Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
Traductores e Interpretadores (CI-3725)
Abril - Julio, 2011

Definición del Lenguaje Vectorinox

NOTA: La presente definición de lenguaje no es final; algunos detalles podrían ser refinados en el futuro. Cualquier cambio realizado le será debidamente reportado en las horas de clase del curso. Adicionalmente, se espera de usted que consulte aquellos elementos que le parezcan ambiguos o que necesiten una descripción más clara, puesto que en esta definición se han dejado intencionalmente de lado algunos detalles ya que es de interés para los que impartimos el curso que sean los estudiantes los que encuentren tales aspectos.

0. Introducción

Vectorinox es un lenguaje de programación imperativo, fuertemente tipeado, con chequeos de tipo estáticos, e interpretado. Vectorinox se distingue de otros lenguajes pues cuenta con los tipos vector y matriz como tipos nativos del lenguaje, incluyendo varias operaciones básicas entre estos tipos. Esta característica permite a sus usuarios escribir rápida y cómodamente programas que recurran al cálculo vectorial para la obtención de resultados.

1. Tipos de datos

Los siguientes son los tipos de datos definidos en Vectorinox:

- **`num`:** representa un número real. Queda a juicio del implementador del lenguaje el número de bits para la representación a bajo nivel del número.
- **`vec`:** representa un vector de números reales. Los vectores pueden ser fila o columna. No pueden existir vectores de tamaño negativo. Podrán existir vectores de tamaño cero (0) pero cualquier operación que se realice sobre este deberá generar un error a tiempo de ejecución.
- **`mat`:** representa una matriz de números reales. Todas las matrices son bidimencionales. No pueden existir matirces con algún tamaño negativo. Podrán existir matrices con alguna dimensión de tamaño cero (0) pero cualquier operación que se realice sobre esta deberá generar un error a tiempo de ejecución.
- ***String*:** representa una cadena de caracteres. El lenguaje sólo manejará *strings* estáticos; no podrán existir variables de este tipo.
- **Booleano:** representa un valor del conjunto verdadero, falso . El lenguaje sólo manejará valores booleanos estáticos; no podrán existir variables de este tipo.

1.1. Transformación automática de tipos (Coerción)

Vectorinox sólo admitirá un tipo de coerción: un elemento del tipo vector podrá ser usado en cualquier contexto en el que se necesite un elemento del tipo matriz. Esta transformación se realizará de la siguiente manera:

- Un vector fila de tamaño N se transforma en una matriz de tamaño $1 \times N$.
- Un vector columna de tamaño N se transforma en una matriz de tamaño $N \times 1$.

2. Expresiones

Vectorinox acepta un amplia gama de expresiones, las cuales deberán ser evaluadas de izquierda a derecha.

2.1. Constantes

Vectorinox acepta expresiones constantes para los tipos `num`, string y booleano.

- **Para el tipo `num`.** Una expresión constante de tipo num es:
  - Un número entero positivo o negativo, e.g. `3`, `-145`.
  - Un número entero positivo o negativo con un punto concatenado al final, e.g. `3.`, `-145.`.
  - Un número entero positivo o negativo, seguido por un punto, seguido por un número entero positivo con tantos ceros a la izquierda como se desee, e.g. `4.2`, `-123.00004`, `1.0`.
  - Un punto seguido de un número entero positivo con tantos ceros a la izquierda como se desee, e.g. `.12`, `.00003`.
- **Para el tipo *string*.** Una expresión constante del tipo *string* es una cadena de caracteres ASCII que comienza y termina con comillas dobles (") o comillas simples (’). Los caracteres no imprimibles ASCII se pueden colocar dentro del *string* en su forma escapada con un slash invertido (\), esto implica que para colocar el caracter en el *string* como un caracter imprimible será necesario escapar su funcionalidad, es decir, escribir \\.

  Adicionalemente, si el *string* comienza por comillas dobles ("), si se desea colocar dicho caracter como un elemento del *string*, será necesario escaparlo. Lo mismo sucede si el *string* comienza por comillas simples. Los siguientes son algunos ejemplos de *strings* correctos:

  ```
  "Hola Mundo"
  ’Este es un "String" correcto’
  "Se debe usar el símbolo \"\\\" para ’escapar’ caracteres\n"
  ```

- **Para el tipo booleano.** Las expresiones booleanas `true` y `false` podrán ser usadas libremente en el lenguaje.

