Trabajo práctico 2: Diseño

CalculadoraProgramable

Normativa

Límite de entrega: Domingo 3 de junio *hasta las 23:59 hs.* Enviar el zip al mail `algo2.dc+tp2@gmail.com`. Ver detalles de entrega abajo.

Normas de entrega: Ver "Información sobre la cursada" en el sitio Web de la materia. (`http://campus.exactas.uba.ar`)

1. Enunciado

Este TP consiste en diseñar módulos para implementar una calculadora programable similar al prototipo ya implementado en el TP anterior. Al igual que en el TP 1, un *programa* está dado por un conjunto de *rutinas*, cada una de las cuales consta de una lista de *instrucciones*, y hay ocho tipos de instrucciones (`oPush`, `oAdd`, `oSub`, `oMul`, `oRead`, `oWrite`, `oJump`, `oJumpz`). Las rutinas y variables se identifican por nombre (un `string`).

La principal diferencia con respecto al TP 1 es que la calculadora permite consultar el *valor histórico* de las variables. La calculadora comienza la ejecución en el instante 0. Cada vez que se ejecuta una instrucción, se incrementa en 1 el instante actual. La calculadora debe contar con una operación para observar cuál era el valor de una variable arbitraria x en un instante arbitrario $t \in \{0, 1, ..., \texttt{instante\_actual}\}$.

La calculadora se inicializa con un parámetro entero $W > 0$, que llamamos **capacidad de ventana**. Se deben proveer las siguientes operaciones, con las complejidades temporales en **peor caso** indicadas:

1. Dado un programa p, una rutina r y una capacidad de ventana W, construir una calculadora inicializada para ejecutar la rutina r del programa p — $O(\#p \cdot (|V| + |R|) + W \cdot \#V)$ donde:
   - $\#p$ es el tamaño del programa p, medido en cantidad total de instrucciones,
   - $|V|$ es la longitud del nombre más largo de alguna de las variables que aparecen en el programa,
   - $|R|$ es la longitud del nombre más largo de alguna de las rutinas que aparecen en el programa, incluyendo la rutina r,
   - $\#V$ es la cantidad total de variables distintas que aparecen en el programa,
   - W es la capacidad de ventana.
2. Determinar si la ejecución finalizó — $O(1)$.
3. Ejecutar la instrucción actual, actualizando correspondientemente el estado de la calculadora — $O(1)$. *Nota:* la restricción sobre la complejidad aplica a los ocho tipos de instrucciones por igual.
4. Darle valor a una variable arbitraria x — $O(|x|)$, donde $|x|$ es la longitud del nombre de x. A diferencia de la operación `oWrite`, este método **no** incrementa el instante actual.
5. Consultar el estado actual de la calculadora:
   - 5.1. Instante actual — $O(1)$.
   - 5.2. Nombre de la rutina actual — $O(1)$.
   - 5.3. Índice de la instrucción actual dentro de dicha rutina — $O(1)$.
   - 5.4. Valor de una variable arbitraria x en un instante arbitrario $t \in \{0, 1, ..., \texttt{instante\_actual}\}$. La complejidad de esta operación depende de si se cumplen las dos condiciones siguientes:
     1) **Acceso reciente.** El instante t consultado está dentro del intervalo de tamaño W que llega hasta el instante actual (es decir, $t \geq \texttt{instante\_actual} - W$).
     2) **Variable relevante.** La variable x aparece en el código fuente del programa.

     Si ambas condiciones se cumplen, la complejidad temporal debe ser $O(|x| + \log W)$ en peor caso, donde $|x|$ es la longitud del nombre de x. Si no se cumple alguna de las dos condiciones anteriores, no hay ninguna restricción sobre la complejidad temporal de esta operación.
   - 5.5. Valor de una variable arbitraria x en el instante actual — $O(|x|)$, donde $|x|$ es la longitud del nombre de x.
   - 5.6. Pila — $O(1)$.

