTS "You rolled: $roll1 and $roll2"
```

4.6. Ejemplos Prácticos

4.6.1. Calculadora de Círculos

Área y Circunferencia de un Círculo

```
1  PROC circle_stats WITH radius DO
2    SET pi 3.14159265359
3    SET area [EXPR $pi * $radius * $radius]
4    SET circumference [EXPR 2 * $pi * $radius]
5
6    PUTS [FORMAT "Radius: %.2f" $radius]
7    PUTS [FORMAT "Area: %.2f" $area]
8    PUTS [FORMAT "Circumference: %.2f" $circumference]
9  END
10
11 circle_stats 5.0
```

Salida:

```
Radius: 5.00
Area: 78.54
Circumference: 31.42
```

4.6.2. Resolución de Ecuaciones Cuadráticas

Resolviendo $ax^2 + bx + c$

```
1  PROC solve_quadratic WITH a b c DO
2    # Calculate discriminant
3    SET disc [EXPR $b*$b - 4*$a*$c]
4
5    IF $disc < 0 THEN
6      PUTS "No real solutions"
7      RETURN
8    END
9
10   # Calculate solutions
11   SET x1 [EXPR (-$b + sqrt($disc)) / (2*$a)]
12   SET x2 [EXPR (-$b - sqrt($disc)) / (2*$a)]
13
14   PUTS [FORMAT "x1 = %.4f" $x1]
15   PUTS [FORMAT "x2 = %.4f" $x2]
16 END
17
18 # Solve x$^2$ - 5x + 6 = 0
```

```
19 solve_quadratic 1 -5 6
```

Salida:

```
x1 = 3.0000
x2 = 2.0000
```

Capítulo 5

Estructuras de Control

Las estructuras de control te permiten cambiar el orden en que tu programa ejecuta comandos basándose en condiciones o repetición.

5.1. IF...THEN...ELSE...END

La instrucción `IF` ejecuta código condicionalmente.

Sintaxis:

```
IF condition THEN
  commands
[ELSEIF condition THEN
  commands]
[ELSE
  commands]
END
```

IF Simple

```
SET temperature 25

IF $temperature > 30 THEN
  PUTS "It's hot!"
ELSEIF $temperature > 20 THEN
  PUTS "It's warm"
ELSEIF $temperature > 10 THEN
  PUTS "It's cool"
ELSE
  PUTS "It's cold!"
END
```

Salida:

```
It's warm
```

IF Anidado

```
SET age 25
SET student 1

```

```
IF $age < 18 THEN
  PUTS "Minor - discounted ticket"
ELSE
  IF $student THEN
    PUTS "Adult student - discounted ticket"
  ELSE
    PUTS "Regular adult ticket"
  END
END
```

5.2. Bucles WHILE

WHILE repite código mientras una condición sea verdadera.

Sintaxis:

```
WHILE condition DO
  commands
END
```

Cuenta Regresiva

```
SET count 5

WHILE $count > 0 DO
  PUTS "Countdown: $count"
  INCR count -1
END

PUTS "Liftoff!"
```

Salida:

```
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Liftoff!
```

Validación de Entrada

```
SET valid 0

WHILE NOT $valid DO
  PUTS "Enter a number between 1 and 10:"
  SET input [GETS stdin]

  IF $input >= 1 AND $input <= 10 THEN
    SET valid 1
    PUTS "Thank you! You entered $input"
  ELSE
```

```
    PUTS "Invalid input. Try again."
  END
END
```

5.3. Bucles FOR

BCL soporta dos estilos de bucles `FOR`.

5.3.1. FOR con Variable Explícita

Bucle FOR - Variable Explícita

```
FOR [SET i 1] TO $i <= 10 DO
  PUTS "Iteration $i"
  INCR i
END
```

5.3.2. FOR con Contador Interno

Bucle FOR - Contador Interno

```
FOR 1 TO 10 DO
  PUTS "Number: $__FOR"
END

# With STEP
FOR 0 TO 100 STEP 10 DO
  PUTS "Value: $__FOR"
END
```

Salida:

```
Number: 1
Number: 2
...
Number: 10
Value: 0
Value: 10
...
Value: 100
```

Nota

La variable de contador interno se llama `$__FOR` y es creada automáticamente por BCL.

5.4. Bucles FOREACH

`FOREACH` itera sobre elementos de lista.

Sintaxis:

```
FOREACH variable IN list DO
  commands
END
```

Iterar Listas

```
SET colors [LIST red green blue yellow]

FOREACH color IN $colors DO
  PUTS "Color: $color"
END
```

Salida:

```
Color: red
Color: green
Color: blue
Color: yellow
```

Procesar Datos

```
SET scores [LIST 85 92 78 95 88]
SET total 0
SET count 0

FOREACH score IN $scores DO
  SET total [EXPR $total + $score]
  INCR count
END

SET average [EXPR $total / $count]
PUTS [FORMAT "Average score: %.2f" $average]
```

Salida:

```
Average score: 87.60
```

5.5. Instrucción SWITCH

`SWITCH` selecciona uno de muchos bloques de código para ejecutar.

Sintaxis:

```
SWITCH expression DO
  CASE value1
    commands
  CASE value2
    commands
  DEFAULT
    commands
END
```

Ejemplo de SWITCH

```
SET day "Tuesday"

SWITCH $day DO
  CASE "Monday"
    PUTS "Start of work week"
  CASE "Tuesday"
    PUTS "Second day"
  CASE "Wednesday"
    PUTS "Midweek"
  CASE "Thursday"
    PUTS "Almost Friday"
  CASE "Friday"
    PUTS "Last work day!"
  DEFAULT
    PUTS "Weekend!"
END
```

Salida:

```
Second day
```

Sistema de Menú

```
PUTS "Select an option:"
PUTS "1. New file"
PUTS "2. Open file"
PUTS "3. Save file"
PUTS "4. Exit"

SET choice [GETS stdin]

SWITCH $choice DO
  CASE "1"
    PUTS "Creating new file..."
  CASE "2"
    PUTS "Opening file..."
  CASE "3"
    PUTS "Saving file..."
  CASE "4"
    PUTS "Goodbye!"
    EXIT
  DEFAULT
    PUTS "Invalid choice"
END
```

5.6. BREAK y CONTINUE

5.6.1. BREAK: Salir del Bucle Anticipadamente

`BREAK` termina el bucle más interno inmediatamente.

Usando BREAK

```
# Search for a number
SET target 7
SET found 0

FOR 1 TO 10 DO
  IF $__FOR == $target THEN
    SET found $__FOR
    BREAK
  END
END

IF $found THEN
  PUTS "Found $target"
ELSE
  PUTS "Not found"
END
```

5.6.2. CONTINUE: Saltar a la Siguiente Iteración

`CONTINUE` salta el resto de la iteración actual y va a la siguiente.

Usando CONTINUE

```
# Print odd numbers only
FOR 1 TO 10 DO
  SET num $__FOR
  SET remainder [EXPR $num % 2]

  IF $remainder == 0 THEN
    CONTINUE  # Skip even numbers
  END

  PUTS $num
END
```

Salida:

```
1
3
5
7
9
```

5.7. EXIT: Terminar el Programa

`EXIT` termina el programa completo (o sesión REPL).

```
EXIT          # Exit with code 0
EXIT 1        # Exit with code 1 (error)
```

5.8. Ejemplos Completos

5.8.1. Juego de Adivinar Números

Juego de Adivinanzas

```
# Generate random number 1-100
SET secret [EXPR int(rand() * 100) + 1]
SET guesses 0
SET found 0

PUTS "I'm thinking of a number between 1 and 100"

WHILE NOT $found DO
  PUTS "Enter your guess:"
  SET guess [GETS stdin]
  INCR guesses

  IF $guess == $secret THEN
    SET found 1
    PUTS "Correct! You won in $guesses guesses!"
  ELSEIF $guess < $secret THEN
    PUTS "Too low!"
  ELSE
    PUTS "Too high!"
  END
END
```

5.8.2. FizzBuzz

FizzBuzz Clásico

```
FOR 1 TO 100 DO
  SET num $__FOR
  SET by3 [EXPR $num % 3]
  SET by5 [EXPR $num % 5]

  IF $by3 == 0 AND $by5 == 0 THEN
    PUTS "FizzBuzz"
  ELSEIF $by3 == 0 THEN
    PUTS "Fizz"
  ELSEIF $by5 == 0 THEN
    PUTS "Buzz"
  ELSE
    PUTS $num
  END
END
```

Capítulo 6

Procedimientos (Funciones)

Los procedimientos (también llamados funciones en otros lenguajes) te permiten empaquetar código en bloques reutilizables.

6.1. Definir Procedimientos

Sintaxis con parámetros:

```
PROC name WITH param1 param2 ... DO
  commands
  RETURN value
END
```

Sintaxis sin parámetros:

```
PROC name DO
  commands
  RETURN value
END
```

Nota

Cuando un procedimiento no tiene parámetros, la palabra clave `WITH` puede omitirse.

Procedimiento con Parámetro

```
PROC greet WITH name DO
  PUTS "Hello, $name!"
END

greet "Alice"
greet "Bob"
```

Salida:

```
Hello, Alice!
Hello, Bob!
```

Procedimiento sin Parámetros

```
PROC show_version DO
  PUTS "BCL Interpreter v1.5.0"
END

show_version
```

Salida:

```
BCL Interpreter v1.5.0
```

Nota

Los procedimientos se invocan por su nombre solo—no se necesita palabra clave `CALL`.

6.2. Parámetros

6.2.1. Parámetros Fijos

Los parámetros se declaran en la cláusula `WITH` y son accesibles como variables dentro del procedimiento.

Múltiples Parámetros

```
PROC add WITH a b DO
  SET result [EXPR $a + $b]
  RETURN $result
END

SET sum [add 5 3]
PUTS "5 + 3 = $sum"
```

Salida:

```
5 + 3 = 8
```

6.2.2. Parámetros Opcionales

Los parámetros opcionales se prefijan con `@`. Pueden o no proporcionarse al llamar al procedimiento.

Parámetros Opcionales

```
PROC greet WITH name @title DO
  IF [INFO EXISTS title] THEN
    PUTS "Hello, $title $name"
  ELSE
    PUTS "Hello, $name"
  END
END

greet "Smith"
```

```
greet "Smith" "Dr."
```

Salida:

```
Hello, Smith
Hello, Dr. Smith
```

Consejo

Usa **INFO EXISTS** para verificar si se proporcionó un parámetro opcional.

6.3. Valores de Retorno

El comando **RETURN** sale de un procedimiento y opcionalmente devuelve un valor.

Devolver Valores

```
PROC square WITH n DO
  SET result [EXPR $n * $n]
  RETURN $result
END

PROC is_even WITH n DO
  SET remainder [EXPR $n % 2]
  IF $remainder == 0 THEN
    RETURN 1  # true
  ELSE
    RETURN 0  # false
  END
END

SET s [square 7]
PUTS "7 squared = $s"

IF [is_even 10] THEN
  PUTS "10 is even"
END
```

Salida:

```
7 squared = 49
10 is even
```

Nota

Si **RETURN** se llama sin un valor, o si un procedimiento alcanza su **END** sin devolver nada, devuelve una cadena vacía.

6.4. Alcance de Variables en Procedimientos

6.4.1. Variables Locales

Las variables creadas con **SET** dentro de un procedimiento son locales—solo existen dentro de ese procedimiento.

Variables Locales

```
PROC calculate WITH x DO
  SET double [EXPR $x * 2]   # local variable
  SET triple [EXPR $x * 3]   # local variable
  PUTS "Inside: double=$double, triple=$triple"
END

calculate 5
# PUTS $double  # ERROR: double doesn't exist here
```

6.4.2. Acceder a Variables Globales

Usa **GLOBAL** para acceder o modificar variables globales desde dentro de un procedimiento.

Variables Globales en Procedimientos

```
SET counter 0

PROC increment WITH amount DO
  GLOBAL counter
  INCR counter $amount
  PUTS "Counter is now: $counter"
END

increment 5
increment 3
PUTS "Final counter: $counter"
```

Salida:

```
Counter is now: 5
Counter is now: 8
Final counter: 8
```

6.5. Procedimientos Recursivos

Los procedimientos pueden llamarse a sí mismos—esto se llama recursión.

Factorial (Recursivo)

```
PROC factorial WITH n DO
  IF $n <= 1 THEN
    RETURN 1
  ELSE
```

```
    SET prev [factorial [EXPR $n - 1]]
    RETURN [EXPR $n * $prev]
  END
END

PUTS "5! = [factorial 5]"
PUTS "10! = [factorial 10]"
```

Salida:

```
5! = 120
10! = 3628800
```

Advertencia

¡Los procedimientos recursivos deben tener un caso base (condición de terminación) para evitar recursión infinita!

Fibonacci (Recursivo)

```
PROC fib WITH n DO
  IF $n <= 1 THEN
    RETURN $n
  ELSE
    SET n1 [EXPR $n - 1]
    SET n2 [EXPR $n - 2]
    SET f1 [fib $n1]
    SET f2 [fib $n2]
    RETURN [EXPR $f1 + $f2]
  END
END

# Print first 10 Fibonacci numbers
FOR 0 TO 9 DO
  SET f [fib $__FOR]
  PUTS "fib($__FOR) = $f"
END
```

6.6. Ejemplos Prácticos

6.6.1. Convertidor de Temperatura

Celsius a Fahrenheit

```
PROC celsius_to_fahrenheit WITH celsius DO
  SET fahrenheit [EXPR ($celsius * 9.0 / 5.0) + 32]
  RETURN $fahrenheit
END

PROC fahrenheit_to_celsius WITH fahrenheit DO
  SET celsius [EXPR ($fahrenheit - 32) * 5.0 / 9.0]
```

```
  RETURN $celsius
END

SET c 25
SET f [celsius_to_fahrenheit $c]
PUTS [FORMAT "%d\textdegreeC = %.1f\textdegreeF" $c $f]

SET f2 100
SET c2 [fahrenheit_to_celsius $f2]
PUTS [FORMAT "%d\textdegreeF = %.1f\textdegreeC" $f2 $c2]
```

Salida:

```
25\textdegreeC = 77.0\textdegreeF
100\textdegreeF = 37.8\textdegreeC
```

6.6.2. Utilidades de Cadenas

Procedimientos Auxiliares de Cadenas

```
PROC reverse_string WITH str DO
  RETURN [STRING REVERSE $str]
END

PROC capitalize WITH str DO
  SET first [STRING INDEX $str 0]
  SET rest [STRING RANGE $str 1 end]
  SET first_upper [STRING TOUPPER $first]
  SET rest_lower [STRING TOLOWER $rest]
  RETURN [STRING CAT $first_upper $rest_lower]
END

SET text "hello world"
PUTS "Original: $text"
PUTS "Reversed: [reverse_string $text]"
PUTS "Capitalized: [capitalize $text]"
```

Salida:

```
Original: hello world
Reversed: dlrow olleh
Capitalized: Hello world
```

Capítulo 7

Listas

Las listas son colecciones de valores almacenados en una sola variable. BCL proporciona comandos completos para manipulación de listas.

7.1. Crear Listas

7.1.1. Comando LIST

Crear Listas

```
SET numbers [LIST 1 2 3 4 5]
SET colors [LIST red green blue]
SET mixed [LIST "hello world" 42 3.14]

PUTS $numbers
PUTS $colors
```

Salida:

```
1 2 3 4 5
red green blue
```

7.1.2. Comando SPLIT

`SPLIT` crea una lista dividiendo una cadena en un separador.

Dividir Cadenas

```
SET csv "apple,banana,cherry,date"
SET fruits [SPLIT $csv ","]

FOREACH fruit IN $fruits DO
  PUTS "Fruit: $fruit"
END
```

Salida:

```
Fruit: apple
Fruit: banana
Fruit: cherry
```

```
Fruit: date
```

7.2. Acceder a Elementos de Listas

7.2.1. LINDEX: Obtener Elemento por Índice

Acceder a Elementos

```
SET names [LIST Alice Bob Charlie Diana]

SET first [LINDEX $names 0]
SET second [LINDEX $names 1]
SET last [LINDEX $names 3]

PUTS "First: $first"
PUTS "Second: $second"
PUTS "Last: $last"
```

Salida:

```
First: Alice
Second: Bob
Last: Diana
```

7.2.2. LRANGE: Obtener Sublista

Extraer Sublistas

```
SET numbers [LIST 10 20 30 40 50 60]

SET first_three [LRANGE $numbers 0 2]
SET last_two [LRANGE $numbers 4 5]
SET middle [LRANGE $numbers 2 4]

PUTS "First three: $first_three"
PUTS "Last two: $last_two"
PUTS "Middle: $middle"
```

Salida:

```
First three: 10 20 30
Last two: 50 60
Middle: 30 40 50
```

7.2.3. LLENGTH: Obtener Longitud de Lista

Longitud de Lista

```
SET items [LIST pen paper pencil eraser]
SET count [LLENGTH $items]

PUTS "The list has $count items"
```

```
# Iterate using length
SET i 0
WHILE $i < $count DO
  SET item [LINDEX $items $i]
  PUTS "Item $i: $item"
  INCR i
END
```

7.3. Modificar Listas

7.3.1. LAPPEND: Agregar al Final

Agregar a Listas

```
SET fruits [LIST apple banana]
SET fruits [LAPPEND $fruits cherry]
SET fruits [LAPPEND $fruits date elderberry]

PUTS $fruits
```

Salida:

```
apple banana cherry date elderberry
```

7.3.2. LINSERT: Insertar en Posición

Insertar Elementos

```
SET numbers [LIST 1 2 4 5]
# Insert 3 at index 2
SET numbers [LINSERT $numbers 2 3]

PUTS $numbers  # Prints: 1 2 3 4 5
```

7.3.3. LREPLACE: Reemplazar Rango

Reemplazar Elementos

```
SET letters [LIST a b c d e]

# Replace index 1-2 with X Y
SET letters [LREPLACE $letters 1 2 X Y]
PUTS $letters  # Prints: a X Y d e

# Delete elements (no replacement)
SET letters [LREPLACE $letters 1 2]
PUTS $letters  # Prints: a d e
```

7.4. Ordenar y Buscar

7.4.1. LSORT: Ordenar Listas

Ordenar

```
SET unsorted [LIST zebra apple monkey dog cat]
SET sorted [LSORT $unsorted]

PUTS "Unsorted: $unsorted"
PUTS "Sorted: $sorted"
```

Salida:

```
Unsorted: zebra apple monkey dog cat
Sorted: apple cat dog monkey zebra
```

Nota

`LSORT` realiza ordenamiento ASCII, que es sensible a mayúsculas/minúsculas. Las letras mayúsculas vienen antes que las minúsculas.

7.4.2. LSEARCH: Encontrar Elemento

Buscar en Listas

```
SET fruits [LIST apple banana cherry date]

SET idx1 [LSEARCH $fruits "banana"]
SET idx2 [LSEARCH $fruits "grape"]

PUTS "Index of banana: $idx1"   # 1
PUTS "Index of grape: $idx2"    # -1 (not found)

IF $idx1 != -1 THEN
  PUTS "Found banana at position $idx1"
END
```

7.5. Operaciones con Listas

7.5.1. JOIN: Lista a Cadena

Unir Listas

```
SET words [LIST Hello World from BCL]

SET sentence [JOIN $words " "]
SET csv [JOIN $words ","]

PUTS $sentence  # Hello World from BCL
PUTS $csv       # Hello,World,from,BCL
```

7.5.2. CONCAT: Combinar Listas

Concatenar Listas

```
SET list1 [LIST 1 2 3]
SET list2 [LIST 4 5 6]
SET list3 [LIST 7 8 9]

SET combined [CONCAT $list1 $list2 $list3]
PUTS $combined  # 1 2 3 4 5 6 7 8 9
```

7.6. Ejemplos Prácticos

7.6.1. Gestor de Lista de Compras

Lista de Compras

```
SET shopping_list [LIST]

PROC add_item WITH item DO
  GLOBAL shopping_list
  SET shopping_list [LAPPEND $shopping_list $item]
  PUTS "Added: $item"
END

PROC show_list DO
  GLOBAL shopping_list
  SET count [LLENGTH $shopping_list]

  IF $count == 0 THEN
    PUTS "Shopping list is empty"
    RETURN
  END

  PUTS "Shopping List ($count items):"
  SET i 0
  FOREACH item IN $shopping_list DO
    INCR i
    PUTS "  $i. $item"
  END
END

add_item "Milk"
add_item "Bread"
add_item "Eggs"
show_list
```

Salida:

```
Added: Milk
Added: Bread
Added: Eggs
Shopping List (3 items):
```

```
1. Milk
2. Bread
3. Eggs
```

7.6.2. Calculadora de Calificaciones

Calcular Promedio de Calificaciones

```
SET grades [LIST 85 92 78 90 88 95]

# Calculate sum
SET sum 0
FOREACH grade IN $grades DO
  SET sum [EXPR $sum + $grade]
END

# Calculate average
SET count [LLENGTH $grades]
SET average [EXPR $sum / $count]

# Find min and max
SET sorted [LSORT $grades]
SET min [LINDEX $sorted 0]
SET max [LINDEX $sorted [EXPR $count - 1]]

PUTS [FORMAT "Count: %d" $count]
PUTS [FORMAT "Average: %.2f" $average]
PUTS [FORMAT "Minimum: %d" $min]
PUTS [FORMAT "Maximum: %d" $max]
```

Salida:

```
Count: 6
Average: 88.00
Minimum: 78
Maximum: 95
```

Capítulo 8

Manipulación de Cadenas

Las cadenas (texto) son la base de BCL. Dado que todo en BCL es una cadena, comprender cómo manipular texto es esencial. Este capítulo cubre todas las herramientas que BCL proporciona para trabajar con cadenas.

8.1. ¿Qué son las Cadenas?

Una cadena es simplemente una secuencia de caracteres—letras, números, símbolos, espacios, etc. En BCL, las cadenas pueden ser:

- Palabras simples: `hello`
- Oraciones: `Hello, World!`
- Números almacenados como texto: `42` o `3.14`
- Vacías: (una cadena sin caracteres)

Crear Cadenas

```
# Simple strings
SET greeting "Hello"
SET message "Welcome to BCL programming!"

# Strings with numbers
SET year "2025"
SET price "19.99"

# Empty string
SET empty ""

# Strings with special characters
SET symbols "!@#$%^&*()"
```

Nota

Recuerda: En BCL, todo es una cadena. Incluso el número `42` se almacena como la cadena de dos caracteres "4ƶ "2".

8.2. El Comando STRING

El comando **STRING** es tu navaja suiza para manipulación de texto. Tiene muchos subcomandos, cada uno realizando una operación específica.

Sintaxis General:

```
STRING subcommand arguments...
```

8.2.1. STRING LENGTH - Medir Texto

STRING LENGTH te dice cuántos caracteres hay en una cadena.

Sintaxis:

```
STRING LENGTH string
```

Longitud de Cadena

```
SET word "hello"
SET len [STRING LENGTH $word]
PUTS "The word '$word' has $len characters"

SET sentence "This is a test"
PUTS "Length: [STRING LENGTH $sentence]"

# Empty string has length 0
SET empty ""
PUTS "Empty string length: [STRING LENGTH $empty]"
```

Salida:

```
The word 'hello' has 5 characters
Length: 14
Empty string length: 0
```

Consejo

La longitud incluye espacios y puntuación. "hi!"tiene longitud 3, no 2.

8.2.2. STRING INDEX - Obtener Caracteres Individuales

STRING INDEX extrae un carácter de una posición específica.

Sintaxis:

```
STRING INDEX string position
```

Nota

¡Las posiciones empiezan en 0! El primer carácter está en la posición 0, el segundo en la posición 1, etc.

Extraer Caracteres

```
SET text "HELLO"

# Get first character (position 0)
SET first [STRING INDEX $text 0]
PUTS "First: $first"  # H

# Get third character (position 2)
SET third [STRING INDEX $text 2]
PUTS "Third: $third"  # L

# Get last character
SET len [STRING LENGTH $text]
SET last_pos [EXPR $len - 1]
SET last [STRING INDEX $text $last_pos]
PUTS "Last: $last"  # O

# You can use 'end' for the last character
SET last2 [STRING INDEX $text end]
PUTS "Also last: $last2"  # O
```

Salida:

```
First: H
Third: L
Last: O
Also last: O
```

8.2.3. STRING RANGE - Extraer Subcadenas

STRING RANGE extrae una porción de una cadena de una posición a otra.

Sintaxis:

```
STRING RANGE string start end
```

Extracción de Subcadenas

```
SET text "Hello, World!"

# Get first 5 characters (positions 0-4)
SET hello [STRING RANGE $text 0 4]
PUTS $hello  # Hello

# Get "World" (positions 7-11)
SET world [STRING RANGE $text 7 11]
PUTS $world  # World

# Get from position 7 to the end
SET rest [STRING RANGE $text 7 end]
PUTS $rest  # World!

# Get last 6 characters
SET last [STRING RANGE $text end-5 end]
```

```
PUTS $last   # World!
```

Salida:

```
Hello
World
World!
World!
```

Consejo

Usa `end` para referirse a la última posición, y `end-N` para contar hacia atrás desde el final.

8.2.4. STRING TOUPPER y TOLOWER - Cambiar Mayúsculas/Minúsculas

Estos comandos convierten cadenas a mayúsculas o minúsculas.

Sintaxis:

```
STRING TOUPPER string
STRING TOLOWER string
```

Conversión de Mayúsculas/Minúsculas

```
SET text "Hello World"

SET upper [STRING TOUPPER $text]
PUTS $upper   # HELLO WORLD

SET lower [STRING TOLOWER $text]
PUTS $lower   # hello world

# Useful for case-insensitive comparisons
SET input "YES"
SET normalized [STRING TOLOWER $input]

IF $normalized == "yes" THEN
  PUTS "User said yes!"
END
```

Salida:

```
HELLO WORLD
hello world
User said yes!
```

8.2.5. STRING TRIM - Eliminar Espacios en Blanco

`STRING TRIM` elimina espacios, tabulaciones y saltos de línea del principio y/o final de una cadena.

Sintaxis:

```
STRING TRIM string                # Remove from both ends
STRING TRIMLEFT string            # Remove from left only
STRING TRIMRIGHT string           # Remove from right only
STRING TRIM string characters     # Remove specific characters
```

Recortar Espacios en Blanco

```
# User input often has extra spaces
SET input "   hello   "

SET clean [STRING TRIM $input]
PUTS "[$clean]"  # [hello]

# Trim only from left
SET left [STRING TRIMLEFT $input]
PUTS "[$left]"  # [hello   ]

# Trim only from right
SET right [STRING TRIMRIGHT $input]
PUTS "[$right]"  # [   hello]

# Trim specific characters
SET text "***Hello***"
SET trimmed [STRING TRIM $text "*"]
PUTS $trimmed  # Hello
```

Salida:

```
[hello]
[hello   ]
[   hello]
Hello
```

8.2.6. STRING COMPARE - Comparar Cadenas

STRING COMPARE compara dos cadenas y devuelve:

- -1 si la primera cadena viene antes que la segunda (alfabéticamente)
- 0 si las cadenas son idénticas
- 1 si la primera cadena viene después de la segunda

Sintaxis:

```
STRING COMPARE string1 string2
STRING COMPARE -nocase string1 string2  # Ignore case
```

Comparación de Cadenas

```
# Exact comparison
SET result [STRING COMPARE "apple" "banana"]
PUTS $result  # -1 (apple comes before banana)

```

```
SET result [STRING COMPARE "zoo" "ant"]
PUTS $result  # 1 (zoo comes after ant)

SET result [STRING COMPARE "hello" "hello"]
PUTS $result  # 0 (identical)

# Case-insensitive comparison
SET r1 [STRING COMPARE "Hello" "hello"]
PUTS "Case-sensitive: $r1"  # 1 (different)

SET r2 [STRING COMPARE -nocase "Hello" "hello"]
PUTS "Case-insensitive: $r2"  # 0 (same)
```

Salida:

```
-1
1
0
Case-sensitive: 1
Case-insensitive: 0
```

8.2.7. STRING FIRST y LAST - Encontrar Subcadenas

Estos comandos encuentran la posición de una subcadena dentro de una cadena.

Sintaxis:

```
STRING FIRST substring string [startpos]
STRING LAST substring string [startpos]
```

Encontrar Subcadenas

```
SET text "hello world, hello BCL"

# Find first occurrence of "hello"
SET pos [STRING FIRST "hello" $text]
PUTS "First 'hello' at position: $pos"  # 0

# Find last occurrence of "hello"
SET pos [STRING LAST "hello" $text]
PUTS "Last 'hello' at position: $pos"  # 13

# Search starting from position 5
SET pos [STRING FIRST "hello" $text 5]
PUTS "Next 'hello' after pos 5: $pos"  # 13

# Not found returns -1
SET pos [STRING FIRST "goodbye" $text]
IF $pos == -1 THEN
  PUTS "'goodbye' not found"
END
```

Salida:

```
First 'hello' at position: 0
```

```
Last 'hello' at position: 13
Next 'hello' after pos 5: 13
'goodbye' not found
```

8.2.8. STRING REPLACE - Reemplazar Texto

STRING REPLACE reemplaza parte de una cadena con texto nuevo.

Sintaxis:

```
STRING REPLACE string start end newtext
```

Reemplazar Partes de Cadenas

```
SET text "Hello World"

# Replace "World" (positions 6-10) with "BCL"
SET new [STRING REPLACE $text 6 10 "BCL"]
PUTS $new  # Hello BCL

# Replace first word
SET new [STRING REPLACE $text 0 4 "Goodbye"]
PUTS $new  # Goodbye World

# Delete part of string (replace with empty)
SET text "Hello, World!"
SET new [STRING REPLACE $text 5 6 ""]
PUTS $new  # Hello World!
```

Salida:

```
Hello BCL
Goodbye World
Hello World!
```

8.2.9. STRING REVERSE - Invertir Texto

STRING REVERSE invierte el orden de los caracteres en una cadena.

Sintaxis:

```
STRING REVERSE string
```

Invertir Cadenas

```
SET text "hello"
SET reversed [STRING REVERSE $text]
PUTS $reversed  # olleh

SET text "racecar"
SET rev [STRING REVERSE $text]
IF $text == $rev THEN
  PUTS "'$text' is a palindrome!"
END
```

Salida:

```
olleh
'racecar' is a palindrome!
```

8.2.10. STRING MATCH - Coincidencia de Patrones

`STRING MATCH` verifica si una cadena coincide con un patrón con comodines.
Patrones:

- `*` - coincide con cualquier secuencia de caracteres
- `?` - coincide con cualquier carácter individual
- `[abc]` - coincide con cualquier carácter entre corchetes

Sintaxis:

```
STRING MATCH pattern string
STRING MATCH -nocase pattern string  # Ignore case
```

Coincidencia de Patrones

```
# Match with wildcards
IF [STRING MATCH "*.txt" "document.txt"] THEN
  PUTS "It's a text file"
END

# Match any 3-letter word
IF [STRING MATCH "???" "cat"] THEN
  PUTS "Three letter word"
END

# Match email pattern
SET email "user@example.com"
IF [STRING MATCH "*@*.*" $email] THEN
  PUTS "Looks like an email"
END

# Character sets
IF [STRING MATCH "\[0-9\]*" "123abc"] THEN
  PUTS "Starts with a digit"
END
```

Salida:

```
It's a text file
Three letter word
Looks like an email
Starts with a digit
```

8.3. FORMAT y SCAN - Texto Formateado

8.3.1. FORMAT: Crear Salida Formateada

`FORMAT` crea cadenas formateadas, similar a printf en C.
Especificadores de Formato Comunes:

Cuadro 8.1: Especificadores de FORMAT

Especificador	Descripción
`%s`	Cadena
`%d`	Entero (decimal)
`%f`	Número de punto flotante
`%x`	Hexadecimal
`%o`	Octal
`%c`	Carácter (desde código ASCII)
`%%`	Signo % literal

Ancho y Precisión:

- `%10s` - Cadena con ancho mínimo 10 (alineada a la derecha)
- `%-10s` - Cadena con ancho mínimo 10 (alineada a la izquierda)
- `%.2f` - Punto flotante con 2 decimales
- `%8.2f` - Ancho 8, 2 decimales

Ejemplos de FORMAT

```
SET name "Alice"
SET age 30
SET height 1.68
SET score 95.5

# Basic formatting
PUTS [FORMAT "Name: %s, Age: %d" $name $age]

# Floating-point precision
PUTS [FORMAT "Height: %.2f meters" $height]
PUTS [FORMAT "Score: %.1f%%" $score]

# Width and alignment
PUTS [FORMAT "%10s | %5d | %6.2f" $name $age $height]
PUTS [FORMAT "%-10s | %-5d | %-6.2f" $name $age $height]

# Creating tables
PUTS [FORMAT "%-10s %8s %8s" "Name" "Age" "Height"]
PUTS [FORMAT "%-10s %8d %8.2f" "Alice" 30 1.68]
PUTS [FORMAT "%-10s %8d %8.2f" "Bob" 25 1.75]

# Numbers in different bases
SET num 255
PUTS [FORMAT "Decimal: %d" $num]
PUTS [FORMAT "Hex: %x" $num]
```

```
PUTS [FORMAT "Octal: %o" $num]
```

Salida:

```
Name: Alice, Age: 30
Height: 1.68 meters
Score: 95.5%
     Alice |    30 |   1.68
Alice      | 30    | 1.68
Name             Age   Height
Alice             30     1.68
Bob               25     1.75
Decimal: 255
Hex: ff
Octal: 377
```

8.3.2. SCAN: Analizar Entrada Formateada

SCAN es lo opuesto a **FORMAT**—extrae valores de una cadena formateada.

Sintaxis:

```
SCAN string format var1 var2 ...
```

Ejemplos de SCAN

```
# Parse structured data
SET data "John 25 180.5"
SCAN $data "%s %d %f" name age height

PUTS "Name: $name"
PUTS "Age: $age"
PUTS "Height: $height"

# Parse date
SET date "2025-10-22"
SCAN $date "%d-%d-%d" year month day
PUTS "Year: $year, Month: $month, Day: $day"

# Parse key=value pairs
SET config "timeout=30"
SCAN $config "%\[^=\]=%d" key value
PUTS "Key: $key, Value: $value"

# Count items parsed
SET count [SCAN "42 3.14 hello" "%d %f %s" a b c]
PUTS "Parsed $count items"
```

Salida:

```
Name: John
Age: 25
Height: 180.5
Year: 2025, Month: 10, Day: 22
```

```
Key: timeout, Value: 30
Parsed 3 items
```

8.4. Ejemplos Prácticos

8.4.1. Validador de Email

Validación Simple de Email

```
PROC is_valid_email WITH email DO
  # Check for @ symbol
  SET at_pos [STRING FIRST "@" $email]
  IF $at_pos == -1 THEN
    RETURN 0
  END

  # Check for dot after @
  SET dot_pos [STRING FIRST "." $email $at_pos]
  IF $dot_pos == -1 THEN
    RETURN 0
  END

  # Basic pattern match
  IF [STRING MATCH "*@*.*" $email] THEN
    RETURN 1
  END

  RETURN 0
END

# Test the validator
SET emails [LIST "user@example.com" "invalid.email"
    "test@domain.co.uk"]
FOREACH email IN $emails DO
  IF [is_valid_email $email] THEN
    PUTS "$email - VALID"
  ELSE
    PUTS "$email - INVALID"
  END
END
```

Salida:

```
user@example.com - VALID
invalid.email - INVALID
test@domain.co.uk - VALID
```

8.4.2. Formateador de Texto

Centrar Texto

```
PROC center_text WITH text width DO
  SET len [STRING LENGTH $text]

  # Text is already too long
  IF $len >= $width THEN
    RETURN $text
  END

  # Calculate padding
  SET total_pad [EXPR $width - $len]
  SET left_pad [EXPR $total_pad / 2]

  # Create padding string
  SET padding ""
  FOR 1 TO $left_pad DO
    APPEND padding " "
  END

  # Return centered text
  RETURN $padding$text
END

# Create a title
SET title "BCL Manual"
SET line [center_text $title 40]
PUTS $line

SET border [STRING REPEAT "=" 40]
PUTS $border
```

Salida:

```
               BCL Manual
========================================
```

8.4.3. Contador de Palabras

Contar Palabras

```
PROC count_words WITH text DO
  # Trim whitespace
  SET clean [STRING TRIM $text]

  # Empty string has 0 words
  IF [STRING LENGTH $clean] = 0 THEN
    RETURN 0
  END

  # Count spaces and add 1
  SET count 1
  SET pos 0
```

```
  WHILE 1 DO
    SET pos [STRING FIRST " " $clean $pos]
    IF $pos == -1 THEN
      BREAK
    END
    INCR count
    INCR pos
  END

  RETURN $count
END

SET sentence "The quick brown fox jumps"
SET wc [count_words $sentence]
PUTS "Words: $wc"

SET text "  Multiple   spaces   between  "
PUTS "Words in '$text': [count_words $text]"
```

Salida:

```
Words: 5
Words in '  Multiple   spaces   between  ': 4
```

8.4.4. Verificador de Fortaleza de Contraseña

Verificar Fortaleza de Contraseña

```
PROC check_password WITH pass DO
  SET len [STRING LENGTH $pass]

  # Too short
  IF $len < 8 THEN
    PUTS "Weak: Too short (minimum 8 characters)"
    RETURN
  END

  # Check for digits
  SET has_digit 0
  FOR 0 TO $len-1 DO
    SET char [STRING INDEX $pass $__FOR]
    IF [STRING MATCH "\[0-9\]" $char] THEN
      SET has_digit 1
      BREAK
    END
  END

  # Check for uppercase
  SET upper [STRING TOUPPER $pass]
  SET has_upper [EXPR $pass != $upper]

  # Check for lowercase
  SET lower [STRING TOLOWER $pass]
  SET has_lower [EXPR $pass != $lower]
```

```
27
28    # Calculate strength
29    SET strength 0
30    IF $len >= 8 THEN
31      INCR strength
32    END
33    IF $len >= 12 THEN
34      INCR strength
35    END
36    IF $has_digit THEN
37      INCR strength
38    END
39    IF $has_upper THEN
40      INCR strength
41    END
42    IF $has_lower THEN
43      INCR strength
44    END
45
46    # Report
47    IF $strength <= 2 THEN
48      PUTS "Weak password"
49    ELSEIF $strength <= 3 THEN
50      PUTS "Medium password"
51    ELSE
52      PUTS "Strong password"
53    END
54  END
55
56  check_password "hello"
57  check_password "hello123"
58  check_password "Hello123"
59  check_password "MyP@ssw0rd2025"
```

Salida:

```
Weak: Too short (minimum 8 characters)
Medium password
Strong password
Strong password
```

8.5. Patrones Comunes de Cadenas

8.5.1. Construir Cadenas Eficientemente

Técnicas de Construcción de Cadenas

```
# Method 1: Using APPEND (efficient for loops)
SET result ""
FOR 1 TO 5 DO
  APPEND result "Line " $__FOR "\n"
END
PUTS $result
```

```
# Method 2: Using STRING CAT
SET str1 "Hello"
SET str2 "World"
SET combined [STRING CAT $str1 " " $str2]
PUTS $combined

# Method 3: Building with FORMAT
SET name "Alice"
SET age 30
SET message [FORMAT "%s is %d years old" $name $age]
PUTS $message
```

8.5.2. Limpieza de Cadenas

Limpiar Entrada de Usuario

```
PROC clean_input WITH text DO
  # Remove leading/trailing whitespace
  SET clean [STRING TRIM $text]

  # Convert to lowercase for consistency
  SET clean [STRING TOLOWER $clean]

  # Remove extra internal spaces
  WHILE [STRING FIRST "  " $clean] != -1 DO
    SET pos [STRING FIRST "  " $clean]
    SET clean [STRING REPLACE $clean $pos $pos+1 " "]
  END

  RETURN $clean
END

SET input "  HELLO    WORLD  "
SET clean [clean_input $input]
PUTS "Original: '$input'"
PUTS "Cleaned:  '$clean'"
```

Salida:

```
Original: '  HELLO    WORLD  '
Cleaned:  'hello world'
```

Consejo

Para procesamiento complejo de cadenas, considera usar expresiones regulares (Capítulo 10) para coincidencia y reemplazo de patrones más potentes.

Capítulo 9

Archivos

BCL proporciona capacidades completas de E/S de archivos.

9.1. Abrir y Cerrar Archivos

Cuadro 9.1: Modos de Apertura de Archivos

Modo	Descripción
`R`	Lectura (el archivo debe existir)
`W`	Escritura (crea o trunca)
`A`	Agregar (crea si es necesario)
`RW`	Lectura y escritura

E/S Básica de Archivos

```
# Write to file
SET fh [OPEN "output.txt" W]
PUTS $fh "Line 1"
PUTS $fh "Line 2"
CLOSE $fh

# Read from file
SET fh [OPEN "output.txt" R]
SET content [READ $fh]
CLOSE $fh

PUTS "File content:"
PUTS $content
```

9.2. Leer Archivos Línea por Línea

Lectura Línea por Línea

```
SET fh [OPEN "data.txt" R]

SET linenum 0
WHILE [EOF $fh] = 0 DO
  SET line [GETS $fh]
  INCR linenum

  IF [STRING LENGTH $line] > 0 THEN
    PUTS "Line $linenum: $line"
  END
END

CLOSE $fh
```

9.3. Comandos de Archivo

Comandos FILE

```
SET filename "test.txt"

# Check existence
IF [FILE EXISTS $filename] THEN
  PUTS "File exists"

  # Get size
  SET size [FILE SIZE $filename]
  PUTS "Size: $size bytes"

  # Rename
  FILE RENAME $filename "test_backup.txt"

  # Delete
  # FILE DELETE "test_backup.txt"
END

# Get current directory
SET cwd [PWD]
PUTS "Current directory: $cwd"

# Find files matching pattern
SET txtfiles [GLOB "*.txt"]
PUTS "Text files: $txtfiles"
```

Capítulo 10

Expresiones Regulares

Las expresiones regulares (a menudo llamadas regex.º regexp") son patrones potentes usados para buscar, coincidir y manipular texto. Piensa en ellas como una herramienta sofisticada de "buscar y reemplazar"que puede coincidir con patrones complejos en lugar de solo texto exacto.

10.1. ¿Qué son las Expresiones Regulares?

Imagina que quieres encontrar todos los números de teléfono en un documento, o validar que una dirección de correo electrónico esté formateada correctamente, o reemplazar todas las fechas de un formato a otro. Las expresiones regulares hacen estas tareas fáciles.

Analogía del Mundo Real

Las expresiones regulares son como describir algo sin conocer su forma exacta:

- Çualquier palabra que comience con 'cat' coincide con çat", çats", çategory"
- Üna secuencia de dígitos coincide con "123", "4567", "999"
- "Texto entre comillas coincide con "hello", "goodbye"

Nota

Si eres nuevo en programación, las expresiones regulares pueden parecer crípticas al principio. ¡No te preocupes! Comenzaremos con patrones simples y construiremos hasta los más complejos.

10.2. Bloques de Construcción de Patrones Básicos

Las expresiones regulares se construyen a partir de piezas simples. Aprendámoslas una a la vez.

10.2.1. Caracteres Literales

El patrón más simple es solo texto normal—coincide exactamente con lo que escribes.

Coincidencia Literal

```
SET text "The cat sat on the mat"

# Match the word "cat"
IF [REGEXP "cat" $text] THEN
  PUTS "Found 'cat' in the text"
END

# Match "mat"
IF [REGEXP "mat" $text] THEN
  PUTS "Found 'mat' in the text"
END

# This won't match because case matters
IF [REGEXP "Cat" $text] THEN
  PUTS "Found 'Cat'"
ELSE
  PUTS "'Cat' not found (wrong case)"
END
```

Salida:

```
Found 'cat' in the text
Found 'mat' in the text
'Cat' not found (wrong case)
```

10.2.2. Caracteres Especiales - Los Comodines

Algunos caracteres tienen significados especiales en las expresiones regulares:

Cuadro 10.1: Caracteres Básicos de Expresiones Regulares

Símbolo	Significado
`.`	Coincide con cualquier carácter individual
`*`	Coincide con 0 o más del carácter anterior
`+`	Coincide con 1 o más del carácter anterior
`?`	Coincide con 0 o 1 del carácter anterior
`^`	Coincide con el inicio de la cadena
`$`	Coincide con el final de la cadena
`|`	Operador OR (coincide con izquierda o derecha)
`(...)`	Agrupa patrones juntos
`[...]`	Coincide con cualquier carácter entre corchetes
`\`	Escapa caracteres especiales

10.2.3. El Punto (.) - Cualquier Carácter

El punto coincide con cualquier carácter individual excepto nueva línea.

Usando el Punto