2.2. Variables

En Vectorinox, los identificadores de variables son un conjunto de caracteres alfanuméricos y *underscores* (_), con la restricción de que el nombre debe comenzar por un caracter alfabético.

- Ejemplos de nombres permitidos: `i, FOO, x2, nombre_largo`.
- Ejemplos de nombres no permitidos: `2x, _hola, 75_2`.

Adicionalmente, para los identificadores de variables no hay distinción de mayúsculas y minúsculas, es decir, el identificador `var` se considera igual al identicicador `Var` ó `VAR`.

2.3. Construcción de vectores y matrices

- Un vector es una lista de expresiones de tipo `num` separada por comas o punto y comas, dicha lista se debe encontrar entre llaves { }, e.g.:

  ```
  {  e1, e2, ... , en }
  {  e1; e2; ... ; en }
  ```

  dónde `e1, e2, ... , en` son expresiones de tipo `num`. Si la lista está separa por comas el resultado será un **vector fila**, si está separada por punto y comas será un **vector columna**.

- Una matriz es una lista de expresiones de tipo vector fila separada por punto y comas, dicha lista se debe encontrar entre llaves { }, e.g.:

  ```
  {  e1, e2, en }
  ```

  dónde `e1, e2, ... , en` , son expresiones de tipo vector fila y todos los vectores son del mismo tamaño. Esto generaría una matriz de tamaño `$e1` $\times n$.

- Además, una matriz puede definirse de la siguiente forma:

  ```
  {  e11, e12, ..., e1m; e21, e22, ..., e2m; ... ; en1, en2, ..., enm }
  ```

  dónde `e11, e12, ..., e1m; e21, e22, ..., e2m; ... ; en1, en2, ..., enm` son expresiones de tipo `num`. Esto generaría una matriz de tamaño $m \times n$.

2.4. Operadores

Vectorinox cuenta con operadores tanto unarios como binarios y ternarios, algunos de ellos sobrecargados, a continuación se define una lista exhaustiva de dichos operadores por cada tipo de datos del lenguaje. La precedencia y asociatividad de todos los operadores de Vectorinox es igual a la de sus equivalentes en el lenguaje de programación C. Adicionalmente:

- El operador `.` tiene la misma prioridad y asociatividad que el operador `*`.
- Los operadores `$` y `@` tienen la misma precedencia que el operador unario `-`.

Se podrán usar libremente los paréntesis `( )` para agrupar expresiones y alterar la asociatividad de los operadores.

2.4.1. Operadores para el tipo `num`

Operadores binarios

Operador	Operación	Tipo arg. izq.	Tipo arg. der.	Tipo Resultado
`+`	suma de reales	`num`	`num`	`num`
`-`	resta de reales	`num`	`num`	`num`
`*`	multiplicación de reales	`num`	`num`	`num`
`/`	división de reales	`num`	`num`	`num`
`%`	módulo de reales	`num`	`num`	`num` (positivo)
`**`	exponenciación de reales	`num`	`num`	`num`
`*`	multiplicación de vector por escalar	`num`	`vec`	`vec`
`*`	multiplicación de matriz por escalar	`num`	`mat`	`mat`
`<`	menor que	`num`	`num`	booleano
`>`	mayor que	`num`	`num`	booleano
`<=`	menor o igual que	`num`	`num`	booleano
`>=`	mayor o igual que	`num`	`num`	booleano
`=`	igual a	`num`	`num`	booleano
`!=`	desigual a	`num`	`num`	booleano

Operadores unarios

Operador	Operación	Tipo arg.	Tipo Resultado
`-`	Cambio de signo	`num` (derecha)	`num`

2.4.2. Operadores para el tipo `vec`

Operadores binarios

Operador	Operación	Tipo arg. izq.	Tipo arg. der.	Tipo Resultado
`+`	suma de vectores	`vec`	`vec`	`vec`
`-`	resta de vectores	`vec`	`vec`	`vec`
`*`	producto cruz de vectores	`vec`	`vec`	`vec`
`.`	producto punto de vectores	`vec`	`vec`	`vec`
`*`	multiplicación de vector por un escalar	`vec`	`num`	`vec`
`/`	división de vector por escalar	`vec`	`num`	`vec`
`[ ]`	acceso a elemento	`vec`	`num` (≥ 1)	`num`

Los elementos de los vectores estarán indexados desde el número uno hasta el tamaño del vector, i.e. para acceder al primer elemento del vector, el segundo argumento del operador `[ ]` deberá ser el número uno, e.g.:

- Sea `vc` un vector columna
  - `vc[1]` retorna el elemento de la primera fila del vector.
  - `vc[3]` retorna el elemento de la tercera fila del vector.
- Sea `vf` un vector fila
  - `vf[1]` retorna el elemento de la primera columna del vector.
  - `vf[3]` retorna el elemento de la tercera columna del vector.

