UNIVERSIDAD Rafael Belloso Chacín

Revista Electrónica de Estudios Telemáticos

TÉLÉMATIQUE

NASPI: UNA NOTACIÓN ALGORÍTMICA ESTÁNDAR PARA PROGRAMACIÓN IMPERATIVA

(NASPI: An Algorithmic Standard Notation for Imperative Programming)

Martínez Morales, Amadís Antonio*
Universidad de Carabobo - Valencia, Venezuela
aamartin@uc.edu.ve, aammorales@acm.org

Rosquete De Mora, Daniel Humberto**
Universidad de Carabobo - Valencia, Venezuela
dhrosquete@uc.edu.ve, dhrosquete@acm.org

RESUMEN

Un algoritmo consiste de una secuencia finita de instrucciones para resolver un tipo específico de problema. Dada la gran diversidad de notaciones para expresar un algoritmo, que incluso pueden variar de un programador a otro, no existe un consenso con respecto a la representación de un algoritmo. Por lo tanto, es frecuente tener las mismas estructuras algorítmicas, pero con una notación diferente. Esta situación contradice la característica de definibilidad de un algoritmo; que establece que cada instrucción debe definirse de modo preciso y sin ambigüedades. En este artículo se presenta NASPI, una propuesta de notación algorítmica estándar para la enseñanza del paradigma de programación imperativa, así como también mejorar la comunicación entre los participantes de proyectos de desarrollo de software. NASPI está basado en los conceptos y constructores provistos por la mayoría de los lenguajes imperativos, para facilitar el proceso de programación de los algoritmos expresados utilizando la notación propuesta.

Palabras clave: Algoritmos, Programación Imperativa, Pseudocódigo.

ABSTRACT

An algorithm consists of a finite sequence of instructions oriented to solve a specific kind of problem. Given the wide diversity of notations to express an algorithm, which can even vary from one programmer to another, there not exists a consensus with respect to the representation of an algorithm. Thus, it is frequent to have the same algorithmic structures, but using different notations. This situation contradicts the definability of an algorithm, which establishes that each instruction must be defined in a precise and unambiguous way. In this paper we present NASPI, a proposal of algorithmic standard notation for teaching of the imperative programming paradigm and to improve the communication between the participants of development software projects. NASPI is based on the concepts and constructors provided by most of the imperative languages, to facilitate the programming process of the algorithms expressed using the proposed notation.

Key words: Algorithms, Imperative Programming, Pseudocode.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

*Ingeniero en Computación (Universidad Simón Bolívar-USB). Magíster en Ciencias de la Computación (USB). Candidato a Doctor en Computación (USB). Profesor Asociado del Departamento de Computación de la FaCyT-UC. Áreas de interés: Teoría de Algoritmos, Algoritmia, Optimización Combinatoria, Grafos e Hipergrafos, Computación de Alto Rendimiento.

**Licenciado en Computación (Universidad de Carabobo-UC). Cursando actualmente el Programa de Maestría en Matemática y Computación (UC). Instructor del Departamento de Computación de la FaCyT-UC. Áreas de interés: Teoría de Algoritmos, Algoritmia, Optimización, Criptografía y Seguridad de la Información, Computación de Alto Rendimiento.

Introducción

Ciencia de la Computación puede definirse como el estudio sistemático de los procesos algorítmicos que describen y transforman la información: su teoría, análisis, diseño, eficiencia, implementación y aplicación (Denning et al., 1989). En esta definición, la noción de algoritmo se percibe como un concepto central en Computación. Un algoritmo es una secuencia finita de instrucciones para resolver un tipo específico de problema. Las características fundamentales de un algoritmo son finitud, definibilidad, entrada, salida y efectividad (Knuth, 1997).

Dada la gran variedad de notaciones para expresar un algoritmo (lenguaje natural, diagramas de flujo, pseudocódigo y lenguajes de programación, entre otras), la característica de definibilidad es la más difícil de mantener. Por ejemplo, en el caso del pseudocódigo, éste puede variar de un programador a otro, lo cual pudiese llevar a diferencias sintácticas y semánticas que, posteriormente, pueden producir errores de lectura e interpretación.

Esta situación se ve con frecuencia en el caso de asignaturas para la enseñanza del paradigma de programación imperativa: debido a que no existe un pseudocódigo estándar, es frecuente tener las mismas estructuras algorítmicas con una notación diferente. Esto implica el aprendizaje de distintas versiones de pseudocódigo, lo cual, además de carecer de sentido en algoritmia, desvía a los estudiantes de la asignatura del objetivo principal de la misma.

Algunos ejemplos de pseudocódigo son los siguientes: GCL (Dijkstra, 1975), Súper Pascal (Aho et al., 1983), Lenguaje Algorítmico (Castro et al., 1993), Lenguaje Pseudoformal (Coto, 2002), Pseudolenguaje Algorítmico (Meza & Ortega, 2006), y UPSAM (Joyanes, 2008).

