Programación

Asignatura 240205 Programación. Curso 2020–2021

Lectura[6]: **Acciones y funciones**

Dr. José Ramón González de Mendívil
mendivil@unavarra.es
Departamento de Estadística, Informática y Matemáticas
Edificio Las Encinas
Universidad Pública de Navarra

Resumen

El objeto de esta Lectura es mostrar las posibilidades que disponemos para codificar algoritmos de manera que puedan ser reutilizados. La parametrización de algoritmos y su encapsulación mediante acciones y funciones permite resolver problemas complejos mediante el uso apropiado del diseño descendente. Explicaremos en esta Lectura cómo codificar acciones y funciones en el Lenguaje C.

Índice

1. Acciones y funciones

A medida que avanzamos en la construcción de algoritmos y de los programas que codifican dichos algoritmos, se hace patente la necesidad de encontrar formas de reducir la complejidad de ciertos problemas. Una técnica es construir la solución como resultado de refinamientos sucesivos. En cada paso de refinamiento, se divide el problema a resolver en un número de subproblemas de menor complejidad. Mediante este proceso es posible llegar a problemas lo suficientemente sencillos para ser resueltos directamente. Entonces, se combinan las soluciones de los subproblemas para resolver el problema que los comprende y así, se van obteniendo soluciones hasta llegar al nivel de mayor complejidad.

Los algoritmos que se introducen en cada nivel de refinamiento pueden aparecer codificados en forma de pequeños subprogramas denominados **acciones** y **funciones**. Las acciones y funciones deben estar declaradas antes de utilizarse en el cuerpo principal del algoritmo o en otras acciones o funciones.

1.1. Acciones

Una **acción** se puede entender como un algoritmo que está diseñado de tal forma que puede ser utilizado por cualquier otro algoritmo u otra acción. Por ejemplo, supongamos que tenemos el algoritmo de intercambio de dos variables.

algoritmo intercambio;
a, b : *entero*;
$\{P : a = A \wedge b = B\}$
var aux: *entero* **fvar**
    aux:= a;
    a:= b;
    b:= aux
$\{Q : a = B \wedge b = A\}$
falgoritmo

Si observas detenidamente el algoritmo *ordena3b* (Ejercicio 2, Sesión 1 de Prácticas), el intercambio se ha empleado tres veces con diferentes variables, y cada vez, se ha tenido que modificar para ajustarse a las variables que se usaban en cada alternativa

intercambiar p por s
intercambiar s por t
intercambiar p por t

Incluso en este caso sencillo, podemos utilizar el concepto de acción. Primero observamos la especificación del algoritmo *intercambio*: la variable a y la variable b se utilizan tanto para contener un dato inicial (son de entrada) y para recoger el resultado final (son de salida). Por lo tanto van a ser considerados como parámetros de entrada/salida (**ent/sal**). Una vez realizada esta identificación procedemos a presentar la declaración de la acción. La declaración incluye el nombre de la acción, los parámetros formales (indicando si son de entrada **ent**, de salida **sal**, o de entrada/salida **ent/sal**) y su tipo. Como en todo algoritmo incluimos la precondición y la postcondición.

acción intercambio(**ent/sal** a: *entero*, **ent/sal** b: *entero*);
$\{P : a = A \wedge b = B\}$
$\{Q : a = B \wedge b = A\}$
facción

La declaración de la acción es suficiente para saber qué hace y cómo otros algoritmos la pueden utilizar. Este hecho es independiente de cómo se resuelva el problema *intercambio* especificado por los predicados P y Q. Se puede utilizar cualquier solución que satisfaga la especificación de la acción. Para ver el efecto en el código, escribimos de nuevo el algoritmo *ordena3b* pero usando la acción *intercambio*:

```
algoritmo ordena3b;
    acción intercambio(ent/sal a: entero, ent/sal b: entero);
    {P : a = A ∧ b = B}
    var aux: entero fvar
        aux:= a;
        a:= b;
        b:= aux
    {Q : a = B ∧ b = A}
    facción
//
x, y, z: entero
p, s, t: entero
{Pre : x = X ∧ y = Y ∧ z = Z}
    p:= x; s:= y; t:= z;
    si   p > s ⟶ intercambio(p,s)
      [] p ≤ s ⟶ continuar
    fsi;
    si   p > t ⟶ intercambio(p,t)
      [] p ≤ t ⟶ continuar
    fsi;
    si   s > t ⟶ intercambio(s,t)
      [] s ≤ t ⟶ continuar
    fsi
{Post : (p, s, t) ∈ Perm(X, Y, Z) ∧ p ≤ s ≤ t}
falgoritmo
```