Operadores unarios

Operador	Operación	Tipo arg.	Tipo Resultado
`$`	número de filas	`vec (derecha)`	`num`
`@`	número de columnas	`vec (derecha)`	`num`
`'`	trasponer	`vec` (izquierda)	`vec`

Operadores ternarios

Operador	Operación	Tipo arg. 1	Tipo arg. 2	Tipo arg. 3	Tipo Resultado
`[ : ]`	Segmento	`vec`	`num`	`num`	`vec`

El operador ternario funciona para generar segmentos de un vector. El vector resultado será del mismo tipo que el original (fila o columna). Este operador funciona de la misma manera que el operador de segmentos del lenguaje de programación Python.

2.4.3. Operadores para el tipo `mat`

Operadores binarios

Operador	Operación	Tipo arg. izq.	Tipo arg. der.	Tipo Resultado
`+`	suma de matrices	`mat`	`mat`	`mat`
`-`	resta de matrices	`mat`	`mat`	`mat`
`*`	producto de matrices	`mat`	`mat`	`mat`
`*`	multiplicación de matriz por escalar	`mat`	`num`	`mat`
`/`	división de matriz por escalar	`mat`	`num`	`mat`

Operadores unarios

Operador	Operación	Tipo arg.	Tipo Resultado
`$`	número de filas	`mat (derecha)`	`num`
`@`	número de columnas	`mat (derecha)`	`num`
`'`	trasponer	`mat` (izquierda)	`mat`

Operadores ternarios

Operador	Operación	Tipo arg. 1	Tipo arg. 2	Tipo arg. 3	Tipo Resultado
`[ , ]`	Acceso a elemento	`mat`	`num` (≥ 1)	`num` (≥ 1)	`num`

El segundo y tercer argumento del operador `[ , ]` corresponden respectivamente al número de fila y columna del elemento a acceder, los elementos de las filas y columnas de las matrices serán indexados desde el número uno hasta el tamaño de la fila o columna, e.g.: Sea `A` una matriz de tamaño 4×5, entonces:

- `A[1,5]` retorna el elemento de la primera fila y quinta columna de `A`
- `A[3,1]` retorna el elemento de la tercera fila y primera columna de `A`

Otros operadores

Las matrices contarán con un operador de **segmentos** o **submatrices** muy parecido al operador de segmentos de los vectores, la cual funcionará de la siguiente manera:

Sea `A` una matriz, y `a`, `b`, `c`, `d`, expresiones de tipo `num`, la expresión `A[ a:b , c:d ]` generará como resultado una submatriz de tamaño `(b-a+1)` $\times$ `(d-c+1)` con los elementos comprendidos en el recuadro de la matriz `A` cuya esquina superior-izquierda es el elemento (`a`,`b`) y la esquina inferior derecha el elemento (`c`,`d`). Además se cumple que:

- Si se suprime al elemento `a` o `c`, sería equivalente a haber escrito el número `1`.
- Si se suprime al elemento `b` o `d`, sería equivalente a haber escrito la expresión `$A` u `@A` respectivamente.

Por ejemplo, sea `A` la matriz:

```
|  1   2   3   4  |
|  5   6   7   8  |
|  9  10  11  12  |
```

la expresión `A[ 2:3, 2:3 ]`, equivalente a `A[ 2:, 2:3 ]`, generaría la matriz:

```
|  6   7 |
| 10  11 |
```

mientras que la expresión `A[ 1:2, 2:4 ]`, equivalente a `A[ :2, 2: ]`, generaría la matriz:

```
| 2  3  4 |
| 6  7  8 |
```

y la expresión `A[ 1:1, 1:4 ]`, equivalente a `A[ :1, : ]`, generaría la matriz:

```
|  1  2  3  4  |
```

2.4.4. Operadores para el tipo *string*

No hay operadores para el tipo *string*.