Guarded Command Language (GCL) es un lenguaje simplificado definido por Edsger Dijkstra, especialmente diseñado para facilitar la experimentación de técnicas de verificación formal: prueba de correctitud de programas, pre y postcondiciones, uso de invariantes, entre otros conceptos (Dijkstra, 1975). Lamentablemente, es un lenguaje de interés puramente teórico, adecuado para cursos avanzados de programación, pero poco conveniente para cursos introductorios o intermedios de programación.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Super Pascal (Aho et al., 1983) es una extensión del lenguaje de programación Pascal, realizada para aumentar la legibilidad del lenguaje. Sin embargo, esta notación algorítmica tiene las mismas desventajas que el lenguaje de programación Pascal, con el que está basado (Kernighan, 1981). Esto limita su uso como herramienta para la enseñanza del paradigma de programación imperativa.

Lenguaje Algorítmico (Castro et al., 1993) está basado en el lenguaje GCL el cual, como se indicó anteriormente, no está orientado a la enseñanza del paradigma de programación imperativa sino a la experimentación de técnicas de verificación formal.

Pseudolenguaje Algorítmico (Meza & Ortega, 2006) también está basado en el lenguaje de programación Pascal, como un subconjunto del mismo. Al igual que Super Pascal, tiene las mismas desventajas que Pascal, y no cuenta con el soporte para las nuevas tecnologías en programación que han surgido en los últimos años. Esto limita su uso como herramienta para la enseñanza del paradigma de programación imperativa.

Lenguaje Pseudoformal (Coto, 2002) y UPSAM (Joyanes, 2008) son notaciones propuestas para enseñar algoritmia en los primeros cursos de programación. Si bien son lenguajes adecuados para la enseñanza, no han tenido suficiente difusión y divulgación en cuanto a su uso como herramienta para la enseñanza del paradigma de programación imperativa.

En este trabajo se presenta NASPI, una propuesta de notación algorítmica estándar para la enseñanza del paradigma de programación imperativa, y para mejorar la comunicación entre los participantes de proyectos de desarrollo de software.

De esta manera, se aborda la problemática explicada anteriormente y se refuerza la característica de definibilidad que debe tener todo algoritmo. NASPI está basado en los conceptos y constructores provistos por la mayoría de los lenguajes imperativos (Sethi, 1996), y permite la especificación de algoritmos independientemente del lenguaje en el que vayan a ser implementados, evitando de esta manera los vicios inherentes al uso de un lenguaje en concreto.

NASPI está fundamentado en Lenguaje Pseudoformal (Coto, 2002) y UPSAM (Joyanes, 2008), extendiéndolos para representar conceptos de programación tanto básicos como más avanzados.

NASPI es un lenguaje pseudoformal que ofrece: (1) una estructura básica de algoritmos, (2) los tipos de datos elementales y estructurados más comunes, (3) declaraciones de constantes y variables asociadas a cualquier tipo de dato, (4) operaciones de entrada/salida básicas, (5) procedimientos y funciones parametrizadas, (6) los tipos de paso de parámetro básicos, y (7) la definición de tipos de datos por parte del usuario.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Este artículo fue estructurado en cuatro secciones, incluyendo la introducción. En la Sección 2 se presenta el modelo de computación de máquinas de acceso directo, sobre el cual se definirá NASPI. La Sección 3 contiene la descripción de los conceptos y constructores de NASPI. Finalmente, la Sección 4 contiene las conclusiones y el trabajo futuro.

Modelo de computación

Un modelo de computación es una abstracción de un sistema de computación que define el conjunto de operaciones permitidas y sus costos computacionales. Los modelos de computación difieren entre sí en su poder de cómputo y en el costo de las operaciones.

El matemático John Von Neumann introdujo en 1945 el concepto de programa almacenado, mejor conocido como arquitectura de Von Neumann, del cual se deriva el modelo de computación de máquinas de acceso directo (Random Access Machines -RAM).

Una RAM es una simplificación de un computador "real" que tiene cuatro componentes: (1) una memoria de acceso directo que almacena la información y es accesible independientemente de su contenido, (2) un algoritmo, (3) un dispositivo de entrada, y (4) un dispositivo de salida. La Figura 1 muestra la relación que existe entre estos componentes.

La memoria consiste de una secuencia de direcciones, cada una de las cuales almacena la misma cantidad de información. El valor almacenado en una dirección de memoria es el contenido de esa dirección.

El algoritmo es una secuencia de instrucciones para asignación, control y entrada/salida. El dispositivo de entrada permite la lectura de datos provistos desde el exterior del modelo a través de la entrada estándar o de archivo(s) de entrada.