Cuando en el algoritmo principal se invoca la ejecución de, por ejemplo,

```
intercambio(p,s)
```

se produce:

1. Una asociación de los argumentos actuales con los parámetros formales en la invocación de acción. En este caso, las variables p y s se asocian con los parámetros formales a y b respectivamente. Esto es debido al orden en que aparecen los parámetros formales. Por otra parte, para que no haya errores, el número de argumentos tiene que coincidir siempre con el número de parámetros formales, y el tipo de cada argumento que se asocia con un parámetro debe ser exáctamente el mismo.

2. La comunicación entre argumentos y parámetros al inicio de la invocación de la acción. El valor de cada argumento es asignado a su correspondiente parámetro en el momento de la llamada a la acción. En el ejemplo, el valor de p se pasa a a y el de s se pasa a b, ya que a y b son parámetros de entrada (aunque también de salida). A partir de ahí, se ejecutan las instrucciones que forman parte del cuerpo de la acción.

3. La comunicación entre los parámetros y los argumentos al final de la ejecución de las instrucciones de la acción. El valor final de cada parámetro de salida se pasa al argumento correspondiente. En el ejemplo, el valor de a se pasa a p y el de b se pasa a s.

Nomenclatura.

Parámetro formal o simplemente 'parámetro': Corresponde a la declaración 'nombre: tipo' que se emplea en la declaración de la acción.
Argumento real o simplemente 'argumento': nombre de la variable o expresión del mismo tipo que el parámetro, y que sustituye a este parámetro en la línea de código donde se emplea la acción.
Declaración de la acción: nombre de la acción seguida de la lista de parámetros con su tipo y etiquetados con su carácteristica de entrada, salida o entrada/salida.
Cuerpo de la acción: declaración de variables y sentencias que se emplean en la declaración de la acción para establecer el código que debe ejecutar la acción para cumplir con su especificación.
Alcance de las variables: las variables declaradas en el programa principal pueden emplearse en cualquier lugar posterior del texto del programa. Es conveniente que las acciones (y las funciones) no utilicen estas variables en su cuerpo para evitar efectos laterales no deseados. Por ese motivo, las acciones (y funciones) se declaran antes que las variables del programa principal. Las variables declaradas en el cuerpo de la acción son 'locales' a la acción, y su alcance está limitado al texto de la acción. Si el nombre de una variable local coincide con el nombre de una variable principal en una sentencia, se toma siempre en consideración la declaración de variable más cercana a la sentencia. Finalmente, en algunos casos, por conveniencia del diseñador, los nombres de los parámetros pueden coincidir con los nombres de los argumentos. Aunque depende de cada caso, esta conincidencia no resulta problemática cuando los parámentros son sólo de entrada, o sólo de salida. Cuando son parámetros de entrada/salida hay que tener un poco de cuidado con las coincidencias de nombres. Ante la duda respecto al comportamiento en ejecución, es conveniente utilizar nombres diferentes entre parámetros y argumentos. Los programadores deben ser conscientes de que cada lenguaje de programación puede emplear convenios diferentes respecto a la nomenclatura y alcande de variables.

Discusión. Un lenguaje de programación debe determinar de forma precisa la manera en que los cambios efectuados sobre los valores de los parámetros en el cuerpo de la acción invocada afectan a los argumentos de la acción principal:

- **Parámetro de entrada**. Si un parámetro formal de una acción se declara de entrada, el argumento que se asocia al parámetro en la invocación no verá alterado su valor al terminar la ejecución de la acción.
- **Parámetro de salida**. Si el parámetro se declara de salida, el argumento asociado al parámetro se verá afectado porque recoge el valor final del parámetro al terminar la ejecución de la acción.
- **Parámetro de entrada/salida**. Si el parámetro es de entrada/salida, el argumento se verá afectado por recoger el resultado final pero, además, su valor inicial será utilizado como parte del cálculo en la acción.

