Sesión 3

Especificación y uso de módulos en C++(II)

3.1 La clase `Poli` para representar polinomios

Disponemos de la clase `Poli` que permite operar con polinomios de coeficientes enteros. Se basa en el módulo *Polinomio* que aparece a continuación, al que se han añadido operaciones de lectura y escritura. Atención: las cabeceras de algunas operaciones cambian significativamente.

En esta sesión os proponemos resolver una serie de ejercicios empleando dicha clase. En los primeros ya os damos el código y sólo tendréis que traducirlo a C++. Antes de resolverlos, podéis ver varios ejemplos de uso en los ficheros de `/assig/prap/cplus/sesion3`.

Módulo Polinomio
{*Especificación*}

Tipo poli
{Representa los polinomios con coeficientes enteros, denotados de la forma
$p = c_0 + c_1x + c_2x^2 + ...$}

Operaciones

función crear_poli (*c*: entero, *n*: natural) **retorna** *p*: poli
{ cierto }
$\{p = cx^n\}$

función sumar_poli (*p*1: poli, *p*2: poli) **retorna** *q*: poli
$\{p1 = a_0 + a_1x + a_2x^2 + ...\ ,\ p2 = b_0 + b_1x + b_2x^2 + ...\}$
$\{q = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + ...\}$

acción modif_coef (**e/s** *p*: poli, **ent** *a*: entero, **ent** *n*: natural)
$\{p = c_0 + c_1x + ... + c_nx^n + ...\}$
$\{p = c_0 + c_1x + ... + ax^n + ...\}$

función grado (p: poli) **retorna** m: natural
{ cierto }
{m es el menor natural tal que los coeficientes c_n de p son 0
para todo n mayor que m}

función coef (p: poli, n: natural) **retorna** c: entero
$\{p = c_0 + c_1 x + \ldots + c_n x^n + \ldots\}$
$\{c = c_n\}$

3.2 Evaluación de polinomios

Emplearemos la estrategia más simple: recorrer los términos del polinomio y sumar la aportación de cada uno. Introducimos la variable auxiliar *pot* para no calcular la potencia cada vez. En los asertos escribimos n en lugar de grado(p) y c_α en lugar de coef(p,α), para simplificar[1].

función eval_poli (p: poli, a: entero) **retorna** *eval*: entero
$\{p = \Sigma\alpha : 0 \leq \alpha \leq n : c_\alpha * x^\alpha\}$
var k: natural; *pot*: entero;
$k := 0$; *pot* $:= 1$;
eval $:=$ coef$(p,0)$;
$\{$ Inv : *eval* $= \Sigma\alpha : 0 \leq \alpha \leq k : c_\alpha * a^\alpha \ \wedge \ n \geq k \geq 0 \ \wedge \ pot = a^k\}$
$\{$ Cota : $n-k\}$
mientras $k <$ grado(p) **hacer**
pot $:=$ *pot* $* a$;
eval $:=$ *eval* $+$ coef$(p,k+1) *$ *pot*;
$k := k+1$;
fmientras ;
retorna *eval*
$\{$*eval* $= \Sigma\alpha : 0 \leq \alpha \leq n : c_\alpha * a^\alpha\}$

Optimizaciones:

- Guardar el grado antes de entrar en el bucle para no calcularlo cada vez
- Actualizar la k antes que el *eval* para no sumar 1 dos veces

Con el algoritmo de Horner nos ahorramos la variable auxiliar *pot*, aunque el invariante es algo más complicado. Notad que los términos se recorren en sentido contrario.

[1] Nótese también que, con los asertos empleados, si evaluamos el polinomio en el valor $a = 0$ queda un término 0^0 que identificamos con 1, aunque ello supone un cierto abuso de la notación.

función eval_poli_Horner (p: poli, a: entero) **retorna** *eval*: entero
$\{p = \Sigma\alpha : 0 \leq \alpha \leq n : c_\alpha * x^\alpha\}$
  var k: nat;
  $k := \text{grado}(p)$;
  $eval := \text{coef}(p,k)$;
  $\{\text{ Inv} : eval = \Sigma\alpha : k \leq \alpha \leq n : c_\alpha * a^{\alpha-k} \ \wedge \ n \geq k \geq 0\}$
  $\{\text{ Cota} : k\}$
  mientras $k > 0$ **hacer**
    $eval := eval * a + \text{coef}(p,k-1)$;
    $k := k-1$;
  fmientras ;
retorna *eval*
$\{eval = \Sigma\alpha : 0 \leq \alpha \leq n : c_\alpha * a^\alpha\}$

Optimizaciones: Restar $k-1$ sólo una vez.

3.3 Derivación de polinomios

Algoritmo clásico: recorrer los términos del polinomio y sumar la aportación de cada uno.

función deriv_poli (p: poli) **retorna** dp: poli
$\{p = \Sigma\alpha : 0 \leq \alpha \leq n : c_\alpha * x^\alpha\}$
  var k: nat;
  $k := 0$;
  $dp := \text{crear\_poli}(0,0)$;
  $\{\text{ Inv} : dp = \Sigma\alpha : 1 \leq \alpha \leq k : \alpha * c_\alpha * x^{\alpha-1} \ \wedge \ n \geq k \geq 0\}$
  $\{\text{ Cota} : n-k\}$
  mientras $k < \text{grado}(p)$ **hacer**
    $\text{modif\_coef}(dp, \text{coef}(p,k+1) * (k+1), k)$;
    $k := k+1$;
  fmientras ;
retorna dp
$\{dp = \Sigma\alpha : 1 \leq \alpha \leq n : \alpha * c_\alpha * x^{\alpha-1}\}$

Optimizaciones: Las mismas.