El dispositivo de salida permite la comunicación de datos hacia el exterior del modelo a través de la salida estándar o de archivo(s) de salida. (Sethi, 1996)

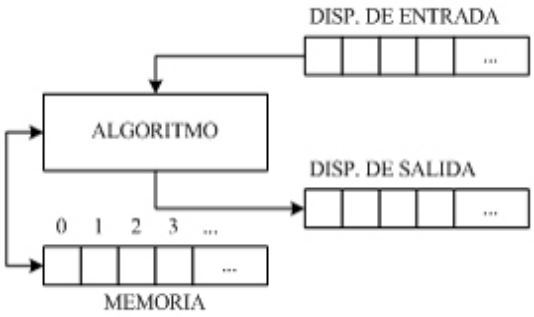

Figura 1. Modelo de computación de máquinas de acceso directo.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Sobre este modelo de computación de máquinas de acceso directo, se presentará nuestra propuesta de notación algorítmica estándar para la enseñanza del paradigma de programación imperativa (NASPI).

NAPSI: la notación algorítmica

Estructura de un algoritmo

La estructura general de un algoritmo en NASPI es la siguiente:

algoritmo nombre
[**const**
// declaraciones de constantes]
[**tipo**
// declaraciones de tipos]
[**var**
// declaraciones de variables globales]
inicio
// cuerpo del algoritmo
fin

Donde "nombre" es el identificador del algoritmo. Los corchetes ([,]) significan que el ítem es opcional. Las palabras reservadas se colocarán en negritas. Existen dos tipos de comentarios.

Para comentarios de una sola línea, se utilizará la doble barra inclinada (//). Este símbolo servirá para ignorar todo lo que aparezca hasta el final de la línea.

Los comentarios de varias líneas se delimitarán utilizando llaves ({, }), que indicarán el inicio y el final del comentario, respectivamente. Todo lo que se encuentre incluido entre estos dos símbolos será ignorado.

Variables y tipos de datos

Una variable representa un espacio de memoria reservado para almacenar un valor que corresponde con un tipo de dato (tipo). Una variable se denota por medio de un identificador único (nombre). (Sethi, 1996)

Dada una variable v, su único tipo T define el dominio (conjunto de valores) que v puede tomar y determina las operaciones que se pueden realizar sobre v. Para denotar que una variable v es de tipo T se escribe v: T. Cada tipo tiene también un identificador único que lo denota, usualmente definido por el dominio correspondiente.

Los tipos de datos se clasifican en dos grupos: elementales (primitivos) y compuestos (estructurados). Los tipos elementales se suponen conocidos y

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

disponibles para utilizar; los tipos compuestos se construyen a partir de otros tipos mediante ciertas reglas.

Una o varias variables de un tipo T, con identificadores v_1, v_2,…, v_n, se declaran de la forma var v_1, v_2,…, v_n: T. También es posible realizar declaraciones en grupo, de la forma:

var

$v_{11}, v_{12},\ldots, v_{1n1} : T_1$

⋮

$v_{k1}, v_{k2},\ldots, v_{knk} : T_k$

El tipo de las variables declaradas puede ser elemental o compuesto. La instrucción fundamental es la asignación, que consiste en asociar un valor con una variable. Su sintaxis es la siguiente: v ← E, donde v es una variable y E es una expresión del mismo tipo que v. Operacionalmente, primero se evalúa E y luego se copia el valor obtenido en v.

Tipos de datos elementales

Los tipos elementales o primitivos se consideran conocidos a priori. En esta sub-sección se definen los tipos elementales de NASPI (entero, real, lógico y símbolo), incluyendo las características principales de cada uno de ellos.

Tipo ordinal

Un tipo elemental T es ordinal sólo si sus elementos están organizados de manera tal que: (1) existen dos elementos notables en T: el primero y el último, (2) para cada elemento en T, excepto el último; existe un elemento único que le sigue llamado sucesor (suc), (3) para cada elemento en T, excepto el primero; existe un elemento único que le precede llamado predecesor (pred), y (4) cada elemento de T tiene un valor entero asociado con él, llamado número ordinal (ord), que representa la posición relativa del elemento en la secuencia de valores que define a T. (Wirth, 1973)

Todos los tipos elementales son ordinales, con excepción del tipo real que no satisface las tres últimas propiedades de la definición. En las Tablas 1 y 2 se presentan las operaciones y relaciones, respectivamente, definidas sobre elementos de tipo ordinal.

Operador	Significado
ord	número ordinal
pred	Predecesor
suc	sucesor

Tabla 1. Operaciones definidas sobre un tipo ordinal.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Operador	Significado
=	Igual a
<	Menor que
>	Mayor que
≤	Menor o igual que
≥	Mayor o igual que
≠	Distinto de

Tabla 2. Relaciones definidas sobre un tipo ordinal.

Tipo entero

Este tipo es un subconjunto finito del conjunto de los números enteros $\mathbb{Z}$. Se utiliza la palabra reservada **entero** en su declaración. En la Tabla 3 se presentan las operaciones definidas sobre elementos de tipo entero.