De lo anterior se deduce que un argumento para un parámetro de salida o de entrada/salida sólo puede ser una variable del mismo tipo que el parámetro. En cambio, el argumento para un parámetro de entrada puede ser una expresión del mismo tipo que el parámetro. Algunos lenguajes de programación consideran un error la posibilidad de que en el código de la acción se modifique el valor de un parámetro de entrada. Otros lenguajes como por ejemplo C, si al ejecutar el código de la acción se produce un cambio mediante alguna instrucción de asignación en el parámetro, el argumento siempre mantendrá su valor original. Se considera conveniente no modificar el valor de los parámetros (sólo) de entrada, aunque ello no tenga ningún efecto sobre

los argumentos asociados, fuera de la acción. El lenguaje algorítmico que utilizamos está pensado para facilitar la escritura de algoritmos y estudiar su corrección, no para ser ejecutado por un ordenador. Por este motivo, en ocasiones tenemos que limitar algunos usos que otros lenguajes permiten. Como **norma** de esta asignatura, en la escritura del código de las acciones y funciones con el lenguaje algorítmico **no permitimos** la modificación de los parámentros de entrada. Para ello, basta con evitar instrucciones de asignación que afecten a estos parámetros de entrada.

El algoritmo *ordena3b*, que acabamos de ver, también puede escribirse en forma de acción. En este caso las variables x, y y z actuarán como parámetros de entrada: se usan sus valores iniciales en los cálculos y no se ven alteradas por la ejecución de las instrucciones del algoritmo. Las variables p, s, y t actuarán como parámetros de salida, ya que contendrán el resultado según la especificación. Teniendo en cuenta la discusión anterior, la declaración de esta acción es

acción ordena3b(**ent** x, y, z : *entero*; **sal** p, s, t : *entero*);
$\{P' : x = X \wedge y = Y \wedge z = Z\}$
$\{Q' : x = X \wedge y = Y \wedge z = Z \wedge (p, s, t) \in \mathrm{Perm}(X, Y, Z) \wedge p \leq s \leq t\}$
facción

Observamos que se ha repetido $x = X \wedge y = Y \wedge z = Z$ en la precondición P' y en la postcondición Q' para que quede claro que actúan sólo como parámetros de entrada. En este curso, no haremos uso de reglas de inferecia para acciones que permiten demostrar la corrección de los algoritmos que las usan, pero sí haremos demostraciones de corrección de algoritmos que usan funciones por lo que la siguiente sección es importante a ese respecto.

1.2. Funciones

Una función es una acción que tiene uno o varios parámetros de entrada y un único parámetro de salida[1]. La funciones tienen una forma muy relacionada con las funciones que estamos habituados a utilizar en matemáticas y son un mecanismo adecuado y rápido para incrementar el repertorio de instrucciones de un lenguaje de programación. Como ejemplo, damos la declaración de una función para calcular el máximo común divisor.

función max_com_div(x, y : *entero*) **dev** mcd: *entero*;
$\{P : x = X \wedge y = Y \wedge X > 0 \wedge Y > 0\}$
$\{Q : x = X \wedge y = Y \wedge mcd = MCD(X, Y)\}$
    dev mcd
ffunción

Por la propia forma en la que está expresada la declaración de la función, se asume que x e y son parámetros de entrada y que el resultado se devuelve en el parámetro de salida mcd. Como en la declaración queda clararamente reflejada la clasificación de los parámetros, no escribimos **ent**, ni **sal**. En el lenguaje algorítmico, las funciones **sólo se utilizan** en combinación con instrucciones de asignación. Por ejemplo,

```
r:= max_com_div(25, 10*z);
```

Como **norma** de la asignatura, en el lenguaje algorítmico **no está permitido** combinar funciones con otras operaciones o funciones en la misma asignación con objeto de facilitar el estudio de la corrección de los algoritmos que las usan. Por ejemplo, la siguiente instrucción de asignación no está permitida

```
r:= max_com_div(25, 10*z) + val_abs(r);
```

En la siguiente subsección explicamos, mediante un ejemplo, otras cuestiones importantes relacionadas con las funciones y su verificación.

