Universidad
Las Palmas de Gran Canaria

Grado en
Ingeniería Informática

Manual de Referencia del lenguaje de programación c3P

Asignatura:

Procesadores de Lenguajes

Autores:

David Parreño Barbuzano

Hector Miguel Martín Álvarez

Índice

1. Introducción

c3P es un lenguaje de programación imperativo, fuertemente tipado, declarativo, e inspirado en otros lenguajes como C, Go, HolyC, y Java. En este manual de referencia, se explica la utilización de **c3P**, en el que se incluye sus elementos léxicos y sintácticos.

2. Elementos léxicos

2.1. Identificadores

Un **identificador** es una secuencia de caracteres con las que se pueden nombrar componentes del lenguaje como variables, funciones y procedimientos. Los caracteres que se utilizan en un **identificador** pueden ser letras minúsculas, dígitos, o el carácter de subrayado '_'. Como en muchos otros lenguajes, el primer carácter de un identificador no puede ser un dígito.

2.2. Palabras clave

Las **palabras clave** son un tipo de identificador particular que se emplean en el lenguaje como secuencia de caracteres reservadas. Esto quiere decir que, cuando se escribe en **c3P** una palabra clave, sólo puede mantener un único propósito y significado.

A continuación, se muestra una lista de las palabras clave en **c3P**:

b, break, c, call, continue, else, endfor, endfunc, endif, endproc, endwhile, f32, f64, for, func, i8, i16, i32, i64, if, proc, ret, while

2.3. Constantes

Una **constante** es un valor numérico o de caracteres que se utilizan como valores literales para definir el valor de las variables. En **c3P**, no se permite asignar variables a valores de un tipo de dato diferente, puesto que en esta asignación no se está realizando una transformación del dato conocida como **casting**.

2.3.1. Constantes enteras

Las constantes enteras se codifican en **c3P** a partir de una secuencia de dígitos. Como este tipo de constantes representan a los números enteros, el rango de valores posible comprende a todos los enteros positivos y negativos.

2.3.2. Constantes de carácter

Las constantes de carácter comprenden a todos los caracteres imprimibles, excepto los caracteres de escape típicos en la mayoría de lenguajes de programación. Cabe mencionar que existe un tipo de datos especifico con el que definir una cadena de caracteres, que se representa delimitado en , mientras que definir un sólo carácter es ”

2.3.3. Constantes de números reales

Las constantes de números reales son aquellas que representan a un número decimal o en coma flotante. La codificación de este tipo de constate consta de una secuencia de dígitos que conforman la parte entera del mundo, seguido de un punto decimal y una secuencia de dígitos que corresponden a la parte fraccionaria. Es importante mencionar que en **c3P** no se puede omitir en la declaración de la constante la parte fraccionaria, y tampoco se puede declarar número cuya parte decimal sea nula.

2.3.4. Constantes de valores booleanos

Las constantes de valores booleanos son aquellas que pueden tener uno de sus dos estados posibles (verdadero o falso), y cumplen las reglas de la aritmética de Bool. En **c3P**, los booleanos no pueden transformarse a datos enteros, como ocurren en lenguajes tales como C.

2.4. Operadores

Un **operador** es una palabra especial que representa en el lenguaje una operación, que puede ser aritmética o lógica, y que afecta a uno o dos operandos.

2.5. Separadores

Un **separador** es un carácter que se utiliza para separar las palabras especiales declaradas en un programa. En **c3P**, hay separadores que se emplean en la indexación, en la definición de un contenedor, en la declaración de una variable, entre otros. En la siguiente tabla se muestra cada separador y su uso en el lenguaje.

Nombre	**Funcionalidad**
( )	- Llamada y declaración de funciones/procedimientos - Declaración de condiciones lógicas en If, For, While, etc. - Bloque de agrupamiento de expresiones aritméticas
[ ]	- Indexación de Contenedores y cadenas de caracteres - Declaración de Contenedores de Elementos
{ }	- Declaración de los Elementos de un Contenedor
.	- Separador de los números decimales
:	- Definición y declaración de variables
,	- Separar parámetros, valores de un contenedor
SPACE	- Carácter de espacio, de tabulación y nueva línea

Cuadro 1: Separadores disponibles

2.6. Comentarios

Los comentarios en c3P se definen mediante «?»

```
? Esto es un comentario
x : i32 = 10
```

3. Tipos de Datos

3.1. Tipos de Datos Primitivos

3.1.1. Tipos Enteros

En **c3P**, los tipos enteros se diferencian no sólo por su identificador (nombre del tipo de entero), sino también por el rango de valores que pueden almacenar, que se miden por bits y pueden alcanzar los 64 bits.