```
# Match "c.t" - c, any character, then t
SET words [LIST "cat" "cot" "cut" "cart" "ct"]

FOREACH word IN $words DO
  IF [REGEXP "c.t" $word] THEN
    PUTS "$word matches c.t"
  ELSE
    PUTS "$word does NOT match c.t"
  END
END
```

Salida:

```
cat matches c.t
cot matches c.t
cut matches c.t
cart matches c.t
ct does NOT match c.t
```

Explicación: El patrón `c.t` requiere exactamente un carácter entre 'c' y 't'. çartçoincide porque contiene çar- "t- çart"que tiene el patrón.

10.2.4. Clases de Caracteres [...]

Los corchetes coinciden con cualquier carácter de un conjunto.

Clases de Caracteres

```
# Match c[aou]t - cat, cot, or cut
SET words [LIST "cat" "cot" "cut" "cit" "cet"]

FOREACH word IN $words DO
  IF [REGEXP "c\[aou\]t" $word] THEN
    PUTS "$word matches"
  END
END

# Match any digit [0-9]
SET text "I have 5 apples"
IF [REGEXP "\[0-9\]" $text] THEN
  PUTS "Contains a digit"
END

# Match any letter [a-z] or [A-Z]
IF [REGEXP "\[a-z\]" $text] THEN
  PUTS "Contains lowercase letters"
END
```

Salida:

```
cat matches
cot matches
cut matches
```

```
Contains a digit
Contains lowercase letters
```

Consejo

Clases de caracteres comunes:

- `[0-9]` - cualquier dígito
- `[a-z]` - cualquier letra minúscula
- `[A-Z]` - cualquier letra mayúscula
- `[a-zA-Z]` - cualquier letra
- `[^0-9]` - cualquier cosa que NO sea un dígito

10.2.5. Repetición: *, +, ?

Estos indican cuántas veces un patrón debe repetirse.

Cuadro 10.2: Operadores de Repetición

Operador	Significado
`*`	0 o más veces (opcional, puede repetir)
`+`	1 o más veces (debe aparecer al menos una vez)
`?`	0 o 1 vez (opcional, aparece una vez o no)
`{n}`	Exactamente n veces
`{n,}`	n o más veces
`{n,m}`	Entre n y m veces

Ejemplos de Repetición

```
# Match one or more digits
SET texts [LIST "abc" "123" "abc123" "a1b2c3"]

FOREACH text IN $texts DO
  IF [REGEXP "\[0-9\]+" $text] THEN
    PUTS "$text contains numbers"
  END
END

# Match optional minus sign followed by digits: -?[0-9]+
SET numbers [LIST "123" "-456" "78" "-9"]
FOREACH num IN $numbers DO
  IF [REGEXP "^-?\[0-9\]+$" $num] THEN
    PUTS "$num is a valid number"
  END
END

# Match "color" or "colour"
```

```
IF [REGEXP "colou?r" "color"] THEN
  PUTS "Matches 'color'"
END
IF [REGEXP "colou?r" "colour"] THEN
  PUTS "Matches 'colour'"
END
```

Salida:

```
123 contains numbers
abc123 contains numbers
a1b2c3 contains numbers
123 is a valid number
-456 is a valid number
78 is a valid number
-9 is a valid number
Matches 'color'
Matches 'colour'
```

10.2.6. Anclas: ^ y $

Las anclas coinciden con posiciones, no con caracteres.

- ^ coincide con el inicio de la cadena
- $ coincide con el final de la cadena

Usando Anclas

```
SET text "hello world"

# Must start with "hello"
IF [REGEXP "^hello" $text] THEN
  PUTS "Starts with 'hello'"
END

# Must end with "world"
IF [REGEXP "world$" $text] THEN
  PUTS "Ends with 'world'"
END

# Must be EXACTLY "hello world" (nothing before or after)
IF [REGEXP "^hello world$" $text] THEN
  PUTS "Exact match"
END

# Won't match - has extra text
SET text2 "say hello world now"
IF [REGEXP "^hello world$" $text2] THEN
  PUTS "Exact match"
ELSE
  PUTS "Not an exact match - has extra text"
END
```

Salida:

```
Starts with 'hello'
Ends with 'world'
Exact match
Not an exact match - has extra text
```

10.2.7. Clases de Caracteres Abreviadas

BCL proporciona atajos para patrones comunes:

Cuadro 10.3: Clases de Caracteres Abreviadas

Abreviación	Equivalente a
\d	[0-9] - cualquier dígito
\D	[^0-9] - cualquier no-dígito
\w	[a-zA-Z0-9_] - carácter de palabra
\W	Carácter no-palabra
\s	Espacio en blanco (espacio, tab, nueva línea)
\S	No-espacio en blanco

Ejemplos de Abreviaciones

```
# Find digits with \d
SET text "Room 101 is on floor 5"
IF [REGEXP "\\d+" $text] THEN
  PUTS "Found numbers"
END

# Find words with \w
IF [REGEXP "\\w+" $text] THEN
  PUTS "Found word characters"
END

# Validate format: word space word
SET input "hello world"
IF [REGEXP "^\\w+\\s+\\w+$" $input] THEN
  PUTS "Valid format: two words separated by space"
END
```

Salida:

```
Found numbers
Found word characters
Valid format: two words separated by space
```

Advertencia

En BCL, necesitas escapar las barras invertidas en cadenas. Escribe `\\d` para representar `\d` en el patrón.

10.3. El Comando REGEXP

REGEXP verifica si un patrón coincide y puede extraer porciones coincidentes.

Sintaxis:

```
REGEXP pattern string                    # Returns 1 if match, 0 if not
REGEXP pattern string matchvar           # Store entire match
REGEXP pattern string matchvar subvar... # Store submatches
```

10.3.1. Coincidencia Básica

Coincidencia Simple de Patrones

```
SET email "user@example.com"

# Check if it looks like an email
IF [REGEXP "@" $email] THEN
  PUTS "Contains @ symbol"
END

# Check for email pattern: word @ word . word
IF [REGEXP "\\w+@\\w+\\.\\w+" $email] THEN
  PUTS "Looks like a valid email"
END

# Validate phone number: exactly 10 digits
SET phone "5551234567"
IF [REGEXP "^\\d\{10\}$" $phone] THEN
  PUTS "Valid 10-digit phone number"
END
```

Salida:

```
Contains @ symbol
Looks like a valid email
Valid 10-digit phone number
```

10.3.2. Capturar Coincidencias

Usa paréntesis `(...)` para capturar partes de la coincidencia.

Extraer Información

```
# Extract year from date
SET date "Today is 2025-10-22"
REGEXP "(\\d\{4\})-(\\d\{2\})-(\\d\{2\})" $date MATCH year month
    day
PUTS "Year: $year"
PUTS "Month: $month"
PUTS "Day: $day"

# Extract name and extension from filename
SET filename "document.pdf"
REGEXP "(.+)\\.(\\w+)$" $filename MATCH name ext
```

```
PUTS "Name: $name"
PUTS "Extension: $ext"

# Extract email parts
SET email "john.doe@example.com"
REGEXP "(.+)@(.+)" $email MATCH username domain
PUTS "Username: $username"
PUTS "Domain: $domain"
```

Salida:

```
Year: 2025
Month: 10
Day: 22
Name: document
Extension: pdf
Username: john.doe
Domain: example.com
```

10.3.3. Opciones de REGEXP

Cuadro 10.4: Opciones de REGEXP

Opción	Descripción
-nocase	Coincidencia insensible a mayúsculas/minúsculas
-line	Tratar cadena como múltiples líneas (^ y $ coinciden con inicio/fin de línea)
-lineanchor	Similar a -line
-expanded	Ignorar espacios en blanco en patrón (para legibilidad)

Coincidencia Insensible a Mayúsculas

```
SET text "Hello World"

# Case-sensitive (won't match)
IF [REGEXP "hello" $text] THEN
  PUTS "Found (sensitive)"
ELSE
  PUTS "Not found (sensitive)"
END

# Case-insensitive (will match)
IF [REGEXP -nocase "hello" $text] THEN
  PUTS "Found (insensitive)"
END
```

Salida:

```
Not found (sensitive)
Found (insensitive)
```

10.4. El Comando REGSUB

REGSUB reemplaza texto que coincide con un patrón.

Sintaxis:

```
REGSUB pattern string replacement          # Replace first match
REGSUB pattern string replacement ALL      # Replace all matches
```

10.4.1. Reemplazo Básico

Reemplazos Simples

```
SET text "Hello, World!"

# Replace first occurrence
SET result [REGSUB "World" $text "BCL"]
PUTS $result  # Hello, BCL!

# Replace all occurrences
SET text2 "foo bar foo baz foo"
SET result2 [REGSUB "foo" $text2 "XXX" ALL]
PUTS $result2  # XXX bar XXX baz XXX

# Remove all digits
SET text3 "Room 101 is on floor 5"
SET clean [REGSUB "\\d+" $text3 "" ALL]
PUTS $clean  # Room  is on floor
```

Salida:

```
Hello, BCL!
XXX bar XXX baz XXX
Room  is on floor
```

10.4.2. Usar Grupos Capturados en Reemplazo

Puedes referenciar grupos capturados en el reemplazo usando `&` o `\1`, `\2`, etc.

Reemplazos Avanzados

```
# Swap first and last name
SET name "John Doe"
SET swapped [REGSUB "(\\w+) (\\w+)" $name "\\2, \\1"]
PUTS $swapped  # Doe, John

# Format phone number: 5551234567 -> (555) 123-4567
SET phone "5551234567"
SET formatted [REGSUB "(\\d\{3\})(\\d\{3\})(\\d\{4\})" $phone
    "(\\1) \\2-\\3"]
```

```
PUTS $formatted  # (555) 123-4567

# Add "http://" to URLs that don't have it
SET url "example.com"
IF [REGEXP "^http" $url] = 0 THEN
  SET url [REGSUB "^" $url "http://"]
END
PUTS $url  # http://example.com
```

Salida:

```
Doe, John
(555) 123-4567
http://example.com
```

10.5. Ejemplos Prácticos

10.5.1. Validador de Email (Avanzado)

Validación Completa de Email

```
PROC validate_email WITH email DO
  # Basic pattern: user@domain.tld
  SET pattern "^\\w+(\\.\\w+)*@\\w+(\\.\\w+)+$"

  IF [REGEXP $pattern $email] THEN
    PUTS "$email is VALID"
    RETURN 1
  ELSE
    PUTS "$email is INVALID"
    RETURN 0
  END
END

# Test cases
validate_email "user@example.com"
validate_email "john.doe@company.co.uk"
validate_email "invalid@"
validate_email "@invalid.com"
validate_email "no-at-sign.com"
```

Salida:

```
user@example.com is VALID
john.doe@company.co.uk is VALID
invalid@ is INVALID
@invalid.com is INVALID
no-at-sign.com is INVALID
```

10.5.2. Validador de Fortaleza de Contraseña

Verificación de Contraseña Basada en Regex

```
PROC check_password_strength WITH password DO
  SET score 0

  # Check length
  IF [STRING LENGTH $password] >= 8 THEN
    INCR score
  END

  # Check for lowercase
  IF [REGEXP "\[a-z\]" $password] THEN
    INCR score
  END

  # Check for uppercase
  IF [REGEXP "\[A-Z\]" $password] THEN
    INCR score
  END

  # Check for digits
  IF [REGEXP "\\d" $password] THEN
    INCR score
  END

  # Check for special characters
  IF [REGEXP "\[!@#$%^&*\]" $password] THEN
    INCR score
  END

  # Report strength
  IF $score < 3 THEN
    RETURN "Weak"
  ELSEIF $score < 5 THEN
    RETURN "Medium"
  ELSE
    RETURN "Strong"
  END
END

SET passwords [LIST "hello" "Hello123" "MyP@ss123" "Complex$Pass9"]
FOREACH pass IN $passwords DO
  SET strength [check_password_strength $pass]
  PUTS "$pass: $strength"
END
```

Salida:

```
hello: Weak
Hello123: Medium
MyP@ss123: Strong
Complex$Pass9: Strong
```

10.5.3. Formatear Números de Teléfono

Normalizar Números de Teléfono

```
PROC format_phone WITH phone DO
  # Remove all non-digits
  SET clean [REGSUB "\\D" $phone "" ALL]

  # Check if we have 10 digits
  IF [STRING LENGTH $clean] != 10 THEN
    RETURN "Invalid phone number"
  END

  # Format as (XXX) XXX-XXXX
  SET formatted [REGSUB "(\\d\{3\})(\\d\{3\})(\\d\{4\})" $clean
     "(\\1) \\2-\\3"]
  RETURN $formatted
END

# Test with various formats
SET phones [LIST "5551234567" "555-123-4567" "(555) 123-4567"
   "555.123.4567"]
FOREACH phone IN $phones DO
  SET formatted [format_phone $phone]
  PUTS "$phone -> $formatted"
END
```

Salida:

```
5551234567 -> (555) 123-4567
555-123-4567 -> (555) 123-4567
(555) 123-4567 -> (555) 123-4567
555.123.4567 -> (555) 123-4567
```

10.6. Patrones Comunes de Expresiones Regulares

Aquí hay una referencia de patrones útiles para tareas comunes:

Cuadro 10.5: Patrones Regex Comunes

Patrón	Descripción
`^\d{4}-\d{2}-\d{2}$`	Fecha (AAAA-MM-DD)
`\w+@\w+\.\w+`	Email simple
`^\d{3}-\d{3}-\d{4}$`	Teléfono (XXX-XXX-XXXX)
`^[01]?\d\d?$`	Número 0-199
`\b\w{3}\b`	Palabra exactamente de 3 letras
`https?://.*`	URL HTTP o HTTPS
`^\s*$`	Vacío o solo espacios en blanco
`\d+\.\d{2}`	Decimal con 2 lugares

Consejo

Las expresiones regulares pueden volverse complejas. Comienza simple y construye gradualmente. Prueba tus patrones con varias entradas para asegurar que funcionen como se espera.

Advertencia

Ten cuidado con patrones como `.*` (coincidir con cualquier cosa) ya que pueden coincidir con más de lo que esperas. Usa patrones específicos cuando sea posible.

Capítulo 11

Tiempo y Fecha

El comando `CLOCK` proporciona funcionalidad de tiempo y fecha.

Operaciones de Tiempo

```
# Get current time
SET now [CLOCK SECONDS]
PUTS "Timestamp: $now"