[1] En el tema de funciones recursivas permitiremos que las funciones tengan más de un parámentro de salida.

1.2.1. Regla de inferencia de funciones

Consideremos el siguiente ejercicio: Derive un algoritmo para el cálculo de una serie exponencial siguiendo especificación

n, s: *entero*
$\{Pre : n = N \ \wedge \ n \geq 0\}$
    serie_exponencial
$\{Post : s = \sum_{0<i\leq n} i^i \ \wedge \ n = N\}$

En el diseño del algoritmo, vamos a ver que es adecuado resolver también la siguiente función

función potencia (a, b: *entero*) **dev** c: *entero*
$\{P : a = A \ \wedge \ a > 0 \ \wedge \ b = B \ \wedge \ b \geq 0\}$
    ...
$\{Q : c = a^b \ \wedge a = A \ \wedge \ b = B\}$
    dev c
ffunción

(1) Observamos que en la especificación de la función reforzamos la idea de que los parámetros (formales) a y b son parámetros de entrada poniendo tanto en la precondición P como en la postcondición Q las expresiones $a = A$ y $b = B$. Por tanto, cuando escribamos el código de la función no usaremos instrucciones de asignación del tipo $a := ...$ ó $b := ....$ En el caso de que el algoritmo empleado (ver cualquiera de la soluciones dadas para el cálculo de la potencia de dos números) para el cuerpo de la función necesite modificar el valor de alguna de esas variables, se pueden copiar antes en variables auxiliares para realizar las instrucciones de asignación correspondientes. De esta forma, si se hace al principio, por ejemplo, $aux\_a := a$, se tiene libertad para modificar aux_a respetando $a = A$ a lo largo de todo el código del cuerpo de la función.

(2) A la hora de escribir la especificación de la función, los parámetros formales de entrada se declaran como variables, en el caso anterior 'a, b: *entero*'. No obstante, dadas las características de los parámetros de entrada, se pueden emplear argumentos que sean expresiones del mismo tipo que los parámetros. Por ejemplo, si se va a utilizar la función potencia() en un algoritmo que tiene una declaración de variables $p, j, k, s : entero$ y, en un determinado punto del algoritmo, se necesita calcular en la variable p el valor de la potencia $(j + 2)^{(3k)}$, basta con se escriba la asignación: $p := potencia(j+2, 3*k)$. El parámetro formal a recoge el valor de la expresión $(j+2)$ y el parámetro b el valor de la expresión $3 * k$. Ambas expresiones constituyen los argumentos para la función y son evaluadas antes de la llamada a la función. El parámetro formal c de la función *potencia* sirve para que el valor que toma después de ejecutarse la función sea copiado en la variable p sobre la que se realiza la asignación. De esta manera, como en la postcondición de la función $c = a^b$ y tanto a como b no cambian su valor inicial, sabemos que cuando la función termina, la asignación dará como resultado $p = (j + 2)^{(3k)}$.

(3) Siguiendo el ejemplo anterior, supongamos que tenemos el siguiente predicado antes de la llamada a la función en la descripción del algoritmo

$\{j = J \wedge k = K \wedge 0 \leq k < j \wedge p = j^k \wedge s = j \cdot k\}$
p:= potencia(j+2, 3*k)

Después de la ejecución de la función, podemos deducir cuál será el predicado resultante

$\{j = J \wedge k = K \wedge 0 \leq k < j \wedge p = j^k \wedge s = j \cdot k\}$
p:= potencia(j+2, 3*k)
$\{j = J \wedge k = K \wedge 0 \leq k < j \wedge p = (j + 2)^{(3k)} \wedge s = j \cdot k\}$

Esta deducción sólo es posible porque estamos asumiendo que: (i) las variables que forman parte de las expresiones, que son argumentos de entrada, nunca cambiarán su valor por efecto de la función; y (ii) la función no tiene efectos colaterales sobre otras variables del algoritmo u acción principal que la invoca. En el ejemplo, la variable s no se ha utilizado (ní se puede utilizar) en el cuerpo de la función, modificando su valor de forma oculta, aunque el lenguaje de programación que se utilice para la codificación lo hubiera permitido.