Identificador	NºBits	Rango
i8	8	-128 a 127
i16	16	-32768 a 32767
i32	32	-214483648 a 2147483647
i64	64	-9223372036854775808 a 9223372036854775807

Cuadro 2: Tipos Enteros

3.1.2. Tipos de Números Reales

Al igual que los números enteros, los tipos de números reales se diferencian tanto por su identificador, sino también por su rango de valores que pueden almacenar, que pueden alcanzar hasta los 64 bits. Cabe destacar que, dependiendo del ordenador que esté ejecutando el programa, los rangos de valores varían.

Identificador	NºBits	Rango
f32	32	F32_MIN a F32_MAX
f64	64	F64_MIN a F64_MAX

Cuadro 3: Tipos Enteros

3.1.3. Tipos de Carácter

Los tipos de caracteres permiten representar uno o varios caracteres de dígitos, letras, u otros símbolos, a excepción de los caracteres especiales.

- ✔ **c**: puede contener un valor que cumple '([a-zA-Z0-9_& %...]+)*'

3.1.4. Tipos Lógicos

Los tipos de datos lógicos se codifican mediante el identificador «b», y pueden tomar dos valores posibles (verdadero codificado en «T», o falso representado en «F»).

3.2. Contenedores

Un **contenedor** es una estructura de datos en la que se almacena valores de un mismo tipo. Estos elementos están organizados internamente a partir de los índices del contenedor, que marcan cada una de los posiciones en las que se almacena su contenido. En **c3P**, los contenedores son de tamaño fijo y permiten operaciones de indexación.

3.2.1. Declaración de contenedores

Un contenedor se declara especificando el tipo de datos que tendrá cada elementos, el nombre de la variable, y el número de elementos que puede almacenar, tal y como se puede observar en el siguiente ejemplo:

```
mi_array : i32[10]
```

Respecto al tamaño de un contenedor, puede declararse tanto con una constante literal entera positiva, como con una variable del mismo tipo. Cabe mencionar que se permiten tamaños de hasta la unidad.

3.2.2. Inicialización de contenedores

Los elementos de un contenedor se pueden inicializar de forma individual por medio de la indexación, o también a partir de la declaración entre llaves de valores literales, donde cada elemento tendrá asociado la posición en la que se define. En este ejemplo se puede observar la segunda inicialización explicada:

```
mi_array : i32[5] = { 0, 1, 2, 3, 4 }
```

Es importante considerar que en la inicialización se tienen que especificar todos los valores del contenedor de elementos, lo que hace esta funcionalidad poco útil en los casos en los que el tamaño de elementos sea muy grandes.

3.2.3. Acceso a los elementos de un Contenedor

En **c3P**, los elementos de un contenedor se pueden acceder por medio de su índice y de la indexación. Para ello, se especifica el nombre del contenedor, seguido del índice del elemento al que se quiere acceder, encerrado entre corchetes. A continuación se muestra un ejemplo de esta funcionalidad:

```
mi_array[0] = 5
```

3.3. Tamaño de Contenedores y Cadena de caracteres

En **c3P** para obtener el tamaño de un contenedor existe la función «arrlen».

```
mi_array : i32[3] = {0, 1, 2}
x : i32 = call arrlen mi_array
```

4. Expresiones y Operadores

4.1. Expresiones

En **c3P**, una **expresión** es un conjunto de uno o más operandos y operadores, mientras que un **operando** son objetos descritos como constantes, variables, y llamadas a funciones que devuelven valores de un tipo determinado. A continuación, se explica cada uno de los operadores:

4.2. Operador de Asignación

Los operadores de asignación se utilizan para almacenan valores en variables, y se codifica en **c3P** mediante el símbolo «=». Como ya se mencionó anteriormente, si el valor derecho que se asigna a la variable presenta un tipo de dato diferente, en **c3P** se hará una operación de **casting**, siempre que la transformación sea posible. En el siguiente ejemplo, se muestra el operador de asignación, y una situación de **casting**, tanto una válida como otra que provoca un error de ejecución.

```
x1 : i32 = 20
y1 : i64 = 10.3

? Casting válido porque ambos datos son numéricos
? y puede transformarse porque puede guardarse la
? parte entera del valor tipo i64 (x1 = 10)
x1 = y1

x2 : i32 = 20
y2 : i64 = 10.3

? Casting válido porque ambos datos son numéricos
? y puede transformarse porque puede guardarse la
? parte entera como decimal (y2 = 20.0)
y2 = x2

carac : c = 'H'
num : i32 = 10

? ¡¡ERROR!! El casting sólo se permite en
? datos numéricos o si ambos valores son caracteres
num = carac
```