# Format timestamp
SET formatted [CLOCK FORMAT $now FORMAT "%Y-%m-%d %H:%M:%S"]
PUTS "Date: $formatted"

# Parse date string
SET timestamp [CLOCK SCAN "2025-12-25 00:00:00"]
PUTS "Christmas: $timestamp"

# Add time
SET tomorrow [CLOCK ADD $now 1 day]
SET next_week [CLOCK ADD $now 7 days]
PUTS "Tomorrow: [CLOCK FORMAT $tomorrow]"
```

Capítulo 12

Interacción con el Sistema

BCL proporciona varios comandos para interactuar con el sistema operativo, ejecutar código dinámicamente, cargar archivos externos y controlar el flujo del programa. Este capítulo cubre estas poderosas características a nivel de sistema.

12.1. Ejecución Dinámica de Código

12.1.1. EVAL - Ejecutar Cadena como Código

EVAL evalúa una cadena como código BCL, permitiéndote ejecutar código que se construye en tiempo de ejecución.

Sintaxis:

```
EVAL string
```

Uso Básico de EVAL

```
# Execute code from a string
SET command "PUTS 'Hello from EVAL'"
EVAL $command

# Build code dynamically
SET varname "result"
SET value 42
SET code "SET $varname $value"
EVAL $code
PUTS "result = $result"  # Prints: result = 42

# Evaluate expressions
SET expr "5 + 3 * 2"
SET answer [EVAL "EXPR $expr"]
PUTS "Answer: $answer"  # Prints: Answer: 11
```

Salida:

```
Hello from EVAL
result = 42
Answer: 11
```

Advertencia

EVAL ejecuta código en el alcance actual con acceso a todas las variables. Ten mucho cuidado al evaluar entrada de usuario, ya que puede ejecutar código arbitrario. ¡Nunca uses **EVAL** con datos no confiables!

12.1.2. Ejemplos Prácticos de EVAL

Despachador de Comandos Dinámico

```
PROC dispatch_command WITH cmd DO
  # Map user commands to BCL code
  SWITCH $cmd
    CASE "greet" DO
      EVAL "PUTS 'Hello, user!'"
    END
    CASE "time" DO
      EVAL "PUTS [CLOCK FORMAT [CLOCK SECONDS]]"
    END
    CASE "random" DO
      EVAL "PUTS [EXPR rand() * 100]"
    END
    DEFAULT DO
      PUTS "Unknown command: $cmd"
    END
  END
END

dispatch_command "greet"
dispatch_command "time"
dispatch_command "random"
```

Calculadora Simple con EVAL

```
PROC calculate WITH expression DO
  # Add EXPR wrapper and evaluate
  SET code "EXPR $expression"

  # Use error handling (if available)
  SET result [EVAL $code]

  RETURN $result
END

PUTS "5 + 3 = [calculate "5 + 3"]"
PUTS "10 * 2.5 = [calculate "10 * 2.5"]"
PUTS "sqrt(16) = [calculate "sqrt(16)"]"
```

Salida:

```
5 + 3 = 8
10 * 2.5 = 25.0
sqrt(16) = 4.0
```

12.2. Cargar Código Externo

12.2.1. SOURCE - Cargar y Ejecutar Archivos

SOURCE carga y ejecuta código BCL desde un archivo externo, permitiéndote organizar programas grandes en módulos.

Sintaxis:

```
SOURCE filepath
```

Usar SOURCE

Crea un archivo `library.bcl`:

```
# library.bcl - Math utilities
PROC square WITH x DO
  RETURN [EXPR $x * $x]
END

PROC cube WITH x DO
  RETURN [EXPR $x * $x * $x]
END

SET PI 3.14159
```

Script principal:

```
# Load the library
SOURCE "library.bcl"

# Use functions and variables from library
PUTS "5 squared = [square 5]"
PUTS "3 cubed = [cube 3]"
PUTS "PI = $PI"
```

Salida:

```
5 squared = 25
3 cubed = 27
PI = 3.14159
```

Consejo

Usa **SOURCE** para:

- Dividir programas grandes en archivos manejables
- Crear módulos de biblioteca reutilizables
- Compartir código común entre múltiples scripts
- Mantener configuración en archivos separados

12.2.2. Ejemplos Prácticos de SOURCE

Archivo de Configuración

Crea `config.bcl`:

```
# Application configuration
SET APP_NAME "MyApp"
SET APP_VERSION "1.0.0"
SET DEBUG_MODE 1
SET MAX_USERS 100
```

Aplicación principal:

```
# Load configuration
SOURCE "config.bcl"

PUTS "=== $APP_NAME v$APP_VERSION ==="
PUTS "Debug mode: $DEBUG_MODE"
PUTS "Max users: $MAX_USERS"

IF $DEBUG_MODE THEN
  PUTS "[DEBUG] Configuration loaded successfully"
END
```

Salida:

```
=== MyApp v1.0.0 ===
Debug mode: 1
Max users: 100
[DEBUG] Configuration loaded successfully
```

Sistema de Módulos

Crea `string_utils.bcl`:

```
PROC reverse_string WITH str DO
  RETURN [STRING REVERSE $str]
END

PROC count_vowels WITH str DO
  SET count 0
  SET len [STRING LENGTH $str]
  FOR 0 TO $len-1 DO
    SET char [STRING INDEX $str $__FOR]
    IF [REGEXP -nocase "\[aeiou\]" $char] THEN
      INCR count
    END
  END
  RETURN $count
END
```

Script principal:

```
SOURCE "string_utils.bcl"

SET text "Hello World"
```

```
PUTS "Original: $text"
PUTS "Reversed: [reverse_string $text]"
PUTS "Vowels: [count_vowels $text]"
```

Salida:

```
Original: Hello World
Reversed: dlroW olleH
Vowels: 3
```

12.3. Temporización y Retrasos

12.3.1. AFTER - Pausar Ejecución

AFTER suspende la ejecución por un número especificado de milisegundos.

Sintaxis:

```
AFTER milliseconds
```

Usar AFTER

```
PUTS "Starting..."
AFTER 1000          # Wait 1 second (1000 ms)
PUTS "1 second later"

AFTER 500           # Wait 0.5 seconds
PUTS "0.5 seconds later"

AFTER 2000          # Wait 2 seconds
PUTS "Done!"
```

Salida: (con retrasos)

```
Starting...
[pausa de 1 segundo]
1 second later
[pausa de 0.5 segundos]
0.5 seconds later
[pausa de 2 segundos]
Done!
```

Animación de Progreso

```
PROC show_progress WITH steps DO
  PUTSN "Progress: "

  FOR 1 TO $steps DO
    PUTSN "."
    AFTER 200
  END

  PUTS " Done!"
```

```
END

PUTS "Loading"
show_progress 10
PUTS "Complete!"
```

Salida:

```
Loading
Progress: .......... Done!
Complete!
```

Temporizador de Cuenta Regresiva

```
PROC countdown WITH seconds DO
  FOR $seconds TO 1 STEP -1 DO
    PUTS "Time remaining: $__FOR seconds"
    AFTER 1000
  END
  PUTS "Time's up!"
END

countdown 5
```

Salida:

```
Time remaining: 5 seconds
Time remaining: 4 seconds
Time remaining: 3 seconds
Time remaining: 2 seconds
Time remaining: 1 seconds
Time's up!
```

12.4. Ejecución de Comandos del Sistema

12.4.1. EXEC - Ejecutar Comandos del Sistema

EXEC ejecuta un comando del sistema operativo y devuelve su salida.
Sintaxis:

```
EXEC command [arg1 arg2 ...]
```

Advertencia

EXEC típicamente solo está disponible en sistemas PC/desktop, no en microcontroladores embebidos. También plantea riesgos de seguridad si se usa con entrada no confiable.

Uso Básico de EXEC

```
# List files (Linux/Unix)
SET files [EXEC "ls"]
```

```
PUTS "Files:\n$files"

# Get current date
SET date [EXEC "date"]
PUTS "Current date: $date"

# Echo text
SET output [EXEC "echo" "Hello from shell"]
PUTS $output

# Count files
SET count [EXEC "ls" "-1" "|" "wc" "-l"]
PUTS "File count: $count"
```

Ejemplos Prácticos de EXEC

```
# Check if a file exists (using shell command)
PROC file_exists_shell WITH filename DO
  # Use test command
  SET result [EXEC "test" "-f" $filename "&&" "echo" "1" "||"
      "echo" "0"]
  RETURN [STRING TRIM $result]
END

# Get disk usage
PROC get_disk_usage DO
  SET usage [EXEC "df" "-h" "."]
  RETURN $usage
END

# Backup a file
PROC backup_file WITH filename DO
  SET timestamp [CLOCK FORMAT [CLOCK SECONDS] FORMAT
      "%Y%m%d_%H%M%S"]
  SET backup_name "$filename.backup_$timestamp"
  EXEC "cp" $filename $backup_name
  PUTS "Backed up to: $backup_name"
END

backup_file "important.txt"
```

12.5. Variables de Entorno

12.5.1. ENV - Acceder a Variables de Entorno

ENV lee variables de entorno del sistema operativo.

Sintaxis:

```
ENV variable_name
```

Leer Variables de Entorno

```
# Get home directory
SET home [ENV HOME]
PUTS "Home directory: $home"

# Get current user
SET user [ENV USER]
PUTS "Current user: $user"

# Get PATH
SET path [ENV PATH]
PUTS "PATH: $path"

# Check if a variable exists
SET editor [ENV EDITOR]
IF [STRING LENGTH $editor] > 0 THEN
  PUTS "Default editor: $editor"
ELSE
  PUTS "No EDITOR set, using vi"
  SET editor "vi"
END
```

Salida:

```
Home directory: /home/user
Current user: user
PATH: /usr/local/bin:/usr/bin:/bin
Default editor: vim
```

Rutas Multiplataforma

```
PROC get_temp_dir DO
  # Try different environment variables
  SET temp [ENV TMPDIR]
  IF [STRING LENGTH $temp] = 0 THEN
    SET temp [ENV TEMP]
  END
  IF [STRING LENGTH $temp] = 0 THEN
    SET temp [ENV TMP]
  END
  IF [STRING LENGTH $temp] = 0 THEN
    SET temp "/tmp"  # Default for Unix
  END
  RETURN $temp
END

SET tempdir [get_temp_dir]
PUTS "Temp directory: $tempdir"
```

12.6. Argumentos de Línea de Comandos

12.6.1. ARGV - Obtener Argumentos del Script

ARGV devuelve la lista de argumentos de línea de comandos pasados al script.
Sintaxis:

```
ARGV
```

Procesar Argumentos

Guardar como `args.bcl`:

```
# Get arguments
SET args [ARGV]
SET count [LLENGTH $args]

PUTS "Received $count argument(s):"

IF $count == 0 THEN
  PUTS "  (none)"
ELSE
  SET i 1
  FOREACH arg IN $args DO
    PUTS "  $i. $arg"
    INCR i
  END
END
```

Ejecutar como: `bcl args.bcl one two three`
Salida:

```
Received 3 argument(s):
  1. one
  2. two
  3. three
```

Herramienta de Línea de Comandos

```
# greet.bcl - A simple greeting tool
SET args [ARGV]
SET argc [LLENGTH $args]

IF $argc == 0 THEN
  PUTS "Usage: bcl greet.bcl <name> [title]"
  EXIT 1
END

SET name [LINDEX $args 0]

IF $argc >= 2 THEN
  SET title [LINDEX $args 1]
  PUTS "Hello, $title $name!"
ELSE
  PUTS "Hello, $name!"
```

```
END
```

Uso:

```
$ bcl greet.bcl Alice
Hello, Alice!

$ bcl greet.bcl Bob Dr.
Hello, Dr. Bob!
```

12.7. Terminación del Programa

12.7.1. EXIT - Terminar Ejecución

EXIT termina el script BCL o intérprete y devuelve un código de salida al sistema operativo.
Sintaxis:

```
EXIT [code]
```

El `code` es opcional:

- `0` indica éxito (predeterminado)
- No-cero indica un error o condición específica

Usar EXIT

```
SET args [ARGV]

IF [LLENGTH $args] == 0 THEN
  PUTS "Error: No arguments provided"
  EXIT 1    # Exit with error code
END

# Process arguments...
PUTS "Processing..."

# Success
EXIT 0
```

Códigos de Salida para Diferentes Errores

```
PROC validate_file WITH filename DO
  # Check if file exists
  IF [FILE EXISTS $filename] = 0 THEN
    PUTS "Error: File '$filename' not found"
    EXIT 2    # File not found
  END

  # Check if file is readable
  # (assuming FILE READABLE command exists)
  # ...

```

```
  PUTS "File validated successfully"
END

SET args [ARGV]
IF [LLENGTH $args] == 0 THEN
  PUTS "Usage: script.bcl <filename>"
  EXIT 1    # Invalid usage
END

validate_file [LINDEX $args 0]
PUTS "Done!"
EXIT 0      # Success
```

12.8. Ejemplos Prácticos de Integración del Sistema

12.8.1. Lanzador de Scripts

Cargador Dinámico de Scripts

```
PROC load_plugin WITH name DO
  SET plugin_file "plugins/$name.bcl"

  IF [FILE EXISTS $plugin_file] THEN
    PUTS "Loading plugin: $name"
    SOURCE $plugin_file
    RETURN 1
  ELSE
    PUTS "Plugin not found: $name"
    RETURN 0
  END
END

# Load multiple plugins
SET plugins [LIST "database" "network" "utils"]
FOREACH plugin IN $plugins DO
  load_plugin $plugin
END
```

12.8.2. Shell Interactivo

Shell BCL Simple

```
PUTS "=== BCL Interactive Shell ==="
PUTS "Type 'exit' or 'quit' to exit"
PUTS ""

WHILE 1 DO
  PUTSN "BCL> "
  SET input [GETS stdin]

```

```
  # Trim input
  SET input [STRING TRIM $input]

  # Check for exit
  IF $input == "exit" OR $input == "quit" THEN
    PUTS "Goodbye!"
    BREAK
  END

  # Skip empty lines
  IF [STRING LENGTH $input] == 0 THEN
    CONTINUE
  END

  # Evaluate input as BCL code
  EVAL $input
END
```

12.8.3. Script de Construcción

Sistema de Construcción Automatizado

```
PROC run_command WITH description command DO
  PUTS ">>> $description"
  SET result [EXEC $command]
  PUTS $result
  RETURN [STRING LENGTH $result]
END

PROC build DO
  PUTS "===================================="
  PUTS "  Starting Build Process"
  PUTS "===================================="
  PUTS ""

  # Clean
  run_command "Cleaning old files..." "make clean"
  AFTER 500

  # Build
  run_command "Compiling..." "make"
  AFTER 500

  # Test
  run_command "Running tests..." "make test"
  AFTER 500

  PUTS ""
  PUTS "===================================="
  PUTS "  Build Complete!"
  PUTS "===================================="
END
```

