Alpha: Una notación algorítmica basada en pseudocódigo

(*Alpha: An algorithm notation based on pseudocode*)

Ramírez, Esmitt
Centro de Computación Gráfica, Universidad Central de Venezuela
esmitt.ramirez@ciens.ucv.ve

RESUMEN

En las Ciencias de la Computación, un algoritmo es un conjunto ordenado de instrucciones que permiten realizar una tarea mediante pasos sucesivos, tomando datos de entrada sobre un estado inicial para arrojar una salida. Así, un algoritmo puede ser expresado de muchas maneras: usando lenguaje natural, algún lenguaje de programación, diagramas de flujo, pseudocódigo, entre otros. Particularmente, el pseudocódigo ofrece una descripción de alto nivel de un algoritmo mezclando el lenguaje natural con sintaxis de uno o muchos lenguajes de programación. El pseudocódigo, es ampliamente empleado en diversos libros de texto, publicaciones científicas, cátedras universitarias, y como producto intermedio durante la fase de desarrollo de software. Sin embargo, el pseudocódigo no está basado en ningún estándar o notación, presentando una gran variación entre distintos grupos de investigación y desarrollo. En este trabajo se propone una notación para la construcción de algoritmos y estructuras de datos que es simple, eficaz y moderna, permitiendo una rápida conversión entre el pseudocódigo y un lenguaje de programación. Entonces, se describe de forma detallada la sintaxis y convenciones empleadas en la notación propuesta ofreciendo una poderosa y útil herramienta con soporte a diversos enfoques de programación. Las pruebas realizadas demuestran su uso en la construcción de estructuras de datos complejas y al mismo tiempo verifican su impacto en un curso de Algoritmia.

Palabras clave: notación algorítmica, algoritmos, programación, pseudocódigo

ABSTRACT

In computer science, an algorithm is an ordered set of instructions that allow you to perform a task by consecutive steps, taking input data for an initial state to obtain an output. Thus, an algorithm can be expressed in different ways: using natural language, some programming language, flowcharts, pseudocode, and others. Particularly, the pseudocode offers a high-level description of an algorithm by mixing the natural language syntax with one or several programming languages. The pseudocode is widely used in various textbooks, scientific papers, university departments, and as an intermediate product during the software development process. However, the pseudocode is not standard-based or any notation offering a large disparity between different research and development groups. In this paper, we present a notation for algorithms and data structures development that are simple, effective and modern, allowing a quick conversion between the pseudocode and a programming language. Then, a detailed syntax and conventions used in the notation proposed is described, offering a powerful and useful tool to support different programming approaches. The performed tests show its use in the construction of complex data structures while verifying its impact in an Algorithms course.

Keywords: algorithm notation, algorithms, programming, pseudocode

1. INTRODUCCIÓN

En la escritura de algoritmos, es posible distinguir tres modelos de construcción y diseño de los éstos: seguir la sintaxis de un lenguaje de programación conocido como Java, PHP, Basic, etc.; emplear un lenguaje natural basado en sentencias lógicas correctas como "*asignar a la variable X el valor leído*"; o emplear un lenguaje híbrido producto de mezclar frases del lenguaje natural con sentencias correctas en uno o más lenguajes de programación (también llamado pseudocódigo).

Tomando el primer modelo, es posible que se dificulte separar la idea del algoritmo de los detalles de implementación. Un ejemplo notable de ello se observa en Sedgewick y Wayne (2011) donde emplean el lenguaje Java para la explicación de los algoritmos. Sin embargo, dichos autores tratan de mitigar dicho problema empleando solo un subconjunto del lenguaje Java y considerando que la sintaxis es muy similar en muchos lenguajes de programación modernos (e.g. C#, C++, F#, entre otros).

El segundo modelo es creado de forma empírica siguiendo un patrón lógico de desarrollo de software. Bajo este concepto, se han diseñado diversos lenguajes de programación tales como Hyper Talk, Lingo, AppleScript, SQL, Python, entre otros, donde la idea detrás de éstos reside en que cualquier persona sin conocimiento directo de un lenguaje en particular pueda entender el código escrito, y posteriormente aprenderlo.