Operador	Significado
-	Menos unario
–	Resta
+	Suma
*	Multiplicación
div	División entera
mod	Resto de la división entera
**	Exponenciación

Tabla 3. Operaciones definidas sobre el tipo entero.

Un ejemplo de declaración de variables de tipo **entero** es el siguiente:

var
Horas, minutos, segundos: **entero**

Tipo real

Este tipo es un subconjunto finito del conjunto de los números reales $\mathbb{R}$. Se utiliza la palabra reservada **real** en su declaración. En la Tabla 4 se presentan las operaciones definidas sobre elementos de tipo real.

Operador	Significado
-	Menos unario
–	Resta
+	Suma
*	Multiplicación
/	División real

Tabla 4. Operaciones definidas sobre el tipo real.

Un ejemplo de declaración de variables de tipo **real** viene dado por:

var
Impuesto, porcentaje: **real**

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Tipo lógico

Este tipo de dato comprende dos valores de verdad que están representados por los identificadores **verdadero** y **falso**. Se utiliza la palabra reservada **booleano** en su declaración. En la Tabla 5 se presentan las operaciones definidas sobre elementos de tipo lógico.

Operador	Significado
¬	Negación lógica
∧	Multiplicación lógica
∨	Suma lógica

Tabla 5. Operaciones definidas sobre el tipo lógico

Un ejemplo de declaración de variables de tipo **booleano** es el siguiente:

var
Existe, encontrado: **booleano**

Tipo símbolo

Este tipo es un conjunto tal que contiene las 26 letras mayúsculas del alfabeto latino, las 26 letras minúsculas, los 10 dígitos de la numeración árabe y otros símbolos especiales. Se utiliza la palabra reservada **símbolo** en su declaración.

Un ejemplo de declaración de variables de tipo **símbolo** viene dado por:

var
Sexo, tipo de cliente: **símbolo**

Precedencia de operadores

Los operadores son asociativos a la izquierda. Los operadores binarios utilizan notación infija. La precedencia entre los operadores está definida en la Tabla 6, la cual se encuentra ordenada descendentemente, siendo los operadores primarios los de mayor precedencia.

Tipo	Operador
Primarios	( ) [ ] Paréntesis en expresiones o en llamadas a procedimientos o funciones. Corchetes en índices de arreglos.
Unarios	-, +, ¬
Multiplicativos	∗, /, div, mod, ∧, ∗∗
Aditivos	+, −, ∨
De cadena	+
De relación	=, <, >, ≤, ≥, ≠

Tabla 6. Precedencia entre operadores.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Un ejemplo para ilustrar la precedencia de operadores sería:

$A * B * C / D - 2 \neq (A * B * C) / (D - 2)$

Tipos de datos compuestos

Los tipos compuestos o estructurados se definen a partir de tipos primitivos y/o de otros ya construidos. La declaración de un tipo compuesto corresponde con su definición, y tiene la forma tipo T= def, donde T es el identificador del tipo que se declara, y def indica la manera de construcción del nuevo tipo.

De manera análoga, como se hizo con las variables, también es posible realizar declaraciones de tipos compuestos en grupo, de acuerdo con la siguiente sintaxis:

tipo
$T_1 = def_1$
⋮
$T_k = def_k$

En esta sección se presentan los tipos compuestos de NASPI, incluyendo las características principales de cada uno de ellos.

Tipo cadena

Una cadena es una secuencia finita de símbolos con longitud variable. Se utiliza la palabra reservada **cadena** en su declaración. Las cadenas son el único tipo de dato compuesto que no requiere declaración. En la Tabla 7 se presentan las operaciones definidas sobre elementos de tipo cadena.

Operador	Significado
+	Concatenación de cadenas
long	Longitud de la cadena

Tabla 7. Operaciones definidas sobre el tipo cadena.

Un ejemplo de declaración de variables de tipo **cadena** es el siguiente:

var
Nombre, apellido: **cadena**

Tipo enumerado

Un enumerado es una secuencia finita y ordenada de valores referenciados por identificadores. Este tipo está definido por la enumeración explícita de los identificadores que lo conforman. La declaración de un tipo enumerado es de la forma $T = (id_1,\ldots, id_k)$, donde: (1) T es el identificador del nuevo tipo, (2) los id_i $(1 \leq i \leq k)$ son

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

los identificadores que forman el tipo, y (3) el orden de los identificadores de T está definido por la regla: $\langle \forall i \ \forall j : 1 \leq i, j \leq k : (id_i < id_j) \Rightarrow (i < j) \rangle$.

El primer elemento de un tipo enumerado tiene asociado el número ordinal 0. Una variable *v* de tipo enumerado *T* sólo puede tomar valores de la lista de identificadores que define a *T*. Los únicos operadores asociados con el tipo enumerado son el de asignación y los de comparación. Un identificador no puede pertenecer a más de un tipo enumerado.