```
31
32 # Check arguments
33 SET args [ARGV]
34 IF [LLENGTH $args] > 0 THEN
35   SET target [LINDEX $args 0]
36   IF $target == "build" THEN
37     build
38   ELSEIF $target == "clean" THEN
39     EXEC "make clean"
40   ELSE
41     PUTS "Unknown target: $target"
42     EXIT 1
43   END
44 ELSE
45   PUTS "Usage: build.bcl <build|clean>"
46   EXIT 1
47 END
```

12.9. Resumen de Comandos del Sistema

Cuadro 12.1: Comandos de Interacción con el Sistema

Comando	Descripción
`EVAL string`	Ejecutar cadena como código BCL
`SOURCE file`	Cargar y ejecutar archivo BCL externo
`AFTER ms`	Pausar ejecución por milisegundos
`EXEC cmd`	Ejecutar comando del sistema, devolver salida
`ENV name`	Obtener variable de entorno
`ARGV`	Obtener lista de argumentos de línea de comandos
`EXIT [code]`	Terminar programa con código de salida

Advertencia

Consideraciones de Seguridad:

- Nunca uses **`EVAL`** o **`EXEC`** con entrada de usuario no confiable
- Valida y sanitiza todos los datos externos antes de usarlos en comandos
- Ten cuidado con **`SOURCE`** para evitar cargar código malicioso
- Usa códigos de salida de manera consistente para mejor integración con scripts de shell

Consejo

Mejores Prácticas:

- Usa **`SOURCE`** para organizar proyectos grandes en módulos
- Almacena configuración en archivos separados cargados con **`SOURCE`**

- Usa variables de entorno para configuraciones específicas del sistema
- Proporciona mensajes de error útiles y códigos de salida apropiados
- Documenta tus módulos y bibliotecas claramente

Capítulo 13

Introspección

La introspección es la capacidad de un programa para examinar su propia estructura y estado mientras se ejecuta. BCL proporciona el comando `INFO`, que te permite inspeccionar variables, procedimientos y el entorno BCL mismo.

13.1. ¿Qué es la Introspección?

Piensa en la introspección como la capacidad de tu programa de "mirarse en el espejoz ver lo que contiene. Puede hacer preguntas como:

- "¿Existe una variable llamada 'x'?"
- "¿Qué procedimientos he definido?"
- "¿Cuáles son los parámetros de este procedimiento?"
- "¿Qué versión de BCL estoy ejecutando?"

Esto es increíblemente útil para:

- **Depuración** - Verificar si las variables existen antes de usarlas
- **Código dinámico** - Tomar decisiones basadas en lo que está disponible
- **Manejo de errores** - Verificar precondiciones antes de operaciones
- **Herramientas de desarrollo** - Construir depuradores, perfiladores o IDEs

13.2. El Comando INFO

El comando `INFO` es tu puerta de entrada a la introspección. Tiene muchos subcomandos, cada uno proporcionando información diferente sobre el estado de tu programa.

Sintaxis General:

```
INFO subcommand [arguments...]
```

13.3. Verificar Variables

13.3.1. INFO EXISTS - ¿Existe una Variable?

INFO EXISTS verifica si una variable ha sido creada y asignada un valor.
Sintaxis:

```
INFO EXISTS varname
```

Devuelve 1 si la variable existe, 0 si no existe.

Verificar Existencia de Variable

```
SET username "Alice"

# Check if variable exists
IF [INFO EXISTS username] THEN
  PUTS "username exists: $username"
ELSE
  PUTS "username doesn't exist"
END

# Check non-existent variable
IF [INFO EXISTS password] THEN
  PUTS "password exists"
ELSE
  PUTS "password doesn't exist - need to set it"
  SET password "secret123"
END

# Now it exists
IF [INFO EXISTS password] THEN
  PUTS "password now exists"
END
```

Salida:

```
username exists: Alice
password doesn't exist - need to set it
password now exists
```

Consejo

Usa **INFO EXISTS** para evitar errores al acceder a variables opcionales o entrada de usuario que podría no haberse proporcionado.

13.3.2. INFO VARS - Listar Todas las Variables

INFO VARS devuelve una lista de todas las variables en el alcance actual.
Sintaxis:

```
INFO VARS               # All variables
INFO VARS pattern       # Variables matching pattern
```

Listar Variables

```
SET name "Alice"
SET age 30
SET city "New York"
SET temp_value 123

# List all variables
SET all_vars [INFO VARS]
PUTS "All variables: $all_vars"

# List variables matching pattern
SET temp_vars [INFO VARS "temp_*"]
PUTS "Temp variables: $temp_vars"

# List variables starting with specific letter
SET a_vars [INFO VARS "a*"]
PUTS "Variables starting with 'a': $a_vars"
```

Salida:

```
All variables: name age city temp_value
Temp variables: temp_value
Variables starting with 'a': age
```

13.3.3. INFO GLOBALS - Listar Variables Globales

`INFO GLOBALS` devuelve una lista de todas las variables globales.

Variables Globales

```
SET global_config "production"
SET global_debug 1

PROC test DO
  SET local_var "I'm local"

  # From inside proc, check globals
  SET globals [INFO GLOBALS]
  PUTS "Global variables: $globals"

  # Check locals
  SET locals [INFO LOCALS]
  PUTS "Local variables: $locals"
END

test
```

Salida:

```
Global variables: global_config global_debug
Local variables: local_var
```

13.3.4. INFO LOCALS - Listar Variables Locales

`INFO LOCALS` devuelve variables locales al procedimiento actual.

Inspección de Variables Locales

```
PROC calculate WITH x y DO
  SET sum [EXPR $x + $y]
  SET product [EXPR $x * $y]
  SET temp "working..."

  # Show all local variables (including parameters)
  SET locals [INFO LOCALS]
  PUTS "Local variables in procedure: $locals"

  # Show just local temp variables
  SET temps [INFO LOCALS "temp*"]
  PUTS "Temp variables: $temps"
END

calculate 5 3
```

Salida:

```
Local variables in procedure: x y sum product temp
Temp variables: temp
```

13.4. Inspeccionar Procedimientos

13.4.1. INFO PROCS - Listar Todos los Procedimientos

`INFO PROCS` devuelve una lista de todos los procedimientos definidos.

Sintaxis:

```
INFO PROCS              # All procedures
INFO PROCS pattern      # Procedures matching pattern
```

Listar Procedimientos

```
PROC calculate WITH x y DO
  RETURN [EXPR $x + $y]
END

PROC format_output WITH text DO
  PUTS "=== $text ==="
END

PROC helper_function DO
  # Helper code
END

# List all procedures
SET procs [INFO PROCS]
PUTS "All procedures:"
```

```
FOREACH proc IN $procs DO
  PUTS "  - $proc"
END

# List only helper procedures
SET helpers [INFO PROCS "helper_*"]
PUTS "\nHelper procedures: $helpers"
```

Salida:

```
All procedures:
  - calculate
  - format_output
  - helper_function

Helper procedures: helper_function
```

13.4.2. INFO ARGS - Obtener Parámetros del Procedimiento

INFO ARGS devuelve los nombres de parámetros para un procedimiento.
Sintaxis:

```
INFO ARGS procname
```

Inspeccionar Parámetros de Procedimiento

```
PROC greet WITH name @title DO
  IF [INFO EXISTS title] THEN
    PUTS "Hello, $title $name"
  ELSE
    PUTS "Hello, $name"
  END
END

# Get parameter list
SET params [INFO ARGS greet]
PUTS "Parameters of 'greet': $params"

# Check how many parameters
SET count [LLENGTH $params]
PUTS "Number of parameters: $count"
```

Salida:

```
Parameters of 'greet': name title
Number of parameters: 2
```

13.4.3. INFO BODY - Obtener Cuerpo del Procedimiento

INFO BODY devuelve el código actual (cuerpo) de un procedimiento.
Sintaxis:

```
INFO BODY procname
```

Examinar Código de Procedimiento

```
PROC add WITH a b DO
  SET result [EXPR $a + $b]
  RETURN $result
END

# Get the procedure body
SET body [INFO BODY add]
PUTS "Procedure 'add' contains:"
PUTS $body
```

Salida:

```
Procedure 'add' contains:
  SET result [EXPR $a + $b]
  RETURN $result
```

Consejo

`INFO BODY` es útil para depuración, crear documentación o implementar herramientas de análisis de código.

13.5. Información del Sistema

13.5.1. INFO BCLVERSION - Obtener Versión de BCL

`INFO BCLVERSION` devuelve la versión de BCL que estás ejecutando.

Verificación de Versión

```
SET version [INFO BCLVERSION]
PUTS "Running BCL version: $version"

# Version-specific features
IF [STRING MATCH "1.5*" $version] THEN
  PUTS "You have BCL 1.5.x - all features available"
ELSE
  PUTS "Consider upgrading to BCL 1.5 or newer"
END
```

Salida:

```
Running BCL version: 1.5.1
You have BCL 1.5.x - all features available
```

13.5.2. INFO COMMANDS - Listar Todos los Comandos Disponibles

`INFO COMMANDS` devuelve una lista de todos los comandos BCL disponibles en el intérprete actual.

Comandos Disponibles

```
# Get all commands
SET commands [INFO COMMANDS]
PUTS "Total commands available: [LLENGTH $commands]"

# Check if a specific command exists
IF [LSEARCH $commands "REGEXP"] != -1 THEN
  PUTS "REGEXP command is available"
END

# List string-related commands
SET string_cmds [LIST]
FOREACH cmd IN $commands DO
  IF [STRING MATCH "STRING*" $cmd] THEN
    SET string_cmds [LAPPEND $string_cmds $cmd]
  END
END
PUTS "String commands: $string_cmds"
```

Salida:

```
Total commands available: 87
REGEXP command is available
String commands: STRING
```

13.6. Aplicaciones Prácticas

13.6.1. Acceso Seguro a Variables

Evitar Errores con INFO EXISTS

```
PROC safe_print WITH varname DO
  # Check if variable exists before accessing
  IF [INFO EXISTS $varname] THEN
    # Get the value using SET without argument
    SET value [SET $varname]
    PUTS "$varname == $value"
  ELSE
    PUTS "Variable '$varname' not found"
  END
END

SET username "Alice"

safe_print username
safe_print password
safe_print email
```

Salida:

```
username = Alice
Variable 'password' not found
Variable 'email' not found
```

13.6.2. Configuración Dinámica

Variables de Configuración Opcionales

```
PROC load_config DO
  # Set defaults
  SET config_host "localhost"
  SET config_port 8080
  SET config_debug 0

  # Check if user provided custom values
  IF [INFO EXISTS USER_HOST] THEN
    SET config_host $USER_HOST
    PUTS "Using custom host: $config_host"
  END

  IF [INFO EXISTS USER_PORT] THEN
    SET config_port $USER_PORT
    PUTS "Using custom port: $config_port"
  END

  IF [INFO EXISTS USER_DEBUG] THEN
    SET config_debug $USER_DEBUG
    PUTS "Debug mode: $config_debug"
  END

  RETURN [LIST $config_host $config_port $config_debug]
END

```

```
# Load with defaults
SET config [load_config]
PUTS "Config: $config"

# Now set custom values
SET USER_HOST "192.168.1.100"
SET USER_PORT 3000
SET config [load_config]
PUTS "Custom config: $config"
```

Salida:

```
Config: localhost 8080 0
Using custom host: 192.168.1.100
Using custom port: 3000
Custom config: 192.168.1.100 3000 0
```

13.6.3. Sistema de Ayuda

Ayuda Interactiva

```
PROC show_help WITH @command DO
  IF [INFO EXISTS command] THEN
    # Show help for specific command
    IF [LSEARCH [INFO PROCS] $command] != -1 THEN
      PUTS "Procedure: $command"
      SET params [INFO ARGS $command]
      PUTS "Parameters: $params"
      PUTS ""
      PUTS "Body:"
      PUTS [INFO BODY $command]
    ELSE
      PUTS "Command '$command' not found"
    END
  ELSE
    # Show all available procedures
    PUTS "Available procedures:"
    SET procs [INFO PROCS]
    FOREACH proc IN $procs DO
      SET params [INFO ARGS $proc]
      PUTS "  $proc ($params)"
    END
  END
END

PROC calculate_area WITH width height DO
  RETURN [EXPR $width * $height]
END

# Show all procedures
show_help

# Show help for specific procedure
```

```
PUTS ""
show_help calculate_area
```

Salida:

```
Available procedures:
  show_help (command)
  calculate_area (width height)

Procedure: calculate_area
Parameters: width height

Body:
  RETURN [EXPR $width * $height]
```

13.6.4. Herramienta de Depuración

Volcado de Variables

```
PROC dump_variables WITH @pattern DO
  # Default pattern: all variables
  IF [INFO EXISTS pattern] = 0 THEN
    SET pattern "*"
  END

  SET vars [INFO VARS $pattern]

  IF [LLENGTH $vars] = 0 THEN
    PUTS "No variables match pattern '$pattern'"
    RETURN
  END

  PUTS "=== Variables matching '$pattern' ==="
  FOREACH var IN $vars DO
    SET value [SET $var]
    SET type "string"

    # Try to determine type
    IF [REGEXP "^-?\[0-9\]+$" $value] THEN
      SET type "integer"
    ELSEIF [REGEXP "^-?\[0-9\]+\\.\[0-9\]+$" $value] THEN
      SET type "float"
    END

    PUTS [FORMAT "  %-15s = %-20s (%s)" $var $value $type]
  END
END

# Create some test variables
SET name "Alice"
SET age 30
SET height 1.68
```

```
SET count 42
SET debug_flag 1

# Dump all variables
dump_variables

# Dump only specific pattern
PUTS ""
dump_variables "debug_*"
```

Salida:

```
=== Variables matching '*' ===
  name            = Alice                 (string)
  age             = 30                    (integer)
  height          = 1.68                  (float)
  count           = 42                    (integer)
  debug_flag      = 1                     (integer)

=== Variables matching 'debug_*' ===
  debug_flag      = 1                     (integer)
```

13.6.5. Generador de Documentación de Procedimientos

Generar Documentación Automáticamente

```
PROC document_procedures DO
  SET procs [INFO PROCS]

  PUTS "==================================="
  PUTS "  BCL Procedure Documentation"
  PUTS "==================================="
  PUTS ""

  FOREACH proc IN $procs DO
    # Skip internal procedures
    IF [STRING MATCH "_*" $proc] THEN
      CONTINUE
    END

    SET params [INFO ARGS $proc]
    SET param_count [LLENGTH $params]

    PUTS "PROCEDURE: $proc"
    PUTS "Parameters: $param_count"

    IF $param_count > 0 THEN
      SET i 1
      FOREACH param IN $params DO
        PUTS "  $i. $param"
        INCR i
      END
    END
```

```
28
29      PUTS ""
30    END
31  END
32
33  # Define some example procedures
34  PROC add WITH a b DO
35    RETURN [EXPR $a + $b]
36  END
37
38  PROC greet WITH name @title DO
39    PUTS "Hello, $name"
40  END
41
42  PROC _internal_helper DO
43    # This won't be documented
44  END
45
46  # Generate documentation
47  document_procedures
```

Salida:

```
==================================
  BCL Procedure Documentation
==================================