4.3. Operadores Aritméticos

c3P proporciona operadores aritméticos como el de suma, resta, multiplicación, y división. A continuación, se muestra ejemplos de estos operadores:

```
x : i32 = 5 + 3
y : f32 = 10.23 + 37.332
z : f32:  = x + y
```

```
x : i32 = 5 - 3
y : f32 = 57.223 - 10.903
z : f64 = x - y

x : i32 = 5 * 3
y : f32 = 47.4 * 1.001
z : f32 = x * y

x : f32 = 5 / 3
y : f64 = 940.0 / 20.2
z : f64 = x / y

x : f32 = 5 % 3
y : f64 = 940.0 % 20.2
z : f64 = x % y

x : f32 = 5 ^ 3
y : f64 = 940.0 ^ 20.2
z : f64 = x ^ y
```

4.4. Operadores de Comparación

Los operadores de comparación se emplean para comprobar si se satisface una condición en la que participan dos operadores que se relacionan. El resultado de este operador siempre es booleano. A continuación, se muestra cada uno de los operadores de comparación que existen en **c3P**.

El operador de igualdad «==» comprueba la igualdad de dos operandos:

```
if (x == y)
  call show ("x es igual a y")
else
  call show ("x no es igual a y")
endif
```

El operador de desigualdad «!=» comprueba la desigualdad de sus dos operandos.

```
if (x != y)
  call show ("x no es igual a y")
else
  call show ("x es igual a y")
endif
```

Además de la igualdad y la desigualdad, existen una serie de operadores con los que comprobar si un valor es menor que, mayor que, menor o igual que, o mayor o igual que otro operando.

```
if (x < y)
  call show ("x es menor que y")
endif
```

```
if (x <= y)
  call show ("x es menor o igual que y")
endif

if (x > y)
  call show ("x es mayor que y")
endif

if (x >= y)
  call show ("x es mayor o igual que y")
endif
```

4.5. Operadores Lógicos

Los operadores lógicos se utilizan para comprobar la veracidad de dos condiciones expresadas mediante valores booleanos. A continuación, se explica cada uno de los proporcionados en **c3P**:

- ✔ El operador **and** comprueba si dos expresiones son ambas verdaderas.

```
if ((x == 5) and (y == 10))
  call show ("x es 5 e y es 10")
endif
```

- ✔ El operador **or** comprueba si al menos una de las dos expresiones es verdadera.

```
if ((x == 5) or (y == 10))
   call showln ("x es 5 o y es 10")
endif
```

- ✔ Puede anteponer a una expresión lógica un operador de negación **not** para invertir el valor de la expresión booleana:

```
if (not (x == 5))
  call showln ("x no es 5")
endif
```

4.6. Llamadas a Funciones como Expresiones

Para facilitar la definición de una expresión, en **c3P** se considera a cualquier función una función, por lo que se pueden asignar la llamada de una de estas a una variable.

```
func y : i32(x : i32)
    ret x * 2
endfunc

a : i32 = 10 + (call function 20);
```

4.7. Precedencia de Operadores

Cuando una expresión contiene varios operadores, como a + b * f(), los operadores se agrupan según las reglas de precedencia.

A continuación se presenta una lista de tipos de expresiones, presentadas primero en orden de mayor precedencia. A veces, dos o más operadores tienen la misma precedencia; todos esos operadores se aplican de izquierda a derecha a menos que se indique lo contrario.

1. Llamadas a funciones.
2. Operadores unarios, incluyendo la negación lógica..
3. Expresiones de multiplicación, división, división modular y exponentes.
4. Expresiones de suma y resta.
5. Expresiones de mayor que, menor que, mayor o igual que, y menor o igual que.
6. Expresiones de igualdad y desigualdad.
7. Expresiones lógicas AND.
8. Expresiones lógicas OR.
9. Todas las expresiones de asignación, incluida la asignación compuesta.
   Cuando aparecen varias sentencias de asignación como subexpresiones en una única expresión mayor, se evalúan de derecha a izquierda.
10. Expresiones con operadores de coma.

4.8. Orden de Evaluación

En c3P todo programa comienza en un procedimiento llamado obligatoriamente «main».