Un ejemplo de declaración de variables de tipo enumerado viene dado por:

tipo
Días = (lunes, martes, miércoles, jueves, viernes)

Tipo intervalo

La declaración de un tipo intervalo es de la forma T= min .. max, donde: (1) T es el identificador del nuevo tipo, (2) min y max son constantes del mismo tipo (**entero**, **símbolo** o tipo enumerado) que especifican los límites inferior y superior del intervalo, y (3) la definición de un tipo intervalo es aceptable sólo si min ≤ max. Las operaciones que se pueden realizar sobre una variable de tipo intervalo son iguales que las del tipo a partir del cual está definido.

Un ejemplo de declaración de variables de tipo intervalo es el siguiente:

tipo
Horas = 0..23
Tipo arreglo

La declaración de un tipo arreglo es de la forma T= **arreglo** [dim_1,…, dim_k] **de** T_0, donde: (1) T es el identificador del nuevo tipo, (2) dim_i ($1 \leq i \leq k$) es un intervalo de tipo ordinal que especifica, para cada dimensión, los índices del primer y del último elemento del arreglo, y (3) T_0 es el identificador de cualquier tipo (elemental o definido por el usuario) que especifica el tipo de todos los elementos del arreglo. El acceso a un elemento de un arreglo se realizará colocando su índice entre corchetes.

Un ejemplo para ilustrar el tipo de dato arreglo sería:

tipo
Num_primos = **arreglo** [1..500] **de entero**

Tipo tupla

Los componentes de una tupla se denominan campos. Los nombres de los campos se conocen también como identificadores de campo o selectores. La declaración de un tipo tupla es de la forma:

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

tipo T= **tupla**
$campo_1$: T_1
⋮
$campo_k$: T_k
ftupla

Donde: (1) T es el identificador del nuevo tipo, (2) $campo_i$ ($1 \leq i \leq k$) es un identificador que se utilizará para referenciar el i-ésimo campo de la tupla; y (3) T_i ($1 \leq i \leq k$) es el tipo (elemental o definido por el usuario) del i-ésimo campo. El acceso al i-ésimo campo de una variable v de tipo tupla se realizará utilizando el punto (.), como en $v.campo_i$.

Un ejemplo para ilustrar el tipo de dato tupla sería:

tipo
Votante = **tupla**
Cédula, teléfono: **entero**
Nombre, dirección: **cadena**
ftupla

Tipo apuntador

Un apuntador es una variable que contiene la dirección de memoria de otra variable de un tipo T_0. La declaración de un tipo apuntador es de la forma ap_T= **apuntador a** T_0, donde ap_T es el identificador del nuevo tipo y T_0 es el tipo referenciado por ap_T.

El tipo referenciado puede ser elemental o compuesto. El apuntador nulo se denota por el valor constante **nulo**. Dada una variable P de tipo apuntador que hace referencia a un tipo T_0, las operaciones permitidas son las siguientes:

- **Referenciación (β):** Si *Y* es una variable de tipo T_0, entonces $P \leftarrow \beta Y$ asigna a P la dirección de memoria de Y.

- **Desreferenciación (↑):** P↑ retorna el contenido de la dirección de memoria almacenada en P.

- **Asignación de memoria:** la ejecución de la instrucción **crear** (P): (1) crea una nueva variable de tipo T_0, (2) accede el valor de la dirección de memoria de la variable creada en el ítem anterior, y (3) asigna este valor a la variable P.

- **Liberación de memoria:** la ejecución de la instrucción **liberar** (P): (1) elimina P↑, y (2) el contenido de P se vuelve indefinido. Pre-condiciones: P↑ fue creado a partir de la instrucción **crear** (P) y *P* todavía hace referencia a este valor.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Equivalencias de tipos

Existen tres posibles relaciones entre dos tipos de datos T_1 y T_2: igualdad, compatibilidad y compatibilidad de asignación.

Igualdad: Dos tipos T_1 y T_2 son iguales si por lo menos una de las siguientes afirmaciones es verdadera: (1) T_1 y T_2 tienen el mismo identificador, o (2) T_1 y T_2 tienen identificadores diferentes y **tipo** $T_1 = T_2$ es una declaración de tipos.

Compatibilidad: Dos tipos T_1 y T_2 son compatibles si por lo menos una de las siguientes afirmaciones es verdadera: (1) T_1 y T_2 son iguales, (2) T_1 es un intervalo de T_2 o T_2 es un intervalo de T_1, o (3) T_1 y T_2 son intervalos del mismo tipo.

Compatibilidad de asignación: Dos tipos T_1 y T_2 son de asignación compatible si por lo menos una de las siguientes afirmaciones es verdadera: (1) T_1 y T_2 son compatibles o (2) uno de ellos es tipo **real** y el otro es tipo **entero** o un intervalo de tipo **entero**.