Aclaraciones: No es necesario que la calculadora provea una operación para ejecutar un programa completo: eventualmente el usuario de la calculadora podría simular este comportamiento con un `while`. Para lograr la complejidad en peor caso de $O(1)$ para todas las instrucciones, se sugiere **convertir el programa que recibe la calculadora a una representación más conveniente**, antes de la ejecución.

1.1. Módulo Ventana(α)

Para diseñar la calculadora puede resultar útil utilizar un módulo VENTANA(α) con la interfaz que se detalla a continuación. En caso de que necesiten usarlo, deben completar el diseño, incluyendo la representación, invariante, algoritmos, etc.

El módulo VENTANA(α) representa un arreglo en el que pueden registrarse elementos secuencialmente. **Mantiene registro de los últimos W elementos registrados y "olvida" todos los anteriores.** El parámetro W se llama la *capacidad* de la ventana. La especificación del TAD Ventana(α) puede consultarse en el apéndice.

Interfaz del módulo VENTANA(α):

- **Parámetros formales:** género α.
- **Géneros:** `ventana(`α`)`.
- **Se explica con:** ventana(α).
- **Operaciones:**

  NUEVAVENTANA(**in** W : `nat`) $\rightarrow$ *res* : `ventana(`α`)`
  Crea una nueva ventana con la capacidad W indicada.
  Pre $\equiv \{\hat{W} > 0\}$
  Post $\equiv \{\widehat{res} =_{\text{obs}} \text{nuevaVentana}(\widehat{W})\}$
  Complejidad: $O(W)$.

  REGISTRAR(**in/out** v : `ventana(`α`)`, **in** a : α)
  Registra un elemento a al final de la ventana. Si la ventana excede la capacidad W, se mantiene registro de los últimos W elementos (incluyendo a) y se olvidan los anteriores.
  Pre $\equiv \{v_0 =_{\text{obs}} \widehat{v}\}$
  Post $\equiv \{\hat{v} =_{\text{obs}} \text{registrar}(v_0, \hat{a})\}$
  Complejidad: $O(1)$.

  CAPACIDAD(**in** v : `ventana(`α`)`) $\rightarrow$ *res* : `nat`
  Devuelve la capacidad de la ventana.
  Pre $\equiv \{\text{true}\}$
  Post $\equiv \{\widehat{res} =_{\text{obs}} \text{capacidad}(\hat{v})\}$
  Complejidad: $O(1)$.

  TAM(**in** v : `ventana(`α`)`) $\rightarrow$ *res* : `nat`
  Devuelve el tamaño (cantidad de elementos) de la ventana. Es un entero entre 0 y W inclusive.
  Pre $\equiv \{\text{true}\}$
  Post $\equiv \{\widehat{res} =_{\text{obs}} \text{long}(\text{elementos}(\hat{v}))\}$
  Complejidad: $O(1)$.

  $\bullet[\bullet]$(**in** v : `ventana(`α`)`, **in** i : `nat`) $\rightarrow$ *res* : α
  Devuelve el i-ésimo elemento de la ventana. Si la ventana tiene tamaño n, el elemento 0-ésimo es el elemento más viejo del cual aún se tiene registro, y el elemento $(n-1)$-ésimo es el último elemento registrado.
  Pre $\equiv \{\hat{i} < \text{tam}(\hat{v})\}$
  Post $\equiv \{\widehat{res} =_{\text{obs}} \text{iésimo}(\text{elementos}(\hat{v}),\hat{i})\}$
  Complejidad: $O(1)$.

La operación iésimo es la usual:

iésimo : secu(α) $s \times$ nat $i \longrightarrow \alpha$ $\{i < \text{long}(s)\}$

iésimo(s, i) $\equiv$ **if** $i = 0$? **then** prim(s) **else** iésimo(fin(s), pred(i)) **fi**

2. Documentación a entregar

Todos los módulos diseñados deben contar con las siguientes partes.