El tercer modelo (junto al primero) es uno de los más empleados en libros de texto y artículos científicos y se conoce como pseudocódigo, pseudoformal o pseudolenguaje. Este se comporta como una descripción compacta, sin detalle de implementación y en ocasiones asociado a un estilo de lenguaje de programación (conocido como *pidgin code*: estilo C++, estilo Java, estilo Fortran, etc.). Del mismo modo, el pseudocódigo se emplea en las aulas de clases de diversos cursos de algoritmia a nivel de educación superior e universitaria.

Diversos autores reconocidos en el área de las Ciencias de la Computación escriben sus textos basados en un pseudocódigo para la explicación de sus algoritmos (Dasgupta, Papadimitriou, & Vazirani, 2008; Cormen, Leiserson, Rivest, & Stein, 2009; Skiena, 2010). Sin embargo, no existe una sintaxis estándar para el pseudocódigo, dado que no es directamente un programa ejecutable por un computador.

En este trabajo se presenta una propuesta de notación para la construcción de algoritmos basado en pseudocódigo que es simple, eficaz y permite una rápida conversión a algún lenguaje de programación. Dicha notación, denominada Alpha, soporta diversas los elementos básicos de un lenguaje de programación tales como variables, sentencias, estructuras iterativas, estructuras de control, tipos de datos, entre otros. La notación Alpha integra elementos fundamentales presentes en lenguajes de programación modernos siendo un instrumento poderoso para la construcción de algoritmos.

La organización de este artículo es como sigue: la sección 2 presenta una serie de trabajos relacionados previos a esta investigación que sirven de soporte del estudio realizado. La sección 3 describe en detalle la notación Alpha, incluyendo la sintaxis y operaciones básicas así como las estructuras de datos y de control en pseudocódigo de la notación. Las pruebas realizadas y resultados obtenidos se muestran en la sección 4. Finalmente, se presentan las conclusiones y trabajos a futuro propuestos en la sección 5.

2. TRABAJOS PREVIOS

Según Zobel (2010), un programador que requiera implementar un algoritmo en específico, especialmente uno al cual no está familiarizado, primero debe comenzar con

una descripción en pseudocódigo y luego convertir esta descripción en el lenguaje destino. Igualmente, en grupos de trabajo de desarrollo con programadores, el intercambio de información e ideas se puede realizar empleando dicho pseudolenguaje para bosquejar y estructurar algoritmos.

Aunado a ello, existen alternativas gráficas al uso de pseudocódigo para la descripción de algoritmos que es empleado en diversos grupos interdisciplinarios, tales como diagramas de flujo (Sharp, 2008) ó diagramas UML (*Unified Modeling Language*) (Larman, 2004), las cuales siguen convenciones formales en su construcción.

En 1989, Bailey y Lundgaard proponen diversas estructuras formales que permiten separar la lógica de programación de un lenguaje de programación en particular. Entre estas estructuras definen un pseudocódigo que junto a los diagramas de flujo, diagramas IPO (*Input-Processing-Output*), gráficos de jerárquicos entre otros, permiten enriquecer la explicación de diferentes técnicas para la resolución de problemas computacionales. Posteriormente, diversos autores proponen la utilización del pseudocódigo para la construcción de software (McConnell, 2004; Gilberg, & Forouzan, 2004). De igual modo, se han propuesto notaciones formales basadas en lenguaje de programación matemática como *Z notation* (Spivey, 1992) o *Vienna Development Method Specification Language* - VDM-SL (Jones, 1990).

Gran parte de los pseudocódigos existentes son diseñados para la programación imperativa. Así, la influencia de lenguajes clásicos como BASIC, Fortran, Pascal y C se puede percibir de forma natural en la construcción de algoritmos de muchos esquemas de pseudocódigo. Actualmente, muchas de las estructuras de datos o instrucciones creadas han sido modificadas para adaptarse a los lenguajes modernos.