En Programación no permitimos efectos colaterales en las funciones ni en las acciones, por eso, ponemos la declaración de las funciones y acciones antes de las variables que se utilizan como argumentos en dichas funciones y acciones. En general, los compiladores de los lenguajes de programación, que empleamos para la codificación, avisan al programador de que hay una 'referencia hacía atrás': una variable se ha utilizado antes de ser declarada.

(4) Teniendo en cuenta lo discutido anteriormente y con objeto de facilitar las demostraciones y comprobaciones con los algoritmos que usan funciones escribiremos la especificación de las funciones de la forma que se muestra en el ejemplo.

función potencia (a, b: *entero*) **dev** c: *entero*
$\{P : a > 0 \ \wedge \ b \geq 0\}$
$\{Q : c = a^b\}$
    dev c
ffunción

Asumiremos y garantizaremos a partir de ahora que:

- Los parámetros de entrada no serán modificados en el código del cuerpo de la función.
- En el código de la función no se utilizarán variables declaradas[2] fuera del ámbito de la función con objeto de evitar posibles efectos colaterales.
- Las variables que forman parte de las expresiones, que se usen como argumentos asociados a los parámetros de entrada, nunca cambiarán su valor por efecto de la ejecución de la función.

(5) Sobre la verificación de los algoritmos que usan funciones: **regla de inferencia de funciones**. La declaración de la función contiene toda la información que necesitamos, si se ha descrito correctamente la precondición y la postcondición de la misma. El algoritmo que utiliza dicha función puede requerir, para ser correcto, que parte de su código cumpla las condiciones dadas por otros predicados. Analicemos el siguiente ejemplo.

algoritmo ejemplo;
    funcion potencia (a, b: *entero*) **dev** c: *entero*
    $\{P : a > 0 \ \wedge \ b \geq 0\}$
    $\{Q : c = a^b\}$
    dev c
    ffuncion
p, j, k, s: *entero*
$\{Pre\}$
    .....
    $\{R\}$
    p:= potencia(Exp1, Exp2)
                (* Exp1, Exp2, expresiones que forman los argumentos *)

[2]Se recomienda que los nombres de las variables declaradas localmente en el cuerpo de la función sean diferentes a los nombres de las variables declaradas fuera del ámbito de la función.

$\{S\}$
.....
$\{Post\}$
falgoritmo

Debes observar que los predicados R y S expresan relaciones sobre las variables del algoritmo. Así que, comenzando en un estado que cumpla R, después de ejecutar la instrucción de asignación que llama a la función, se llegue a un estado que cumpla S se tiene que verificar:

(i) $[[R \Rightarrow P(a \leftarrow Exp1, b \leftarrow Exp2)]]$, es decir, se tienen que cumplir las condiciones de la precondición de la función sobre los argumentos de entrada. La sustitución múltiple anterior se hace de manera simultánea.

(ii) Se tiene que cumplir que la implementación de la función es correcta: Cumple con su especificación y termina!

(iii) Sea R' la parte de R que no incluye condiciones sobre la variable p entonces se tiene que cumplir $[[R' \wedge Q(c \leftarrow p, a \leftarrow Exp1, b \leftarrow Exp2) \Rightarrow S]]$, 'lo que no cambia en R por efecto de la función más los resultados obtenidos por la función implican S', es decir, $\{R' \wedge Q(c \leftarrow p, a \leftarrow Exp1, b \leftarrow Exp2)\} \subseteq \{S\}$.

1.2.2. Ejemplo: Derivación del algoritmo para la serie exponencial

Recuperamos la especificación del problema del cálculo de una serie exponencial

n, s: *entero*
$\{Pre : n = N \ \wedge \ n \geq 0\}$
serie_exponencial
$\{Post : s = \sum_{0<i\leq n} i^i \ \wedge \ n = N\}$

Se trata de derivar un algoritmo para este problema utilizando la función *potencia* previamente especificada.