1. **Interfaz.**
   - 1.1. *Tipo abstracto ("se explica con ...").* Género (TAD) que sirve para explicar las instancias del módulo, escrito en el lenguaje de especificación **formal** de la materia. Pueden utilizar la especificación que se incluye en el apéndice.
   - 1.2. *Signatura.* Listado de todas las funciones públicas que provee el módulo. La signatura se puede escribir, dependiendo de sus preferencias:
     - Con la notación de módulos de la materia:
       apilar(**in**/**out** pila : PILA, **in** x : ELEMENTO)
     - Con notación de C++:
       `void Pila::apilar(const Elemento& x)`
   - 1.3. *Contrato.* Precondición y postcondición de todas las funciones públicas. Las precondiciones de las funciones de la interfaz deben estar expresadas **formalmente** en lógica de primer orden.[1]
   - 1.4. *Complejidades.* Complejidades de todas las funciones públicas, cuando corresponda.
   - 1.5. *Aspectos de* aliasing. De ser necesario, aclarar cuáles son los parámetros y resultados de los métodos que se pasan por copia y cuáles por referencia, y si hay *aliasing* entre algunas de las estructuras.
2. **Implementación.**
   - 2.1. *Representación ("se representa con ...").* Módulo con el que se representan las instancias del módulo actual.
   - 2.2. *Invariante de representación.* Puede estar expresado en lenguaje natural o formal.
   - 2.3. *Función de abstracción.* Puede estar expresada en lenguaje natural o formal.
   - 2.4. *Algoritmos.* Pueden estar expresados en pseudocódigo, usando si es necesario la notación del lenguaje de módulos de la materia o notación tipo C++. Las pre y postcondiciones de las funciones auxiliares pueden estar expresadas en lenguaje natural (no es necesario que sean formales). Se debe indicar de qué manera los algoritmos cumplen con el contrato declarado en la interfaz y con las complejidades pedidas. No se espera una demostración formal, pero sí una justificación adecuada.
3. **Servicios usados.** Módulos que se utilizan, detallando las complejidades, *aliasing* y otros aspectos que dichos módulos deben proveer para que el módulo actual pueda cumplir con su interfaz.

Sobre el uso de lenguaje natural y formal

Las precondiciones y poscondiciones de las funciones auxiliares, el invariante y la función de abstracción pueden estar expresados en lenguaje natural. No es necesario que sean formales. Asimismo, los algoritmos pueden estar expresados en pseudocódigo. Por otro lado, está permitido que utilicen fórmulas en lógica de primer orden en algunos lugares puntuales, si consideran que mejora la presentación o subsana alguna ambigüedad. El objetivo del diseño es convencer al lector, y a ustedes mismos, de que la interfaz pública se puede implementar usando la representación propuesta y respetando las complejidades pedidas. Se recomienda aplicar el sentido común para priorizar la **claridad** y **legibilidad** antes que el rigor lógico por sí mismo. Por ejemplo:

Más claro	**Menos claro**
"Cada clave del diccionario D debe ser una lista sin elementos repetidos." ✓ "sinRepetidos?(claves(D))" ✓	"No puede haber repetidos." (¿En qué estructura?).
"Ordenar la lista A usando mergesort." ✓ "A.mergesort()" ✓	"Ordenar los elementos." (¿Qué elementos? ¿Cómo se ordenan?).
"Para cada tupla (x, y) en el conjunto C { x.apilar(y) n++ }" ✓	"Miro las tuplas del conjunto, apilo la segunda componente en la primera y voy incrementando un contador." (Ambiguo y difícil de entender).

[1] Si la implementación requiere usar funciones auxiliares, sus pre y postcondiciones pueden estar escritas en lenguaje natural, pero esto no forma parte de la interfaz.

A. Especificación formal

Usaremos los siguientes tipos auxiliares. El tipo Int representa el tipo de las constantes numéricas que maneja la calculadora. Omitimos su especificación formal. Al implementarlo, esperamos que sea un entero de precisión acotada (un `int` en C++). El tipo String es un arreglo dimensionable de caracteres, extendido con una operación $\bullet = \bullet : \text{string} \times \text{string} \to \text{bool}$ para compararlas por igualdad. Cada caracter es un número entre 0 y 255. El TAD Operación representa un código de operación. Tiene 8 generadores y un único observador $\bullet = \bullet : \text{operación} \times \text{operación} \to \text{bool}$ para compararlas por igualdad.