2.4.5. Operadores para el tipo booleano

Operadores binarios

Operador	Operación	Tipo arg. izq.	Tipo arg. der.	Tipo Resultado
`&&`	conjunción con cortocircuito	booleano	booleano	booleano
`||`	disjunción con cortocircuito	booleano	booleano	booleano
`=`	equivalencia	booleano	booleano	booleano

Operadores unarios

Operador	Operación	Tipo arg.	Tipo Resultado
`!`	negación lógica	booleano (derecha)	booleano

2.5. Llamadas a funciones

Una llamada a una función consta del identificador de la función seguido de unos paréntesis `( )` y entre ellos una lista separada por comas de expresiones, cuyos tipos deberán corresponder a los tipos de la firma de la función. En la sección 3.2 se explica con detalle cómo definir a una función.

Las siguientes son llamadas válidas a funciones:

```
f(3, 4, a, b+c)
g(v[2:3], a.c*4, $A)
```

2.6. Funciones embebidas

Vectorinox cuenta con las siguientes funciones embebidas:

- `zeroes`
- `range`
- `eye`

2.6.1. Función `zeroes`

La función zeroes puede tomar un argumento o dos argumentos del tipo `num`.

- Si la función recibe un argumento, retorna un vector fila con la cantidad de elementos indicados. Por ejemplo, `zeroes(4)` genera un vector fila de 4 elementos.
- Si la función recibe dos argumentos, retorna una matriz cuya cantidad de filas y columnas corresponden respectivamente a los dos valores de los argumentos pasados a la función. Por ejemplo, `zeroes(3, 2)` devuelve una matriz de 3×2.

2.6.2. Función `range`

La función range recibe un argumento del tipo `num`, y retorna un vector fila con la cantidad de elementos indicados, y cuyos valores corresponden a su respectivo índice, e.g.:

- range(5) deberá retornar el vector { `1, 2, 3, 4, 5` }

2.6.3. Función `eye`

Esta función recibirá un solo elemento de tipo `num` y retornará una matriz identidad de tamaño `N` × `N` donde `N` es el valor del argumento pasado a la función.

3. Estructura de un programa de Vectorinox

En esta sección se presentará la estructura de un programa en Vectorinox. *NOTA:* Todo elemento encerrado entre corchetes es opcional.

Un programa en Vectorinox se escribe de esta forma:

```
[declaración de subrutinas]
instruccion
```

3.1. Definición de funciones

El lenguaje Vectorinox permite definir funciones. Estas funciones se definen de la siguiente forma:

```
define nombre(lista de variables) of type tipo as
    instruccion
```

tal que la lista de variables es una lista (posiblemente vacía) de definiciones de variable, de la forma `nombre: tipo`. El nombre de una función es un identificador de la misma forma que un identificador de variable (Sección 2.2), mientras que `tipo` corresponde a un identificador de tipo (sección 1).

3.2. Instrucciones del lenguaje

A continuación, se presentan las distintas instrucciones válidas en el lenguaje Vectorinox.

3.2.1. Asignación

Las asignaciones se escriben de la siguiente forma:

```
asignable := expresion
```

donde `asignable` es una expresión asignable y `expresion` una expresión del tipo correspondiente, expuestos en la sección 2. Una expresión asignable puede ser o una variable, un elemento de un vector o matriz, o un segmento de vector/matriz. Algunos ejemplos de asignación son los siguientes:

```
a := 2
b := {4, 3; 2, 5}
b[1] := 3
b[2, 1] := 4
c[2:3, 2:4] := {1, 2, 3; 4, 5, 6}
```

3.2.2. Bloques de instrucciones

Si se desea ejecutar o más instrucciones en secuencia, estas deben residir en un bloque de instrucciones. Dicho bloque presenta la siguiente sintaxis:

```
begin
    [declaraciones de variable]
    lista de instrucciones
end
```

donde `declaraciones de variable` corresponde a la declaración de un conjunto de variables y `lista de instrucciones` es una lista de instrucciones separadas usando punto y coma (;). La última instrucción no debe terminar con dicho punto y coma.

Se pueden definir variables locales dentro de un bloque. Estas variables solo existen a nivel del bloque (y de otros bloques que estén dentro de éste). Se pueden usar nombres definidos en bloques externos – en este caso, el nombre corresponde a la variable declarada adentro.