No se puede asumir que las subexpresiones múltiples se evalúan en el orden que parece natural.

5. Sentencia

5.1. Expresiones de Sentencia

En **c3P** cada expresión, sentencia o declaración que se codifique se escribe en una sola línea, por lo que no se pueden representar más de una declaración en una línea. Esto no afecta a las condiciones y expresiones aritmético/lógicas, pero sí a las estructuras de control, declaración de funciones y procedimientos, variables, etc. A continuación, se muestra ejemplos válidos e inválidos de declaraciones.

```
? Declaración de expresiones correcta
x : i32 = 10
y : i32 = 20
z : i32 = x + y
z = z + x * 2 / 10
call show "z = "
call showln z

? Declaración de expresiones incorrecta
```

```
if (x == y) if (y != 10)

endif endif

? Incorrecto
x : i32 = 10, y = 10
```

5.2. Sentencia if

En **c3P** se puede utilizar la sentencia «if» para controlar el flujo del programa mediante una operación condicional.

```
if (test)
  codigo1
else
  codigo2
endif
```

Si «test» se evalúa como verdadera, entonces se ejecutará «codigo1» mientras que «codigo2» no lo hará. Si el resultado de la prueba es falso, tendrá lugar el efecto contrario «codigo2» se ejecutará mientras «codigo1» no lo hará. La cláusula «else» es opcional.

Es posible emplear varias sentencias «if» para comprobar múltiples condiciones, como se muestra en el ejemplo:

```
if (x == 1)
  call show  ("x es 1")
else if (x == 2)
  call show  ("x es 2")
else if (x == 3)
  call show  ("x es 3")
else
  call show  ("x es otra cosa")
endif
```

5.3. Sentencia while

La declaración «while» es un bucle en el que en cada iteración se comprueba una condición, que si se satisface, se repite la ejecución del bucle hasta que no se cumpla el criterio de parada. A continuación se muestra un ejemplo:

```
counter : i32 = 0

while (counter < 10)
  call show counter
  counter = counter + 1
endwhile
```

5.4. Sentencia for

El bucle con la declaración «for» es similar al «while», pero con la característica de que inicializa una variable, que en cada iteración varía, hasta que se cumpla una condición. La codificación del bucle se hace primero inicializando la variable, luego estableciendo la condición de parada, y después indicando la expresión que se va a ejecutar para cambiar la variable tras la iteración. Cada parte del «for» está separado por comas, encerrado en paréntesis, y el cuerpo del bucle termina en «endfor».

```
for (x : i32 = 0, x = x + 1, x < 10)
  call show x
endfor
```

5.5. Bloques

5.5.1. Sentencia break

La declaración break se utiliza para terminar la ejecución de un bucle. Cabe destacar que esta sentencia afecta al bucle más interno en el que esté dentro. A continuación se muestra un ejemplO:

```
for (x : i32 = 1, x = x + 1, x <= 10)
    if (x == 8)
      break
    else
      call show x
    endif
endfor
```

5.5.2. Sentencia continue

La declaración continue permite terminar la iteración actual del bucle y comenzar la siguiente sin salir de él. Hay que tener en cuenta que esta declaración sólo afecta al bucle más interno en el que esté dentro.

```
suma : i32 = 0

for (x : i32 = 0, x = x + 1, x < 100)
    if (x % 2 == 0)
      continue
    else
      suma = suma + x
    endif
endfor
```

Si pones una sentencia «continue» dentro de un bucle que a su vez está dentro de un bucle, entonces sólo afecta al bucle más interno.

5.5.3. Sentencia ret

La sentencia ret se utiliza para indicar el valor que va a devolver una función al terminar su ejecución. En **c3P**, siempre se tiene que devolver en una función un valor, pero jamás se puede usar en un procedimiento.

```
func cumsum : i32(x : i32[])
    value : i32 = 0
    len : i32 = call arrlen x

    for (i : i32 = 0, i = i + 1, i < len)
        value = value + x[i]
    endfor

    ret value
endfunc
```

6. Funciones y Procedimiento

6.1. Declaración de Funciones y Procedimiento

Por un lado, una función se declara en **c3P** mediante la palabra clave func, seguido de su nombre, del tipo de dato que devuelve, y encerrado entre paréntesis, de una lista de parámetros separados por comas. Además de todas estas componentes, toda función en **c3P** tiene que terminar con la palabra clave ret, que es en la que se indica el valor que devuelve.

```
func nombre : retorno (parámetros)
    ...
endfunc
```

Por otro lado, un procedimiento se declara en **c3P** mediante la palabra clave proc, seguido de su nombre, y encerrado entre paréntesis, de una lista de parámetros separados por comas. A diferencia de las funciones, los procedimientos nunca incluyen la palabra clave ret, porque no devuelven ningún valor. Además, ret no se puede usar para terminar la ejecución de un procedimiento.

```
proc nombre (parámetros)
    ...
endproc
```

El nombre puede ser cualquier identificador válido.