- **TAD** Char es Enum(0, 1, .., 255)
- **TAD** String es Arreglo dimensionable(Char)
- **TAD** Variable es String
- **TAD** Rutina es String
- **TAD** Operación es Enum(`oPush`, `oAdd`, `oSub`, `oMul`, `oRead`, `oWrite`, `oJump`, `oJumpz`)

A.1. Instrucción

TAD Instrucción

géneros instrucción

usa Int, Variable, Rutina, Operación

exporta *observadores*, *generadores*

observadores básicos

op	: instrucción	$\longrightarrow$	operación		
constanteNumérica	: instrucción i	$\longrightarrow$	int		$\{\text{op}(i) = \texttt{oPush}\}$
nombreVariable	: instrucción i	$\longrightarrow$	variable		$\{\text{op}(i) = \texttt{oRead} \vee \text{op}(i) = \texttt{oWrite}\}$
nombreRutina	: instrucción i	$\longrightarrow$	rutina		$\{\text{op}(i) = \texttt{oJump} \vee \text{op}(i) = \texttt{oJumpz}\}$

generadores

iPush	: int	$\longrightarrow$	instrucción
iAdd	:	$\longrightarrow$	instrucción
iSub	:	$\longrightarrow$	instrucción
iMul	:	$\longrightarrow$	instrucción
iRead	: variable	$\longrightarrow$	instrucción
iWrite	: variable	$\longrightarrow$	instrucción
iJump	: rutina	$\longrightarrow$	instrucción
iJumpz	: rutina	$\longrightarrow$	instrucción

axiomas

($\forall i$: instrucción, $\forall n$: int, $\forall v$: variable, $\forall r$: rutina)

$\text{op}(\text{iPush}(n))$	$\equiv$	`oPush`	$\text{constanteNumérica}(\text{iPush}(n)) \equiv n$
$\text{op}(\text{iAdd})$	$\equiv$	`oAdd`	
$\text{op}(\text{iSub})$	$\equiv$	`oSub`	$\text{nombreVariable}(\text{iRead}(v)) \equiv v$
$\text{op}(\text{iMul})$	$\equiv$	`oMul`	$\text{nombreVariable}(\text{iWrite}(v)) \equiv v$
$\text{op}(\text{iRead}(v))$	$\equiv$	`oRead`	
$\text{op}(\text{iWrite}(v))$	$\equiv$	`oWrite`	$\text{nombreRutina}(\text{iJump}(r)) \equiv r$
$\text{op}(\text{iJump}(r))$	$\equiv$	`oJump`	$\text{nombreRutina}(\text{iJumpz}(r)) \equiv r$
$\text{op}(\text{iJumpz}(r))$	$\equiv$	`oJumpz`	

Fin TAD

A.2. Programa

TAD Programa

géneros programa

usa Conjunto, Rutina, Instrucción

exporta *observadores*, *generadores*

observadores básicos

rutinas : programa $\longrightarrow$ conj(rutina)

longitud : programa p $\times$ rutina r $\longrightarrow$ nat $\{r \in \text{rutinas}(p)\}$

instrucción : programa p $\times$ rutina r $\times$ nat n $\longrightarrow$ instrucción $\{r \in \text{rutinas}(p) \wedge_{\text{L}} n < \text{longitud}(p, r)\}$

generadores

nuevoPrograma $\longrightarrow$ programa

agInstrucción : programa $\times$ rutina $\times$ instrucción $\longrightarrow$ programa

axiomas

($\forall p$: programa, $\forall r, r'$: rutina, $\forall i$: instrucción, $\forall n$: nat)

rutinas(nuevoPrograma) $\equiv \emptyset$

rutinas(agInstrucción(p, r, i)) $\equiv$ Ag(r, rutinas(p))

longitud(agInstrucción(p, r, i), r') $\equiv$ **if** $r' \in$ rutinas(p) **then** longitud(p, r') **else** 0 **fi** + **if** $r = r'$ **then** 1 **else** 0 **fi**

instrucción(agInstrucción(p, r, i), r', n) $\equiv$ **if** $r = r' \wedge$ longitud(agInstrucción(p, r, i), r') $= n+1$ **then**
i
else
instrucción(p, r', n)
fi