Nota: Una opción equivalente para la postcondición es

$$dp = \Sigma\alpha : 0 \leq \alpha \leq n-1 : \alpha+1 * c_{\alpha+1} * x^\alpha$$

3.4 División de polinomios

En esta sección y en la siguiente tendréis que diseñar una parte de los algoritmos antes de traducirlos a C++. Comencemos por la división de polinomios, que se especifica de manera similar a la de números.

función div_poli (p_1, p_2: poli) **retorna** c, r: poli
$\{\text{coef}(p_2, \text{grado}(p_2)) = 1\}$
$\{p_1 = p_2 * c + r \text{ y } \text{grado}(r) < \text{grado}(p_2)\}$

La precondición se refiere a que no todo par de polinomios de coeficientes enteros produce un cociente y un resto de coeficientes enteros. Si bien podría establecerse una precondición un poco más débil, para nuestros propósitos es suficiente que el coeficiente de mayor grado del divisor sea igual a 1.

Podéis diseñar fácilmente la división sistematizando iterativa o recursivamente el método clásico de restas sucesivas, de forma que se vaya calculando simultáneamente el cociente y el resto, como en el siguiente ejemplo

$$
\begin{array}{rll}
r \rightarrow & 3x^2+2x+5 & \big| \; x+3 \\
\cline{3-3}
aux \rightarrow & \underline{-3x^2-9x} & 3x \quad \leftarrow c_1 \\
r \rightarrow & -7x+5 & \big| \; x+3 \\
\cline{3-3}
aux \rightarrow & \underline{7x+21} & -7 \quad \leftarrow c_0 \\
r \rightarrow & 26 &
\end{array}
$$

Notad que en cada paso se calcula uno de los coeficientes del cociente $(3x-7)$, se obtiene el polinomio *aux* y se actualiza el resto *r* mediante la correspondiente suma. El grado de los sucesivos restos va decreciendo en cada iteración. El resultado de la última suma es el resto definitivo de la división (26). Si $p_2 = 1$ este proceso no es válido y los resultados se han de devolver directamente.

Como ayuda para calcular *aux*, podéis usar la función de multiplicar un polinomio por un monomio que aparece en los ejemplos del directorio de la sesión. De todas formas, los cálculos también se pueden realizar sin disponer explícitamente de *aux*.

Podéis guardar los resultados como parámetros de salida (habrá que crearlos antes de llamar a la operación), lo que daría lugar a una accion, o bien podéis crear una clase que contenga un par de polinomios y usarla como resultado. Por último, aseguraos de que la división no modifica los polinomios originales.

3.5 Raíces enteras de un polinomio

Un entero e es raíz de un polinomio p si el resultado de evaluar p en e es 0, o equivalentemente, si la división de p entre $x-e$ es exacta (el resto es el polinomio 0). La multiplicidad de e como raíz de p es el mayor valor k que hace posible que la división de p entre $(x-e)^k$ sea exacta. Dicho de otro modo, la multiplicidad de e como raíz de p es el número de divisiones que pueden realizarse con $x-e$ como divisor, con p como dividendo de la primera división y usando en las siguientes el cociente de la inmediatamente anterior.

Por ejemplo, el polinomio $x-1$ tiene una raíz entera $e=1$, con multiplicidad 1. El polinomio x^3-3x-2 tiene dos raíces $e_1=-1$, de multiplicidad 2 y $e_2=2$, de multiplicidad 1.

En esta sesión nos centraremos en polinomios que no sean múltiplos de x, ya que el caso contrario se reduce fácilmente a éste. Podéis comprobarlo como ejercicio y programar la solución completa, que sólo requiere añadir una función más.

Considerando todo lo anterior, especificamos la siguiente función

función raiz_poli (p: poli) **retorna** v: vector, n: nat
{p no es múltiplo de x}
{n es la suma de las multiplicidades de las raíces enteras de p,
los valores de $v[1..n]$ son las raíces enteras de p y
cada una aparece tantas veces como su multiplicidad}

Solución 1

Un polinomio p posee, como máximo, grado(p) raíces enteras. Si el polinomio no es múltiplo de x, dichas raíces se encuentran entre los divisores (positivos o negativos) de su término independiente, coef$(p,0)$. Utilizando esta idea, la solución se obtiene comprobando si cada uno de los mencionados valores es una raíz y, en caso afirmativo, obteniendo su multiplicidad. Para ello, basta con aplicar repetidamente, pero con cuidado, la operación de división del apartado anterior.

Solución 2

Inspirándonos en la llamada *regla de Ruffini*, encontramos una solución más sencilla y eficiente. Si nos fijamos bien, todas las divisiones que se efectúan tienen como divisor un polinomio de la forma $x-e$. Podemos diseñar una función especial para realizar divisiones de esas características y utilizarla en lugar de la división normal.

La postcondición de la división de un polinomio p cualquiera entre $x-e$ se reescribe así

$$p = c*(x-e)+r \text{ y } \text{grado}(r)=0$$

Ahora, si denotamos el grado de p como g, tenemos dos casos. Si $g=0$ entonces los resultados son $c=0$ y $r=p$. En caso contrario, separamos los valores de cada coeficiente de p, que se pueden escribir en términos de los de c y r:

$$p[g] = c[g-1]$$
$$\forall \alpha : 0 < \alpha < g : p[\alpha] = -e*c[\alpha] + c[\alpha-1]$$
$$p[0] = -e*c[0] + r[0]$$

a partir de lo cual es sencillo derivar los coeficientes de c y r.