Los parámetros consta de cero o más parámetros, separados por comas. Un parámetro consiste en un nombre y un tipo de datos para el parámetro. Estos parámetros son opcionales

6.2. Definición de Funciones y Procedimiento

La definición de una función o un procedimiento se escribe para especificar lo que realmente hace una función o un procedimiento. Una definición de función consiste en información sobre el nombre

de la función, el tipo de retorno únicamente en las funciones y los tipos y nombres de los parámetros, junto con el cuerpo de la función.

```
func add_values : i64 (x : i32, y : i32)
  ret x + y
endfunc
```

```
proc showSum (x : i32, y : i32)
  call show x + y
endproc
```

6.3. Llamadas a Funciones y Procedimiento

Para llamar a una función o procedimiento en **c3P**, se utiliza la palabra clave, seguido del nombre de la función, y si es necesario, de los parámetros necesarios, separados por comas.

```
x : i64 = call add_values 5, 3
call fibonacci 10
```

6.4. Parámetros de Funciones y Procedimiento

En las funciones y procedimientos, los parámetros pueden ser cualquier expresión, como un valor literal, un valor almacenado en una variable, o una expresión más compleja construida mediante la combinación de estos.

En el cuerpo de la función o procedimiento, el parámetro se pasa por valor, lo que significa que no se puede cambiar el valor pasado cambiando la copia local.

```
x : i32 = 23
call y1 x
call y2 x

func y1 : i32 (a : i32)
  a = 2 * a
  ret a
endfunc

proc y2 (a : i32)
  call show a
endproc
```

6.5. Función Principal

Cualquier ejecución de un programa en **c3P** tiene que empezar siempre en un procedimiento llamado «main» que no tiene parámetros.

```
proc main ()
  call show "Hello world"
endproc
```

6.6. Funciones Recursivas

En **c3P** es posible definir funciones o procedimientos recursivos, es decir, que en su ejecución se llamen a sí mismos, con en el siguiente ejemplo.

```
func fibonacci(n : i32)
    if (n > 1)
        f1 : i32 = call fibonacci n - 1
        f2 : i32 = call fibonacci n - 2
        ret f1 + f2
    endif

    ret n
endfunc
```

7. Estructura del programa

Un programa en **c3P** puede existir en un sólo fichero que tiene que tener obligatoriamente el método main. Cada fichero escrito en **c3P** tiene que tener la extensión .c3P o .c3p

7.1. Alcance de los datos

Una variable declarada puede ser accesible sólo dentro de una función o procedimiento, o dentro de un fichero. Las declaraciones realizadas fuera de funciones o procedimientos son accesibles para todo el archivo. Respecto a las variables locales, éstas son visibles dentro del cuerpo de una función pero no fuera de ésta. Además de todo esto, hay que considerar que una declaración en **c3P** no es visible por las declaraciones que la preceden.

8. Programas de Ejemplo

8.1. hello_world.c3p

```
proc main()
    call showln "Hello, world!"
endproc
```

8.2. mean_age.c3p

```
proc main()
    i32 : total_ages
    i16 : age_mean

    total_ages = call input "Inserta el número de edades ", "%i"

    for (i : i32 = 0, i = i + 1, i < total_ages)
        i32 : age = call input "Inserta la edad "
```

```
        age_mean = age_mean + age
    endfor

    age_mean = age_mean / total_ages
    call show "La edad media es:
    call showln age_mean
endproc
```

8.3. fibonacci.c3p

```
func fibonacci(n : i32)
    if (n > 1)
        f1 : i32 = call fibonacci n - 1
        f2 : i32 = call fibonacci n - 2
        ret f1 + f2
    endif

    ret n
endfunc

proc main()
    n10 : i32 = call fibonacci 10
    n20 : i32 = call fibonacci 20
    n30 : i32 = call fibonacci 30
    n40 : i32 = call fibonacci 40
    call show "fib(10) = "
    call showln n10
    call show "fib(20) = "
    call showln n20
    call show "fib(30) = "
    call showln n30
    call show "fib(40) = "
    call showln n40
endproc
```