Fin TAD

A.3. Calculadora

TAD Calculadora

géneros calculadora

usa Bool, Nat, Pila, Int, Variable, Rutina, Instrucción, Programa

exporta *observadores*, *generadores*, ejecutando?, valorActualVariable

observadores básicos

programa	: calculadora	$\longrightarrow$	programa	
rutinaActual	: calculadora	$\longrightarrow$	rutina	
índiceInstrucciónActual	: calculadora	$\longrightarrow$	nat	
pila	: calculadora	$\longrightarrow$	pila(int)	
instanteActual	: calculadora	$\longrightarrow$	nat	
valorHistóricoVariable	: calculadora c × variable × nat t	$\longrightarrow$	int	$\{t \leq \text{instanteActual}(c)\}$

generadores

nuevaCalculadora	: programa × rutina	$\longrightarrow$	calculadora	
ejecutarUnPaso	: calculadora c	$\longrightarrow$	calculadora	{ejecutando?(c)}
asignarVariable	: calculadora × variable × int	$\longrightarrow$	calculadora	

otras operaciones

ejecutando?	: calculadora	$\longrightarrow$	bool	
instrucciónActual	: calculadora c	$\longrightarrow$	instrucción	{ejecutando?(c)}
opActual	: calculadora c	$\longrightarrow$	operación	{ejecutando?(c)}
haySalto?	: calculadora c	$\longrightarrow$	bool	{ejecutando?(c)}
primeroPila	: calculadora	$\longrightarrow$	int	
segundoPila	: calculadora	$\longrightarrow$	int	
pilaSinUno	: calculadora	$\longrightarrow$	pila(int)	
pilaSinDos	: calculadora	$\longrightarrow$	pila(int)	
valorActualVariable	: calculadora × variable	$\longrightarrow$	int	
escribiendoVariable?	: calculadora × variable	$\longrightarrow$	bool	

axiomas

($\forall c$: calculadora, $\forall t$: nat, $\forall p$: programa, $\forall r$: rutina, $\forall v, v'$: variable, $\forall n$: int)

programa(nuevaCalculadora(p, r)) $\equiv$ p
programa(ejecutarUnPaso(c)) $\equiv$ programa(c)
programa(asignarVariable(c,v,n)) $\equiv$ programa(c)

rutinaActual(nuevaCalculadora(p, r)) $\equiv$ r
rutinaActual(ejecutarUnPaso(c)) $\equiv$ **if** haySalto?(c) **then**
nombreRutina(instrucciónActual(c))
else
rutinaActual(c)
fi
rutinaActual(asignarVariable(c,v,n)) $\equiv$ rutinaActual(c)

índiceInstrucciónActual(nuevaCalculadora(p, r)) $\equiv$ 0
índiceInstrucciónActual(ejecutarUnPaso(c)) $\equiv$ **if** haySalto?(c) **then**
0
else
1 + índiceInstrucciónActual(c)
fi
índiceInstrucciónActual(asignarVariable(c,v,n)) $\equiv$ índiceInstrucciónActual(c)

pila(nuevaCalculadora(p, r)) $\equiv$ vacía

pila(ejecutarUnPaso(c)) ≡
if opActual(c) = `oPush` **then** apilar(constanteNumérica(instrucciónActual(c)), pila(c))
else if opActual(c) = `oAdd` **then** apilar(segundoPila(c) + primeroPila(c), pilaSinDos(c))
else if opActual(c) = `oSub` **then** apilar(segundoPila(c) - primeroPila(c), pilaSinDos(c))
else if opActual(c) = `oMul` **then** apilar(segundoPila(c) * primeroPila(c), pilaSinDos(c))
else if opActual(c) = `oRead` **then** apilar(
valorActualVariable(c, nombreVariable(instrucciónActual(c))),
pila(c))
else if opActual(c) = `oWrite` **then** pilaSinUno(c)
else if opActual(c) = `oJump` **then** pila(c)
else (opActual(c) = `oJumpz`) pilaSinUno(c) **fi fi fi fi fi fi fi**
pila(asignarVariable(c,v,n)) ≡ pila(c)