Constantes

Una constante puede verse como una variable cuyo valor no cambia durante su tiempo de vida. Una constante es representada y utilizada a través de un identificador único (nombre) que la denota. Una constante *c* con valor V se declara de la forma **const** c ← V. El tipo T de una constante *c* siempre es elemental, y viene dado por el dominio sobre el cual esté definido el valor V utilizado en la declaración.

Así como en el caso de las variables, también es posible declarar varias constantes simultáneamente, de la siguiente forma:

const
$c_1 \leftarrow V_1$
⋮
$c_k \leftarrow V_k$

Un ejemplo para ilustrar las constantes sería:

const
PI ← 3.1415926

Estructuras de control

Las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un algoritmo, de acuerdo con ciertas condiciones.

Una condición es una expresión de tipo lógico que puede ser: (1) una constante de tipo **booleano**, (2) ¬*e*, donde *e* es una expresión de tipo lógico, o (3) e_1 op e_2, donde

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

e_1, e_2 son expresiones del mismo tipo y *op* es un operador lógico binario (Tabla 5) o de comparación (Tabla 2).

Secuenciación

Si S_1, …, S_k (k > 0) son instrucciones, entonces su composición secuencial es una lista de instrucciones de la forma:

S_1
⋮
S_k

Operacionalmente, primero se ejecuta S_1, después S_2, y así sucesivamente hasta la última instrucción S_k.

Condicional

Simple: Sean cond, una expresión de tipo lógico, y S una secuencia de instrucciones. Un condicional simple es de la forma **si** (cond) **entonces** S **fsi**. Si cond es verdadera, entonces se ejecutan las instrucciones de S. Un ejemplo de instrucción condicional simple es el siguiente:

si (nota ≥ 10) **entonces**
aprobados ← aprobados + 1
fsi

Doble: Sean cond, una expresión de tipo lógico, y S_1, S_2 secuencias de instrucciones. Un condicional doble es de la forma:

si (cond) **entonces** S_1 **sino** S_2 **fsi**

Si cond es verdadera, se ejecutan las instrucciones de S_1; en caso contrario, se ejecutan las instrucciones de S_2. Un ejemplo de instrucción condicional doble viene dado por:

si (a = b) **entonces**
a ← b
sino
b ← a
fsi

Múltiple: Sean expr, una expresión de tipo ordinal, y S_1, …, S_k (k > 0) secuencias de instrucciones. Un condicional múltiple es de la forma:

selección (expr) **de**
lst_1 : S_1

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

⋮
lst_{k-1} : S_{k-1}
[**sino**
S_k]
fselección

Donde lst_i ($1 \leq i \leq k$) consistirá de uno o más valores, separados por comas, del mismo tipo que expr. Si el valor de expr coincide con alguno de los valores de la primera lista de valores (lst_1), entonces se ejecutan las instrucciones correspondientes (S_1) y sale de la estructura.

En caso contrario, evalúa la siguiente lista de valores, y así sucesivamente. Las acciones de la cláusula **sino** sólo se ejecutarán si ningún valor de lst_i ($1 \leq i < k$) coincide con expr. Un ejemplo de instrucción condicional múltiple, donde a es una variable que puede contener valores de 1 a 3, es el siguiente:

selección (a) **de**
1 : result ← 4
2 : result ← 8
3 : result ← 12
sino
escribir("Caso no definido")
fselección

Iteración

Estructura mientras: Sean cond, una expresión de tipo lógico, y S una secuencia de instrucciones. Un ciclo **mientras** es de la forma:

mientras (cond) **hacer**
S
fmientras

La expresión cond y la secuencia de instrucciones S se evalúan alternativamente mientras cond sea verdadera. Cuando cond resulta falsa, el ciclo mientras finaliza. Para ejemplificar este tipo de estructura iterativa:

a ← 0
mientras (a < 8) **hacer**
a ← a + 1
fmientras

Estructura repetir: Sean cond una expresión de tipo lógico y S una secuencia de instrucciones. Un ciclo **repetir** es de la forma:

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

repetir
S
mientras (cond)

La secuencia de instrucciones S y la expresión cond se evalúan alternativamente mientras cond sea verdadera. Cuando cond resulta falsa, el ciclo repetir finaliza. Para ejemplificar este tipo de estructura iterativa:

a ← 10
repetir
a ← a − 1
mientras (a > 8)