PROCEDURE: document_procedures
Parameters: 0

PROCEDURE: add
Parameters: 2
  1. a
  2. b

PROCEDURE: greet
Parameters: 2
  1. name
  2. title
```

13.7. Referencia del Comando INFO

Cuadro 13.1: Referencia de Subcomandos INFO

Subcomando	Descripción
`EXISTS varname`	Verificar si existe variable
`VARS [pattern]`	Listar todas las variables (o que coincidan con patrón)
`GLOBALS [pattern]`	Listar variables globales
`LOCALS [pattern]`	Listar variables locales

Cuadro 13.1 – continuación

Subcomando	Descripción
`PROCS [pattern]`	Listar todos los procedimientos
`COMMANDS [pattern]`	Listar todos los comandos BCL
`ARGS procname`	Obtener parámetros del procedimiento
`BODY procname`	Obtener cuerpo del procedimiento (código)
`BCLVERSION`	Obtener cadena de versión de BCL

Consejo

¡La introspección es poderosa para construir herramientas de desarrollo, depuradores y código auto-documentado. Úsala para hacer tus programas más inteligentes y robustos!

Advertencia

Ten cuidado al usar **`INFO BODY`** con nombres de procedimiento proporcionados por el usuario, ya que expone tu código. En sistemas de producción, considera restringir el acceso a comandos de introspección.

Capítulo 14

Ejemplos

14.1. Procesador de Archivos de Texto

Contador de Palabras

```
PROC count_words WITH filename DO
  IF [FILE EXISTS $filename] = 0 THEN
    PUTS "Error: File not found"
    RETURN 0
  END

  SET fh [OPEN $filename R]
  SET content [READ $fh]
  CLOSE $fh

  # Split on whitespace
  SET words [SPLIT $content " "]
  SET count [LLENGTH $words]

  RETURN $count
END

SET wc [count_words "document.txt"]
PUTS "Word count: $wc"
```

14.2. Calculadora Simple

Calculadora Interactiva

```
PUTS "Simple Calculator"
PUTS "Enter expressions (e.g., 5 + 3)"
PUTS "Type 'quit' to exit"
PUTS ""

SET running 1
WHILE $running DO
  PUTSN "> "
  SET input [GETS stdin]
```

```
10
11    IF $input == "quit" THEN
12      SET running 0
13      CONTINUE
14    END
15
16    SET result [EXPR $input]
17    PUTS "= $result"
18  END
19
20  PUTS "Goodbye!"
```

14.3. Gestor de Lista de Tareas

Aplicación de Lista de Tareas

```
1   SET todos [LIST]
2
3   PROC add_todo WITH task DO
4     GLOBAL todos
5     SET todos [LAPPEND $todos $task]
6     PUTS "Added: $task"
7   END
8
9   PROC list_todos DO
10    GLOBAL todos
11    SET count [LLENGTH $todos]
12
13    IF $count == 0 THEN
14      PUTS "No todos"
15      RETURN
16    END
17
18    PUTS "Todo List:"
19    SET i 0
20    FOREACH todo IN $todos DO
21      INCR i
22      PUTS "  $i. $todo"
23    END
24  END
25
26  PROC remove_todo WITH index DO
27    GLOBAL todos
28    SET idx [EXPR $index - 1]
29    SET count [LLENGTH $todos]
30
31    IF $idx >= 0 AND $idx < $count THEN
32      SET todos [LREPLACE $todos $idx $idx]
33      PUTS "Removed todo #$index"
34    ELSE
35      PUTS "Invalid index"
36    END
```

```
END

# Main loop
SET running 1
WHILE $running DO
  PUTS ""
  PUTS "1. Add todo"
  PUTS "2. List todos"
  PUTS "3. Remove todo"
  PUTS "4. Quit"
  PUTSN "Choice: "

  SET choice [GETS stdin]

  SWITCH $choice DO
    CASE "1"
      PUTS "Enter task:"
      SET task [GETS stdin]
      add_todo $task
    CASE "2"
      list_todos
    CASE "3"
      PUTS "Enter todo number:"
      SET num [GETS stdin]
      remove_todo $num
    CASE "4"
      SET running 0
    DEFAULT
      PUTS "Invalid choice"
  END
END
```

14.4. Analizador de Archivos de Log

Analizar Logs

```
PROC analyze_log WITH logfile DO
  IF [FILE EXISTS $logfile] = 0 THEN
    PUTS "Log file not found"
    RETURN
  END

  SET fh [OPEN $logfile R]

  SET errors 0
  SET warnings 0
  SET info 0
  SET total 0

  WHILE [EOF $fh] = 0 DO
    SET line [GETS $fh]
    INCR total
```

```
17
18      IF [REGEXP "ERROR" $line NOCASE] THEN
19        INCR errors
20      ELSEIF [REGEXP "WARN" $line NOCASE] THEN
21        INCR warnings
22      ELSEIF [REGEXP "INFO" $line NOCASE] THEN
23        INCR info
24      END
25    END
26
27    CLOSE $fh
28
29    PUTS "Log Analysis Results:"
30    PUTS "  Total lines: $total"
31    PUTS "  Errors: $errors"
32    PUTS "  Warnings: $warnings"
33    PUTS "  Info: $info"
34  END
35
36  analyze_log "application.log"
```

Capítulo 15

Referencia Rápida

Este capítulo proporciona una referencia alfabética rápida de todos los comandos BCL.

Cuadro 15.1: Referencia de Comandos BCL

Comando	Descripción
`AFTER`	Pausar ejecución por milisegundos
`APPEND`	Agregar texto a variable
`ARGV`	Obtener argumentos de línea de comandos
`BREAK`	Salir del bucle actual
`CLOCK`	Operaciones de tiempo y fecha
`CLOSE`	Cerrar manejador de archivo
`CONCAT`	Concatenar listas
`CONTINUE`	Saltar a la siguiente iteración del bucle
`ENV`	Acceder a variables de entorno
`EOF`	Verificar fin de archivo
`EVAL`	Evaluar cadena como código
`EXEC`	Ejecutar comando del sistema
`EXIT`	Terminar programa
`EXPR`	Evaluar expresión
`FILE`	Operaciones de archivo (EXISTS, SIZE, DELETE, RENAME)
`FOR`	Bucle for
`FOREACH`	Iterar sobre lista
`FORMAT`	Formatear cadena (estilo printf)
`GETS`	Leer línea de entrada
`GLOB`	Coincidencia de patrones de archivo
`GLOBAL`	Declarar variable global
`IF`	Ejecución condicional
`INCR`	Incrementar variable
`INFO`	Comandos de introspección
`JOIN`	Unir lista a cadena
`LAPPEND`	Agregar a lista
`LINDEX`	Obtener elemento de lista
`LINSERT`	Insertar en lista
`LIST`	Crear lista
`LLENGTH`	Obtener longitud de lista

Cuadro 15.1 – continuación

Comando	Descripción
LRANGE	Extraer sublista
LREPLACE	Reemplazar elementos de lista
LSEARCH	Buscar en lista
LSORT	Ordenar lista
OPEN	Abrir archivo
PROC	Definir procedimiento
PUTS	Imprimir con nueva línea
PUTSN	Imprimir sin nueva línea
PWD	Obtener directorio actual
READ	Leer de archivo
REGEXP	Coincidencia de expresión regular
REGSUB	Sustitución de expresión regular
RETURN	Retornar de procedimiento
SCAN	Analizar cadena formateada
SEEK	Establecer posición de archivo
SET	Asignar/leer variable
SOURCE	Ejecutar archivo de script
SPLIT	Dividir cadena a lista
STRING	Operaciones de cadena
SWITCH	Rama de múltiples vías
TELL	Obtener posición de archivo
UNSET	Eliminar variable
WHILE	Bucle while

Capítulo 16

Arrays Asociativos

BCL soporta arrays asociativos (también conocidos como diccionarios o mapas hash en otros lenguajes), inspirados en la sintaxis de arrays de Tcl. Los arrays permiten almacenar valores indexados por claves arbitrarias, ya sean números o cadenas de texto.

16.1. Sintaxis Básica

Asignación:

```
SET nombreArray(indice) valor
```

Acceso:

```
$nombreArray(indice)
```

Array Simple

```
SET frutas(1) "manzana"
SET frutas(2) "naranja"
SET frutas(3) "platano"

PUTS $frutas(1)     # manzana
PUTS $frutas(2)     # naranja
PUTS $frutas(3)     # platano
```

Salida:

```
manzana
naranja
plátano
```

16.2. Índices Numéricos

Los arrays pueden usar números como índices, similar a los arrays tradicionales:

Índices Numéricos con Bucle

```
SET colores(1) "rojo"
SET colores(2) "verde"
```

```
SET colores(3) "azul"

SET i 1
WHILE $i <= 3 DO
  PUTS "colores($i) = $colores($i)"
  INCR i
END
```

Salida:

```
colores(1) = rojo
colores(2) = verde
colores(3) = azul
```

16.3. Índices de Texto (Asociativos)

El verdadero poder de los arrays de BCL viene de usar claves de texto:

Directorio Telefónico

```
SET telefono(Juan) "555-1234"
SET telefono(Maria) "555-5678"
SET telefono(Pedro) "555-9012"

SET contacto "Maria"
PUTS "Telefono de $contacto: $telefono($contacto)"
```

Salida:

```
Teléfono de María: 555-5678
```

16.4. Índices Variables

Puedes usar variables como índices de array:

Variable como Índice

```
SET datos(lunes) "10"
SET datos(martes) "15"
SET datos(miercoles) "12"

SET dia "martes"
PUTS "Datos del $dia: $datos($dia)"
```

Salida:

```
Datos del martes: 15
```

16.5. Índices con Expresiones

Los índices pueden ser expresiones, permitiendo acceso calculado al array:

Expresión como Índice

```
SET tabla(1) "A"
SET tabla(2) "B"
SET tabla(3) "C"

SET i 1
SET j [EXPR $i + 1]
PUTS $tabla($j)     # B

SET k [EXPR $i * 3]
PUTS $tabla($k)     # C
```

Salida:

```
B
C
```

16.6. Arrays Multidimensionales (Simulados)

BCL simula arrays multidimensionales usando índices compuestos:

Matriz 2D

```
SET matriz(1,1) "A"
SET matriz(1,2) "B"
SET matriz(2,1) "C"
SET matriz(2,2) "D"

PUTS "$matriz(1,1) $matriz(1,2)"
PUTS "$matriz(2,1) $matriz(2,2)"
```

Salida:

```
A B
C D
```

16.7. Verificar Existencia de Elementos

Usa `INFO EXISTS` para verificar si un elemento del array existe:

Verificar Existencia

```
SET config(debug) "true"

IF [INFO EXISTS config(debug)] THEN
  PUTS "Debug es: $config(debug)"
ELSE
  PUTS "Debug no establecido"
END

IF [INFO EXISTS config(timeout)] THEN
```

```
  PUTS "Timeout es: $config(timeout)"
ELSE
  PUTS "Timeout no configurado"
END
```

Salida:

```
Debug es: true
Timeout no configurado
```

16.8. Ejemplos Prácticos

16.8.1. Configuración de Aplicación

Configuración

```
SET config(host) "localhost"
SET config(port) "8080"
SET config(timeout) "30"
SET config(debug) "false"

PUTS "Servidor: $config(host):$config(port)"
PUTS "Timeout: $config(timeout)s"
PUTS "Modo debug: $config(debug)"
```

Salida:

```
Servidor: localhost:8080
Timeout: 30s
Modo debug: false
```

16.8.2. Contador de Eventos

Rastreo de Eventos

```
SET contador(login) "0"
SET contador(logout) "0"
SET contador(error) "0"

# Simular eventos
SET contador(login) [EXPR $contador(login) + 1]
SET contador(login) [EXPR $contador(login) + 1]
SET contador(logout) [EXPR $contador(logout) + 1]
SET contador(error) [EXPR $contador(error) + 1]
SET contador(login) [EXPR $contador(login) + 1]

PUTS "Login: $contador(login)"
PUTS "Logout: $contador(logout)"
PUTS "Error: $contador(error)"
```

Salida:

```
Login: 3
```

```
Logout: 1
Error: 1
```

16.8.3. Tabla de Multiplicar

Tabla de Multiplicar

```
SET num 5
SET i 1
WHILE $i <= 10 DO
  SET tabla($num,$i) [EXPR $num * $i]
  PUTS "$num x $i = $tabla($num,$i)"
  INCR i
END
```

Salida:

```
5 x 1 = 5
5 x 2 = 10
...
5 x 10 = 50
```

16.9. Arrays vs Listas

Consejo

Cuándo usar Arrays:

- Mapeos clave-valor (directorio telefónico, configuración)
- Índices arbitrarios (números no consecutivos o cadenas)
- Búsqueda rápida por clave

Cuándo usar Listas:

- Colecciones ordenadas
- Procesamiento secuencial
- Operaciones como ordenar, unir, dividir

16.10. Arrays en Procedimientos

Los arrays se pueden usar con la palabra clave **GLOBAL** en procedimientos:

Arrays Globales en Procedimientos

```
PROC mostrar_persona DO
  GLOBAL persona
  PUTS "Nombre: $persona(nombre)"
```

```
  PUTS "Edad: $persona(edad)"
END

SET persona(nombre) "Alicia"
SET persona(edad) "30"
mostrar_persona
```

Salida:

```
Nombre: Alicia
Edad: 30
```

Nota

Los elementos de array se almacenan como variables individuales con nombres como `nombrearray(indice)`. Se comportan como variables regulares y siguen las mismas reglas de alcance.

16.11. El Comando ARRAY

BCL proporciona el comando **ARRAY** para la manipulacion avanzada de arrays, inspirado en Tcl. Este comando ofrece operaciones poderosas para trabajar con arrays como estructuras completas.

16.11.1. ARRAY EXISTS

Verifica si un array existe (tiene al menos un elemento):

```
ARRAY EXISTS nombreArray
```

Devuelve `"1"` si el array existe, `"0"` en caso contrario.

Verificar Existencia de Array

```
SET resultado [ARRAY EXISTS config]     # "0"
SET config(debug) "true"
SET resultado [ARRAY EXISTS config]     # "1"
```

16.11.2. ARRAY SIZE

Obtiene el numero de elementos en un array:

```
ARRAY SIZE nombreArray
```

Tamano de Array

```
SET colores(rojo) "#FF0000"
SET colores(verde) "#00FF00"
SET colores(azul) "#0000FF"

SET tam [ARRAY SIZE colores]        # "3"
```

```
PUTS "El array tiene $tam elementos"
```

Salida:

```
El array tiene 3 elementos
```

16.11.3. ARRAY NAMES

Obtiene una lista de indices del array, opcionalmente filtrados por un patron:

```
ARRAY NAMES nombreArray ?patron?
```

Los patrones soportan comodines glob: `*` (cualquier caracter), `?` (un caracter), `[abc]` (conjunto de caracteres).

Listar Indices de Array

```
SET datos(nombre) "Juan"
SET datos(edad) "30"
SET datos(nota1) "8"
SET datos(nota2) "9"

SET todos [ARRAY NAMES datos]              # Todos los indices
SET notas [ARRAY NAMES datos "nota*"]      # Solo notas
SET con_a [ARRAY NAMES datos "*a*"]        # Que contengan 'a'