instanteActual(nuevaCalculadora(p, r)) ≡ 0
instanteActual(ejecutarUnPaso(c)) ≡ 1 + instanteActual(c)
instanteActual(asignarVariable(c,v,n)) ≡ instanteActual(c)

valorHistóricoVariable(nuevaCalculadora(p, r), v) ≡ 0
valorHistóricoVariable(ejecutarUnPaso(c), v, t) ≡ **if** instanteActual(c) = t **then**
if escribiendoVariable?(v) **then**
primeroPila(c)
else
valorHistóricoVariable(c, v, $t - 1$)
fi
else
valorHistóricoVariable(c, v, t)
fi
valorHistóricoVariable(asignarVariable(c,v,n), v', t) ≡ **if** instanteActual(c) = $t \wedge v = v'$ **then**
n
else
valorHistóricoVariable(c, v', t)
fi

ejecutando?(c) ≡ rutinaActual(c) ∈ rutinas(programa(c)) $\wedge_{\text{L}}$
índiceInstrucciónActual(c) < longitud(programa(c), rutinaActual(c))

instrucciónActual(c) ≡ instrucción(programa(c), rutinaActual(c), índiceInstrucciónActual(c))

opActual(c) ≡ op(instrucciónActual(c)

haySalto?(c) ≡ op(instrucciónActual(c)) = `oJump` $\vee$
(op(instrucciónActual(c)) = `oJumpz` $\wedge_{\text{L}}$ primeroPila(c) = 0)

primeroPila(c) ≡ **if** vacía?(pila(c)) **then** 0 **else** tope(pila(c)) **fi**

segundoPila(c) ≡ **if** vacía?(pila(c)) $\vee_{\text{L}}$vacía?(desapilar(pila(c))) **then**
0
else
tope(desapilar(pila(c)))
fi

pilaSinUno(c) ≡ **if** vacía?(pila(c)) **then** vacía **else** desapilar(pila(c)) **fi**

pilaSinDos(c) ≡ **if** vacía?(pila(c)) $\vee_{\text{L}}$vacía?(desapilar(pila(c))) **then**
vacía
else
desapilar(desapilar(pila(c)))
fi

valorActualVariable(c, v) ≡ valorHistóricoVariable(c, v, instanteActual(c))

$$\begin{array}{rcll} \text{escribiendoVariable?}(c,v) & \equiv & \text{ejecutando?}(c) & \wedge_{\text{L}} \quad \text{opActual}(c) = \texttt{oWrite} \\ & & & \wedge_{\text{L}} \quad \text{nombreVariable}(\text{instrucciónActual}(c)) = v \end{array}$$

Fin TAD

A.4. Ventana(α)

TAD Ventana(α)

géneros ventana

usa NAT

exporta *observadores*, *generadores*

observadores básicos

capacidad : ventana(α) $\longrightarrow$ nat
elementos : ventana(α) $\longrightarrow$ secu(α)

generadores

nuevaVentana : nat c $\longrightarrow$ ventana(α) $\{c > 0\}$
registrar : ventana(α) $\times$ α $\longrightarrow$ ventana(α)

axiomas

($\forall v$: ventana(α), $\forall a, b$: α, $\forall i$: nat)

capacidad(nuevaVentana(w)) $\equiv$ w
capacidad(registrar(v, a)) $\equiv$ capacidad(v)

elementos(nuevaVentana(w)) $\equiv$ <>
elementos(registrar(v, a)) $\equiv$ **if** long(elementos(v)) $<$ capacidad(v) **then**
elementos(v) $\circ$ a
else
fin(elementos(v)) $\circ$ a
fi

Fin TAD