Estructura para: Sea cont una variable de tipo entero. Dadas v_i, v_f y expr constantes de tipo entero, un ciclo **para** es de la forma:

para cont ← v_i **hasta** v_f **en** expr **hacer**
S
fpara

La secuencia de instrucciones S se evalúa para cada valor de cont comprendido entre v_i y v_f. expr especifica, de acuerdo con su signo, la cantidad en la cual se incrementa o decrementa el valor de la variable cont para la siguiente ejecución de S. Un ejemplo que realice un ciclo que incremente de 1 en 1 sería:

para i ← 1 **hasta** 8 **en** 1 **hacer**
prom ← prom + i
fpara

Un ejemplo que realice un ciclo que decremente de 2 en 2 sería:

para i ← 8 **hasta** 0 **en** -2 **hacer**
prom ← prom + i
fpara

Entrada / salida

Son instrucciones que permiten la comunicación de datos desde y hacia los dispositivos de entrada/salida estándar. **Leer** (v_1, …, v_k), lee una o más variables desde la entrada estándar. **Escribir** ($expr_1$, …, $expr_k$), escribe una o más expresiones en la salida estándar, donde se coloca entre comillas dobles el mensaje y fuera de las comillas, separado por comas, las variables. Asumiendo que i y j son variables de tipo **entero**, un ejemplo para esta sección viene dado por:

leer (i, j)
escribir (“Valor de i: ”, i, “Valor de j: ”, j)

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Funciones y procedimientos

El ámbito de las variables declaradas dentro de un módulo (función o procedimiento) es local, y el tiempo de vida de dicha variable será el tiempo de ejecución de ese módulo.

La declaración de una función tiene la siguiente sintaxis:

func nombre([F_1; …; F_k]) : T
[declaraciones locales]
inicio
⋮
retornar(expr)
ffunc

La declaración de un procedimiento tiene la siguiente sintaxis:

proc nombre([F_1; …; F_k])
[declaraciones locales]
inicio
⋮
fproc

Donde:

- nombre es el identificador de la función o procedimiento.
- Cada F_i ($1 \leq i \leq k$) es un grupo de parámetros formales de la siguiente forma: {**val** | **ref**} p_1, …, p_m : T_i, donde:
    - **val** o **ref** indican si el paso de parámetros se realiza por valor o por referencia, respectivamente.
    - p_j ($1 \leq j \leq m$) es el identificador del j-ésimo parámetro formal del i-ésimo grupo.
    - T_i es el tipo (elemental o definido por el usuario) de los parámetros que forman el i-ésimo grupo.

Además, en el caso de una función:

- T es el tipo (elemental o definido por el usuario) que retorna la función.
- expr es el valor de retorno de la función. El tipo de expr y T deben ser iguales.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

La llamada a una función es de la forma: nombre([A_1, …, A_k]); la llamada a un procedimiento se realiza de la siguiente manera: [**llamar a**] nombre([A_1, …, A_k]). En ambos casos, los parámetros actuales en A_i deben coincidir en número, orden y tipo con los parámetros formales de la declaración F_i ($1 \leq i \leq k$). En el caso de las funciones, como retornan un valor, la llamada siempre debe hacerse dentro de una expresión.

Un ejemplo de función es la definición recursiva del cálculo del factorial de un número entero positivo n. La declaración de la función correspondiente es la siguiente:

```
func factorial(val n : entero) : entero
inicio
si (n < 2) entonces
retornar(1)
sino
retornar(n ∗ factorial(n − 1))
fsi
ffunc
```

Un ejemplo de procedimiento es la carga secuencial de datos por filas en una matriz M. La declaración del procedimiento correspondiente es la siguiente:

```
proc carga(ref M : arreglo [1..N, 1..N] de entero)
var
i, j, k : entero
inicio
k ← 1
para i ← 1 hasta N en 1 hacer
para j ← 1 hasta N en 1 hacer
M[ i ][ j ] ← k
k ← k + 1
fpara
fpara
fproc
```

Archivos

Un archivo es un conjunto de unidades de información, denominadas registros, que están dispuestas sobre un soporte físico de almacenamiento permanente de información (discos magnéticos u ópticos) con una determinada organización lógica.

Todo archivo tiene asociado un identificador (nombre) que contiene la información necesaria para poder realizar correctamente todas las operaciones sobre el archivo. De esta manera, en todos los algoritmos que utilicen archivos será necesario declarar su identificador de acuerdo con el tipo de archivo a utilizar. Los archivos se clasifican,

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

según el tipo de acceso a los mismos, en dos grupos: archivos secuenciales y archivos directos.

Archivos de acceso secuencial

En un archivo secuencial, para acceder a un registro, es necesario recorrer todos los registros que le preceden. La declaración de una variable de tipo archivo secuencial es de la forma f: **archivo_s de** T_0, donde: (1) f es el identificador de la variable y (2) T_0 es el identificador de cualquier tipo elemental o definido por el usuario. Las operaciones permitidas son las siguientes:

Apertura: abrir(f, modo, nombre), donde f es una variable de tipo archivo secuencial, modo indica el tipo de operación que se realizará sobre el archivo, y nombre es una expresión de tipo cadena con el calificativo que se dará al archivo. Los valores posibles para modo son los siguientes:

- "l": lectura al comienzo del archivo. El archivo debe existir previamente.
- "e": escritura al comienzo del archivo. Si el archivo no existe, primero crea un archivo vacío. Si el archivo existe, sobrescribe los datos que tenga.
- "a": escritura al final del archivo. Si el archivo no existe, primero crea un archivo vacío.