• *Propuesta de invariante.* El método a emplear para obtener el invariante del algoritmo principal es el de sustitución de la variable n (que actúa como una constante ya que su valor no cambia en el algoritmo) por una nueva variable j (de tipo entero). Así el invariante sería

$$Inv \equiv s = \sum_{0<i\leq j} i^i \ \wedge \ 0 \leq j \leq n \ \wedge \ n = N.$$

• *Cálculo de las instrucciones de inicio.* Puedes observar que el primer valor para j es 0, ya que j sustituye a n y 0 es un (primer) valor posible para dicha variable. Entonces, el dominio del sumatorio es vacío y por lo tanto habrá que iniciar s a 0.

• *Cálculo de la condición de continuación.* La condición de continuación es $B : j \neq n$, ya que $[[Inv \wedge \neg B \Rightarrow Post]]$.

• *Cálculo de las instrucciones de avanzar.* Si j comienza el bucle con el valor 0 y el bucle debe terminar cuando $j = n$, siendo n en general un número entero positivo, la decisión para avanzar es $j := j + 1$.

• *Cálculo de las instrucciones de restablecer.* En el cálculo del bucle, al avanzar con la instrucción $j := j+1$, será necesario restablecer el valor de la variable s con el valor de la expresión $(j+1)^{(j+1)}$ mediante $s := s + (j+1)^{(j+1)}$. Como esta expresión no se puede calcular directamente, podemos reforzar el invariante introduciendo una nueva variable entera z que sea igual a dicha expresión. De manera que, una nueva propuesta de invariante puede ser

$$Inv' \equiv s = \sum_{0<i\leq j} i^i \ \wedge \ 0 \leq j \leq n \ \wedge \ z = (j+1)^{(j+1)} \ \wedge \ n = N.$$

En la revisión del programa con el nuevo invariante Inv', al avanzar con $j := j+1$, deberemos resolver otro problema que es el cálculo de $z = (j+2)^{(j+2)}$. El problema que encontramos aquí, es que no podemos restablecer el valor de z con alguna expresión sencilla formada por su valor previo y el valor de j.

La conclusión es que el reforzamiento que hemos introducido es en valde, ya que debemos calcular explícitamente con un algoritmo $(j+1)^{(j+1)}$ en el caso de que utilicemos como invariante Inv, el primero que hemos propuesto, o bien, $(j+2)^{(j+2)}$ si seguimos con el invariante Inv' que tiene dicho reforzamiento. Para no tener que escribir el código del algoritmo que calcula cualquiera de esas potencias dentro del código del algoritmo principal, podemos emplear la función *potencia* que ya tenemos especificada. También es importante darse cuenta de que en este caso no es obligatorio reforzar el invariante Inv, aunque sí vamos a necesitar una variable auxiliar para obtener el valor de $(j+1)^{(j+1)}$ con objeto de restablecer el valor de la variable s.

Los estudiantes deben completar el algoritmo que está esbozado a continuación y realizar las comprobaciones que se indican utilizando la regla de inferencia de la función. También deben dar el cuerpo de la función potencia. En el algoritmo 'val_ini_j' y 'val_ini_s' son las expresiones de los valores iniciales para las variables j y s respectivamente. En la función potencia, 'exp_a' y 'exp_b' son las expresiones para los argumentos de entrada de la función (en la discusión anterior están las soluciones).

algoritmo serie_exponencial;
  función potencia (a, b: entero) **dev** c: entero
  $\{P : a > 0 \ \wedge \ b \geq 0\}$
    ...
  $\{Q : c = a^b\}$
    dev c
  ffunción
n, s: *entero*
$\{Pre \equiv n = N \ \wedge \ N \geq 0\}$
var j, aux: *entero* **fvar**
  j:= 'val_ini_j';
  s:= 'val_ini_s';
$\{Inv\}$ $\{cota = n - j\}$
  mientras j $\neq$ n **hacer**
    aux:= potencia('exp_a', 'exp_b');
    s:= s + aux;
    j:= j + 1
  fmientras
$\{Post \equiv s = \sum_{0<i\leq n} i^i \ \wedge \ n = N\}$
falgoritmo

Se debe comporbar:

- $[[Pre \Rightarrow ((Inv)^s_{val\_ini\_s})^j_{val\_ini\_j}]]$
- $[[Inv \wedge \neg B \Rightarrow Post]]$
- $[[Inv \wedge B \Rightarrow P(a \leftarrow exp\_a, b \leftarrow exp\_b)]]$
- $[[Inv \wedge B \wedge Q(c \leftarrow aux, a \leftarrow exp\_a, b \leftarrow exp\_b) \Rightarrow ((Inv)^j_{j+1})^s_{s+aux}]]$
- $[[Inv \wedge B \Rightarrow cota > 0]]$
- $[[Inv \wedge B \wedge cota = T \Rightarrow (cota < T)^j_{j+1}]]$

El algoritmo de la serie exponencial puede transformarse fácilmente en una función para poder emplearse en otro algoritmo. Complete el ejercicio construyendo la función *serie_exponencial* a partir del algoritmo construido. Identifique en la especificación las variables que actuarán como parámetros de entrada y salida, y complete la declaración

función serie_exponencial(.....) **dev** ....: .....
$\{P' :$ .....$\}$
$\{Q' :$ .....$\}$
    dev .....
ffunción

2. Codificación de acciones y funciones en C

Hasta el momento hemos analizado las acciones y funciones desde el punto de vista del lenguaje algorítmico, en esta sección vamos a estudiar la definición y el uso de funciones en C. En el lenguaje C no hay palabras reservadas para indicar la declaración de acciones o funciones como en otros lenguajes como Python o Pascal. Por otra parte, es común decir que C sólo admite la declaración de funciones. El aspecto de la declaración de una función en C es bastante simple

```
tipo identificador(parametros)
{
    cuerpo de la funcion
}
```

El `identificador` es el nombre de la función y debe ser un identificador válido como los que se emplean para dar nombres a las variables. El tipo de la función indica de qué tipo de datos es el valor que devuelve la función. Entre los paréntesis aparece la declaración de los parámetros separados por comas. El cuerpo de la función debe ir encerrado entre llaves. Aquí encontraremos la declaración de variables locales a la función y las instrucciones a ejecutar. Veamos un ejemplo:

funcion sumatorio(a, b : *entero*) **dev** s: *entero*;
$\{P : 0 \leq a \leq b\}$
var i: entero **fvar**
    i:= a;
    s:= a;
    $\{Inv : s = \sum k : a \leq k \leq i : k \wedge a \leq i \leq b\}$
    mientras $i \neq b$ **hacer**
        s:= s + (i+1);
        i:= i + 1
    fmientras;
    dev s
$\{Q : s = \sum_{k=a}^{b} k\}$
ffuncion

La codificación en C de la función anterior es la siguiente:

```
int sumatorio(int a, int b)
{
    int i, s;
    i = a;
    s = a;
    while (i != b)
    {
        s = s + (i + 1);
        i = i + 1;
    }
    return s;
}
```

La última sentencia `return s;` es la sentencia que devuelve el valor contenido en la expresión (en este caso la variable `s`). La ejecución de esta sentencia finaliza la ejecución de la función. Las variables que se declaran en el cuerpo de la función se crean cuando se invoca la ejecución de la función desde otra función (recuerde que la primera función que se ejecuta es la función `main(void)`) y desaparecen cuando termina la ejecución de la función. Una función que no tiene parámetros se declara de la misma forma pero utilizando el término `void`. Es posible que una función no devuelva nada y por tanto, el tipo de retorno de la función también es `void`.

En el lenguaje algorítmico los parámetros se clasifican como de entrada, salida, entrada/salida. En la función en C se hace implicita esta clasificación según cómo sea la declaración de parámetros.

En el ejemplo anterior los parámetros `a` y `b` son de entrada y C se **asegura** de que si modificas su valor en el cuerpo de la función, el argumento que usa la función no se ve alterado después de terminar la ejecución de la función. Aunque nosotros no permitimos modificaciones en el cuerpo de la función de los parámetros de entrada en el lenguaje algorítmico para asegurar la corrección, el lenguaje C sí te lo permite.