Declaración de variables

En el lenguaje Vectorinox, toda variable debe ser declarada antes de ser usada. Dicha declaración se escribe de la siguiente forma:

```
vars
    -lista de variables-
```

donde `lista de variables` es una lista, separada por punto y coma, de definiciones de la forma:

```
nombre_1 [, nombre_2, ..., nombre_n] : tipo
```

3.2.3. Selección condicional

Vectorinox presenta una construcción para control de flujo condicional, escrita de la siguiente forma:

```
if condicion then
    instruccion1
[ else
    instruccion2
]
```

donde `condicion` corresponde a una expresión cuyo valor evaluado debe ser un valor booleano.

3.2.4. Iteración condicional

Vectorinox presenta una instrucción de iteración condicional, cuya sintaxis es la siguiente:

```
while condicion do
    instruccion
```

3.2.5. Iteración sobre un vector/matriz

Vectorinox cuenta, además, con una instrucción que permite iterar sobre un vector o matriz, permitiendo poder iterar sobre los valores de esta. Esta instrucción se escribe de la siguiente forma:

```
foreach nombre in expresion do
    instruccion
```

Observe que expresion debe evaluarse en un vector o una matriz. En el caso que sea un vector, cada iteración asigna a la variable nombre cada elemento en el vector dado, mientras que en el caso que sea una matriz, cada iteración hace que variable represente una fila de la matriz evaluada. `variable` no puede estar declarada al momento de ejecutar la instrucción.

Las instrucciones dentro de este tipo de iteración no pueden cambiar el valor de la variable `nombre`.

3.2.6. Instrucción `read`

La instrucción `read` permite leer de entrada estándar uno o varios valores y asignarlos a la variable. Esta instrucción se escribe de la siguiente forma:

```
read asignable
```

Si `asignable` es de tipo `num`, entonces se solicita una variable. Si en cambio es de tipo `vec` o `mat`, se solicitará en entrada estandar todos los valores necesarios de manera secuencial.

3.2.7. Instrucción `write`

La instrucción `write` permite imprimir en salida estándar una cadena de caracteres o variables. Se escribe de la siguiente forma:

```
write expresion_1 [, expresion_2, ..., expresion_n]
```

donde `expresion_1, ... expresion_n` corresponden o a cadenas de caracteres (strings) o a expresiones. Entre cada expresión debe imprimirse un espacio para separarlos. Dependiendo del tipo de la expresión, se debe imprimir de una manera partícular:

- Si la expresión es un string, imprimirlo en pantalla. Si dicho string contiene caracteres escapados (por ejemplo, `\n`), debe de imprimirse dicho escape (usando el mismo ejemplo, un salto de línea).
- Si la expresión es de tipo `num`, imprimir el número.
- Si la expresión es de tipo `vect` o `mat`, imprimirlo de manera legible. Por ejemplo, una matriz 3×3 se debe imprimir de la siguiente forma:

```
| 17.2    4    3 |
|    3    9    5 |
|   42   17 3.14 |
```

3.2.8. Instrucción `return`

Esta instrucción sirve para devolver un valor dentro de una función. Su sintaxis es:

```
return expresion
```

Al ser una instrucción que solo tiene sentido dentro de funciones, esta instrucción solo puede aparecer en el cuerpo de una función.

4. Programa de ejemplo

El siguiente es un programa de ejemplo en Vectorinox.

```
define prod(x: mat, y:vec) of type num as
    return ((x[1, 1:] * 2) * (y . 3))[1]

begin
    vars
        c : mat;
        d : vec;
        e, i : num
    a := zeroes(3, 3);
    b := zeroes(, 3);
    i := 0
    while i = 2 do
    begin
        e := prod(c, d);
        write "El resultado es", e
    end
end
```

5. Ejecución de un programa de Vectorinox

Vectorinox es un lenguaje interpretado que recibirá su entrada (un programa) mediante archivos con extension .vec. El intérprete, llamado `vecti`, se ejecuta de la siguiente forma:

```
$ vecti nombre-archivo
```

También se le podrá facilitar un programa entero usando la opción de consola e.

```
$ vecti -e programa
```

Por ejemplo:

```
$ vecti -e "print 'Hola mundo!'"
```