Lectura: leer(f, v), lee en la variable v el siguiente registro del archivo abierto para lectura representado por f. El tipo de v debe coincidir con el tipo base del archivo definido en la declaración de tipo.

Escritura: escribir(f, expr), escribe el valor de la expresión expr en el archivo abierto para escritura y representado por f. El tipo de expr debe coincidir con el tipo base del archivo definido en la declaración de tipo.

Archivos de acceso directo

En un archivo directo cualquier registro es directamente accesible por medio de un índice que especifica la posición del registro con respecto al origen del archivo. La declaración de una variable de tipo archivo directo es de la forma f: **archivo_d de** T_0, donde: (1) f es el identificador de la variable y (2) T_0 es el identificador de cualquier tipo elemental o definido por el usuario. Las operaciones permitidas son las siguientes:

Apertura: abrir(f, modo, nombre), donde f es una variable de tipo archivo directo, modo indica el tipo de operación que se realizará con el archivo, y nombre es una expresión de tipo cadena con el nombre que se dará al archivo. Los valores posibles para modo son los siguientes:

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

- "ld": lectura al comienzo del archivo. El archivo debe existir previamente.

- "ed": escritura al comienzo del archivo. Si el archivo no existe, primero crea un archivo vacío. Si el archivo existe, sobrescribe los datos que tenga.

- "ad": lectura/escritura al comienzo del archivo. Si el archivo no existe, primero crea un archivo vacío.

Lectura: leer(f, pos, v), lee en la variable v el registro situado en la posición relativa pos del archivo abierto para lectura representado por f. El tipo de v debe coincidir con el tipo base del archivo definido en la declaración de tipo.

Escritura: escribir(f, pos, v), escribe el contenido de la variable v en la posición relativa pos del archivo abierto para escritura representado por f. El tipo de v debe coincidir con el tipo base del archivo definido en la declaración de tipo.

Operaciones comunes

Clausura: cerrar(f_1, …, f_k), cierra los k archivos abiertos previamente.

Detección del final del archivo: fda(f) retorna un valor booleano que permite detectar si se ha llegado o no a la marca de final del archivo representado por f, la cual se encuentra justo después del último registro del archivo.

Conclusiones y trabajo futuro

En este artículo se presentó NASPI, una propuesta de notación algorítmica estándar para la enseñanza del paradigma de programación imperativa, así como también para mejorar la comunicación entre los participantes de proyectos de desarrollo de software.

NASPI contiene la mayoría de los conceptos y constructores provistos por los lenguajes imperativos (Sethi, 1996), por lo que la traducción a cualquiera de estos lenguajes sería una tarea relativamente sencilla.

En los momentos actuales, los autores se encuentran desarrollando una plantilla, basada en el paquete algorithm2e de LaTeX, para la escritura de algoritmos en este ambiente usando NASPI. Como trabajo futuro se propone extender NASPI para proporcionar soporte al paradigma de programación orientada a objetos.

Referencias Bibliográficas

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). **Data Structures and Algorithms.** Reading, MA, USA: Addison-Wesley.

Castro, J., Cucker, F., Messeguer, X., Rubio, A., Solano, L., & Valles, B. (1993). **Curso de Programación.** Madrid, España: McGraw-Hill Interamerica S.A.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009

Coto, E. (2002). **Lenguaje Pseudoformal para la Construcción de Algoritmos**. (Tech. Rep. ND 2002-08). Universidad Central de Venezuela.

Denning, P. J., Comer, D., Gries, D., Mulder, M. C., Tucker, A. B., Turner, A. J., et al. (1989). **Computing as a Discipline.** Communications of the ACM, 32(1), 9–23.

Dijkstra, E. W. (1975). **Guarded Commands, Non-determinacy and Formal Derivation of Programs.** Communications of the ACM, 18(8), 453–457.

Joyanes, L. (2008). **Fundamentos de Programación** (Cuarta ed.). Madrid, España: McGraw-Hill Interamerica S.A.

Kernighan, B. W. (1981). **Why Pascal is Not My Favorite Programming Language** (Computing Science Technical Report No. 100). AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

Knuth, D. E. (1997). **The Art of Computer Programming: Fundamental Algorithms** (Third ed., Vol. 1). Redwood City, CA, USA: Addison-Wesley.

Meza, O., & Ortega, M. (2006). **Grafos y Algoritmos (Segunda ed.).** Universidad Simón Bolívar: Editorial Equinoccio.

Sethi, R. (1996). **Programming Languages: Concepts and Constructs** (Second ed.). Boston, MA, USA: Addison-Wesley.

Wirth, N. (1973). **Systematic Programming: An Introduction**. Englewood Cliffs, NJ, USA: Prentice Hall PTR.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 8 Edición No. 3 – Año 2009