La cuestión es, ¿cómo se indica en C que los parámetros son de salida o de entrada/salida?. Básicamente hay que indicar que el parámetro es una dirección de memoria que apunta a un determinado tipo de datos. Recordemos que si declaramos una variable `int x`, `&x` nos devuelve su dirección. Esta dirección, `&x`, apunta a un número del tipo `int`. C dispone de una operación que te permite acceder al contenido de una dirección, esta operación es el asterisco `*`[3]. Así pues, el valor contenido en la dirección `&x`, es `*&x`, y por lo tanto, `*&x == x`. Cuando escribimos una declaración de la forma `int * p`, estamos indicando que el parámetro `p` contiene una dirección que apunta a un tipo `int`. Por ese motivo se suele decir que `p` es un *apuntador*.

En el siguiente ejemplo codificamos en C la acción intercambio para que analice lo expuesto anteriormente.

accion intercambio(**ent/sal** a: *entero*, **ent/sal** b: *entero*);
$\{P : a = A \wedge b = B\}$
var aux: entero **fvar**
    aux:= a;
    a:= b;
    b:= aux
$\{Q : a = B \wedge b = A\}$
faccion

[3]No se debe confundir el uso de `*` como operador de acceso al contenido de una dirección con la operación de multiplicación ya que ambas usan el mismo símbolo.

Un programa en C que codifica la acción anterior se muestra a continuación.

```
#include <stdio.h>
//
void intercambio(int *a, int *b)
{
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
    return;
}
//
int main(void)
{
    int x;
    int y;
    y = 21;
    x = 12;
    intercambio(&x, &y);
    printf("x es%d e y es%d \n", x, y);
    return 0;
}
```

Debemos observar que la variable `a` de tipo entero de la acción, al ser de entrada/salida la codificamos como `int *a` para reflejar en la función de C ese hecho. Ahora `a` es un apuntador. En la codificación de las instrucciones del cuerpo de la función en C debemos escribir `*a` para acceder al contenido apuntado por `a`. Lo mismo podemos decir para la variable `b` de la acción.

Como hemos visto, las cosas son un poco más sencillas en el caso de codificar funciones del lenguaje algorítmico a C. Veamos otro ejemplo sencillo de codificación de una función. Supongamos que se quiere codificar en C la siguiente función

función val_abs(x: *entero*) **dev** y: *entero*;
{P : verdadero}
    si x $\geq 0 \longrightarrow$ y:= x
      [] x $< 0 \longrightarrow$ y:= -x
    fsi;
{$Q : y = |x|$}
    dev y
ffunción

Observamos que x es un parámetro de entrada y por tanto, no realizamos instrucciones de asignación sobre él. La codificación en C de esta función es la siguiente,

```
#include <stdio.h>
//
int val_abs(int x)
{
    int y;
    if (x >= 0)
        y = x;
    else
        y = -x;
    return y;
}
```

```
//
int main(void)
{
    ....
    return 0;
}
```

Hemos declarado la variable local `y` dentro de la función que hace las veces del parámetro de salida. Además la instrucción (**dev** y) en el lenguaje algorítmico simplemente se codifica como la sentencia `return y;` en C. Todo lo que hemos explicado en este capítulo es válido para cualquier tipo de datos simple. En las sesiones de prácticas dedicadas a la estructura de datos del tipo tabla volveremos sobre la forma de definición de parámetros y paso de argumentos con variables de este tipo de datos.

3. Ejercicios

1. Sea N una constante positiva. Diseñe un algoritmo para la siguiente especificación:

   h, g: **tabla** [1..N] **de** *entero*;
   s: *entero*;
   $\{Pre : h = H \wedge g = G\}$
   cuenta_divisores
   $\{Post : h = H \wedge g = G \wedge s = \sum k : 1..N : (\#i : 1..N : g[i] \bmod h[k] = 0)\}$

2. Sean N y M dos constantes positivas. Diseñe un algoritmo iterativo que cuente el número de veces que cada elemento de la tabla t aparece repetido en la tabla h, de acuerdo con la especificación siguiente,

   t, b: **tabla** [1..N] **de** *entero*;
   h: **tabla** [1..M] **de** *entero*;
   $\{Pre : t = T \ \wedge \ h = H\}$
   cuenta_elementos
   $\{Post : t = T \ \wedge \ h = H \ \wedge \ (\forall k : 1..N : b[k] = (\#l : 1..M : h[l] = t[k]))\}$

Referencias

[1] J. Castro y otros. Curso de Programación. McGraw-Hill. 1993