PUTS "Todos: $todos"
PUTS "Notas: $notas"
PUTS "Con 'a': $con_a"
```

Salida:

```
Todos: nombre edad nota1 nota2
Notas: nota1 nota2
Con 'a': nombre edad nota1 nota2
```

16.11.4. ARRAY GET

Obtiene el contenido del array como una lista de pares indice-valor:

```
ARRAY GET nombreArray ?patron?
```

Devuelve indices y valores alternados, opcionalmente filtrados por patron.

Obtener Contenido de Array

```
SET colores(rojo) "255,0,0"
SET colores(verde) "0,255,0"
SET colores(azul) "0,0,255"

SET todos [ARRAY GET colores]
PUTS "Todos los colores: $todos"
```

Salida:

```
Todos los colores: rojo 255,0,0 verde 0,255,0 azul 0,0,255
```

16.11.5. ARRAY SET

Puebla un array desde una lista de pares indice-valor:

```
ARRAY SET nombreArray lista
```

La lista debe tener un numero par de elementos.

Establecer Array desde Lista

```
# Crear array desde lista
ARRAY SET config "host localhost port 8080 debug true"

PUTS "Host: $config(host)"
PUTS "Puerto: $config(port)"
PUTS "Debug: $config(debug)"
```

Salida:

```
Host: localhost
Puerto: 8080
Debug: true
```

Copiar Arrays:

Copiar un Array

```
SET original(a) "1"
SET original(b) "2"
SET original(c) "3"

# Copiar array
SET datos [ARRAY GET original]
ARRAY SET copia $datos

PUTS "Tamano copia: [ARRAY SIZE copia]"
PUTS "copia(a) = $copia(a)"
```

Salida:

```
Tamano copia: 3
copia(a) = 1
```

16.11.6. ARRAY UNSET

Elimina elementos del array que coincidan con un patron:

```
ARRAY UNSET nombreArray ?patron?
```

Si no se especifica patron, elimina el array completo.

Eliminacion Selectiva

```
SET cache(temp_1) "data1"
SET cache(temp_2) "data2"
SET cache(perm_1) "data3"

PUTS "Antes: [ARRAY SIZE cache]"    # "3"
ARRAY UNSET cache "temp_*"
PUTS "Despues: [ARRAY SIZE cache]"  # "1"
```

Salida:

```
Antes: 3
Despues: 1
```

Eliminar Array Completo

```
SET miarray(1) "valor1"
SET miarray(2) "valor2"

ARRAY UNSET miarray

SET existe [ARRAY EXISTS miarray]    # "0"
PUTS "Array existe: $existe"
```

Salida:

```
Array existe: 0
```

16.11.7. Ejemplo Practico: Gestor de Configuracion

Gestor de Configuracion

```
# Cargar configuracion
ARRAY SET config "
    db_host localhost
    db_port 3306
    db_name myapp
    cache_enabled true
    cache_ttl 300
"

# Mostrar configuracion de base de datos
PUTS "Configuracion Base de Datos:"
SET db_keys [ARRAY NAMES config "db_*"]
SET i 0
WHILE $i < [LLENGTH $db_keys] DO
    SET key [LINDEX $db_keys $i]
    PUTS "  $key = $config($key)"
    INCR i
END

# Mostrar configuracion de cache
```

```
PUTS ""
PUTS "Configuracion Cache:"
SET cache_keys [ARRAY NAMES config "cache_*"]
SET i 0
WHILE $i < [LLENGTH $cache_keys] DO
    SET key [LINDEX $cache_keys $i]
    PUTS "  $key = $config($key)"
    INCR i
END
```

Salida:

```
Configuracion Base de Datos:
  db_host = localhost
  db_port = 3306
  db_name = myapp

Configuracion Cache:
  cache_enabled = true
  cache_ttl = 300
```

Capítulo 17

Datos Binarios

El comando **BINARY** proporciona facilidades para manipular datos binarios en BCL, permitiendo la conversion entre representaciones binarias y valores BCL. Esto es esencial para trabajar con formatos de archivo, protocolos de red y estructuras de datos de bajo nivel.

17.1. Vision General

Sintaxis:

```
BINARY subcomando argumentos
```

Dos subcomandos principales:

- **FORMAT** - Construir cadenas binarias desde valores BCL
- **SCAN** - Extraer valores BCL desde cadenas binarias

17.2. BINARY FORMAT

Construye una cadena binaria segun una especificacion de formato.
Sintaxis:

```
BINARY FORMAT formatString ?arg arg ...?
```

La cadena de formato contiene uno o mas especificadores de campo, cada uno consistente en un caracter de tipo y un conteo opcional.

17.2.1. Codigos de Formato

17.2.2. Ejemplos Basicos

Cadena con Padding

```
# Cadena con padding nulo (10 bytes)
SET data [BINARY FORMAT a10 "hola"]
PUTS "Longitud: [STRING LENGTH $data]"

# Cadena con padding espacios (10 bytes)
SET data [BINARY FORMAT A10 "mundo"]
PUTS "Longitud: [STRING LENGTH $data]"
```

Codigo	Descripcion
a	Cadena ASCII, padding con nulos
A	Cadena ASCII, padding con espacios
c	Enteros sin signo de 8 bits
s	Enteros de 16 bits, little-endian
S	Enteros de 16 bits, big-endian
i	Enteros de 32 bits, little-endian
I	Enteros de 32 bits, big-endian
H	Digitos hex, nibble alto primero
h	Digitos hex, nibble bajo primero
x	Insertar bytes nulos
X	Retroceder cursor
@	Posicion absoluta

Salida:

```
Longitud: 4
Longitud: 10
```

Enteros de 8 bits

```
# Empaquetar bytes: 65='A', 66='B', 67='C'
SET data [BINARY FORMAT c3 "65 66 67"]
PUTS "Data: $data"
PUTS "Longitud: [STRING LENGTH $data]"
```

Salida:

```
Data: ABC
Longitud: 3
```

Hexadecimal

```
# Convertir cadena hex a binario
SET data [BINARY FORMAT H8 "deadbeef"]
PUTS "Longitud: [STRING LENGTH $data]"

# Usar * para consumir toda la cadena
SET data [BINARY FORMAT H* "0123456789abcdef"]
PUTS "Longitud: [STRING LENGTH $data]"
```

Salida:

```
Longitud: 4
Longitud: 1
```

17.3. BINARY SCAN

Extrae campos de una cadena binaria y los almacena en variables.
Sintaxis:

```
BINARY SCAN string formatString ?varName varName ...?
```

Retorna el numero de conversiones exitosas.

17.3.1. Ejemplos de Scan

Extraer Cadenas

```
# Crear datos binarios
SET data [BINARY FORMAT a10 "hola"]

# Extraer 5 bytes
SET count [BINARY SCAN $data a4 var1]
PUTS "Extraido: '$var1'"
PUTS "Conversiones: $count"

# Extraer con recorte
SET data [BINARY FORMAT A10 "hola"]
SET count [BINARY SCAN $data A* var2]
PUTS "Recortado: '$var2'"
```

Salida:

```
Extraido: 'hola'
Conversiones: 1
Recortado: 'hola'
```

Extraer Enteros

```
# Empaquetar enteros
SET data [BINARY FORMAT c5 "10 20 30 40 50"]

# Desempaquetar enteros
SET count [BINARY SCAN $data c5 numeros]
PUTS "Numeros: $numeros"
PUTS "Conversiones: $count"
```

Salida:

```
Numeros: 10 20 30 40 50
Conversiones: 1
```

Extraer Hexadecimal

```
# Crear datos binarios desde hex
SET data [BINARY FORMAT H8 "cafebabe"]

# Extraer como cadena hex
SET count [BINARY SCAN $data H* hex]
PUTS "Hex: $hex"
```

Salida:

```
Hex: cafebabe
```

17.4. Ejemplos Practicos

17.4.1. Conversion Completa

Empaquetar y Desempaquetar

```
# Datos originales
SET orig_a "10"
SET orig_b "20"
SET orig_c "30"

# Empaquetar
SET packed [BINARY FORMAT c3 "$orig_a $orig_b $orig_c"]
PUTS "Longitud empaquetado: [STRING LENGTH $packed]"

# Desempaquetar
SET count [BINARY SCAN $packed c3 unpacked]
PUTS "Desempaquetado: $unpacked"
PUTS "Conversiones: $count"
```

Salida:

```
Longitud empaquetado: 3
Desempaquetado: 10 20 30
Conversiones: 1
```

17.4.2. Serializacion de Estructuras

Formato Simple de Registro

```
# Crear registro: nombre(10 bytes) + edad(8-bit) + id(8-bit)
SET nombre "Alicia"
SET edad "25"
SET id "42"

SET record [BINARY FORMAT a10c2 "Alicia" "$edad $id"]
PUTS "Tamano registro: [STRING LENGTH $record]"

# Leer registro
SET count [BINARY SCAN $record a10c2 nombre_guardado data]
SET edad_guardada [LINDEX $data 0]
SET id_guardado [LINDEX $data 1]

PUTS "Nombre: '$nombre_guardado'"
PUTS "Edad: $edad_guardada"
PUTS "ID: $id_guardado"
```

Salida:

```
Tamano registro: 12
Nombre: 'Alicia'
Edad: 25
ID: 42
```

17.4.3. Utilidad Hex Dump

Volcado Hexadecimal

```
PROC hexdump WITH data DO
    SET len [STRING LENGTH $data]
    SET i 0

    WHILE $i < $len DO
        # Obtener un byte
        SET byte [STRING INDEX $data $i]
        SET tmp [BINARY FORMAT a1 $byte]
        BINARY SCAN $tmp H2 hex

        PUTSN "$hex "
        INCR i

        # Nueva linea cada 16 bytes
        IF [EXPR $i % 16] == 0 THEN
            PUTS ""
        END
    END
    PUTS ""
END

SET data "Hola Mundo!"
hexdump $data
```

Salida:

```
48 6f 6c 61 20 4d 75 6e 64 6f 21
```

17.5. Endianness

BCL soporta ordenamiento de bytes little-endian y big-endian:

Consejo

Little-Endian (s, i):

- Byte menos significativo primero
- Comun en arquitecturas x86/x64
- Ejemplo: 0x1234 se almacena como [0x34, 0x12]

Big-Endian (S, I):

- Byte mas significativo primero
- Comun en protocolos de red (orden de bytes de red)
- Ejemplo: 0x1234 se almacena como [0x12, 0x34]

Para intercambio de datos multiplataforma, preferir big-endian (S, I).

17.6. Limitaciones

Nota

Importante: BCL usa cadenas C terminadas en nulo internamente. Esto significa:

- Los datos binarios que contienen bytes nulos (0x00) en el medio pueden truncarse
- Esto afecta principalmente a enteros de 16/32 bits con valores pequenos
- **Solucion:** Usar enteros de 8 bits (c) o hexadecimal (H) cuando sea posible

La implementacion actual NO soporta:

- Tipos float/double ('f', 'd')
- Representacion binaria de digitos ('b', 'B')

17.7. Patrones Comunes

17.7.1. Calculo de Checksum

Suma Simple de Bytes

```
PROC checksum WITH data DO
    SET sum 0
    SET i 0

    WHILE $i < [STRING LENGTH $data] DO
        SET byte [STRING INDEX $data $i]
        BINARY SCAN $byte c val
        SET sum [EXPR $sum + $val]
        INCR i
    END

    RETURN [EXPR $sum % 256]
END

SET msg "Hola"
SET cs [checksum $msg]
PUTS "Checksum de '$msg': $cs"
```

Salida:

```
Checksum de 'Hola': 172
```

17.7.2. Cabecera de Protocolo Simple

Protocolo de Mensaje

```
# Crear mensaje: tipo(8-bit) + longitud(8-bit) + datos
SET msg_tipo "1"
SET msg_datos "Hola"
SET msg_long [STRING LENGTH $msg_datos]
```

```
5
6  SET paquete [BINARY FORMAT c2a* "$msg_tipo $msg_long" $msg_datos]
7  PUTS "Tamano paquete: [STRING LENGTH $paquete]"
8
9  # Analizar mensaje
10 BINARY SCAN $paquete c2a* cabecera payload
11 SET tipo [LINDEX $cabecera 0]
12 SET long [LINDEX $cabecera 1]
13
14 PUTS "Tipo: $tipo"
15 PUTS "Longitud: $long"
16 PUTS "Payload: $payload"
```

Salida:

```
Tamano paquete: 6
Tipo: 1
Longitud: 4
Payload: Hola
```

17.8. Referencia Rapida

Tipo	Tamano	Proposito
a/A	variable	Cadenas ASCII
c	1 byte	Enteros 8-bit sin signo
s/S	2 bytes	Enteros 16-bit (little/big)
i/I	4 bytes	Enteros 32-bit (little/big)
H/h	variable	Digitos hexadecimales
x	variable	Padding/espaciado
X	-	Retroceder cursor
@	-	Posicionamiento absoluto

Modificadores de Conteo:

- Numero: conteo exacto
- * : consumir todo disponible
- Omitir: por defecto 1

Apéndice A

Instalación

A.1. Requisitos del Sistema

BCL puede ejecutarse en:

- Plataformas PC (Windows, Linux, macOS)
- Sistemas embebidos con recursos suficientes
- Microcontroladores (versión con recursos limitados)

A.2. Pasos de Instalación

A.2.1. Desde Binario

1. Descarga la distribución de BCL para tu plataforma
2. Extrae el archivo
3. Agrega el directorio `bin` a tu PATH
4. Verifica la instalación: `bcl -version`

A.2.2. Desde Código Fuente

Consulta el archivo BUILD.md en la distribución de código fuente.

Apéndice B

Solución de Problemas

B.1. Errores Comunes

Variable no encontrada Verifica la ortografía y asegúrate de que la variable se haya establecido con `SET`

Archivo no encontrado Verifica la ruta del archivo y los permisos

Error de sintaxis Verifica que no falten palabras clave `END` y la sintaxis correcta del comando

Incompatibilidad de tipo Asegúrate de que las operaciones numéricas usen números válidos

Apéndice C

Diferencias con Tcl

BCL está inspirado en Tcl pero tiene varias diferencias clave:

- **Sintaxis**: Usa palabras clave al estilo BASIC (`IF...THEN...END` en lugar de llaves)
- **Insensibilidad a mayúsculas/minúsculas**: Los comandos no distinguen entre mayúsculas y minúsculas
- **Expansión de variables**: Solo forma `$var` (no `${var}`)
- **Invocación de procedimientos**: Invocación directa por nombre (no `CALL`)
- **Fin de bloque**: Todos los bloques terminan con `END`
- **Parámetros opcionales**: Usa prefijo `@param`

Apéndice D

Mejoras Futuras

Características planificadas para futuras versiones de BCL:

- Tipo de dato diccionario/matriz asociativa
- Soporte de espacios de nombres
- Características de programación orientada a objetos
- Biblioteca estándar más completa
- Herramientas de depuración y perfilado
- Sistema de gestión de paquetes