En el año 2002, Coto presenta una serie de especificaciones con el nombre de Lenguaje PseudoFormal, escrito en idioma castellano para la construcción de algoritmos. En dicho trabajo se presenta detalladamente una guía empleando pseudocódigo basado en C y Pascal. De este trabajo (Coto, 2002), surgieron otras propuestas igualmente en el idioma castellano para el diseño de programas computacionales, (Peña, 2005; Pes, 2006) para la enseñanza en el área de algoritmia.

Posteriormente Martínez y Rosquete (2009) crean una propuesta de notación algorítmica estándar para programación imperativa denominada NASPI la cual formaliza un pseudocódigo que es empleado durante la docencia de algoritmos y programación a nivel universitario. De este modo, se busca unificar las estructuras empleadas independientes del lenguaje de programación utilizado.

En el 2011 (Ottogalli, Martínez, & León, 2011) presentan NASPOO, una propuesta de notación algorítmica estándar para la enseñanza de programación orientada a objetos (POO) la cual complementa a la propuesta NASPI, ofreciendo estructuras de datos que soporten las características esenciales del la POO. Dicha investigación se basa en emplear NASPI (Martínez, & Rosquete, 2009), la notación empleada en el texto de (Joyanes, 2008) y Lenguaje PseudoFormal (Coto, 2002).

El objetivo de esta investigación es proponer una notación basada en pseudocódigo para la enseñanza de algoritmos y estructuras de datos a nivel universitario, extrayendo las características relevantes de las notaciones existentes, así como incluir características comunes de los lenguajes de programación modernos (e.g. Javascript, Go, Python, Ruby,

Lua, entre otros) que no estaban presentes en los lenguajes clásicos, y permitir una conversión rápida entre el pseudocódigo y algún lenguaje de programación.

3. NOTACIÓN ALPHA

En el 2004, McConnell propone unas reglas simples para la escritura de cualquier pseudocódigo:

- Emplear el idioma inglés

- No utilizar elementos sintácticos de algún lenguaje de programación específico

- Escribir el pseudocódigo para responder a “qué” y no “cómo” solucionar los aspectos de un problema de forma algorítmica

- Debe ser los suficientemente a bajo nivel para permitir que un desarrollador escriba su código basado en éste

Basado en estas reglas, se propone la notación Alpha para la construcción de algoritmos empleando pseudocódigo tal que permita separar la enseñanza de la lógica en la resolución de problemas algorítmicos de las características específicas de un lenguaje, considerando que el proceso de conversión pseudocódigo-lenguaje sea simple. Al mismo tiempo, Alpha define estructuras de datos esenciales que pueden ser empleadas en enfoques de programación orientada a eventos, orientada a objetos y estructurada. Es importante destacar que Alpha puede ser extendida para soportar programación funcional u otro tipo como programación declarativa.

La notación Alpha está diseñada para que una serie de instrucciones sea ejecutada línea a línea, es decir, no requiere de un punto de entrada de forma explícita (i.e. creación de una función principal - *main*). De esta forma, la explicación de los algoritmos se hace con mayor simplicidad y permite diseñar un algoritmo rápidamente.

3.1 Sintaxis básica

En la notación Alpha, los identificadores se consideran válidos si cumplen las mismas restricciones de muchos lenguajes tales como: no puede empezar con un número o símbolo, no contener espacios, no coincidir con alguna palabra reservada del pseudocódigo y estar compuesto únicamente por letras, números y el símbolo guión ‘-’ y piso ‘_’.

Para mayor legibilidad, en esta investigación se tomarán las siguientes convenciones:

- En las definiciones de tipo de datos, los símbolos que estén dentro de corchetes […] se consideran opcionales.

- En una sola línea de código, se definen variables de un solo tipo de dato.

- Para una mejor legibilidad de los algoritmos, en la medida de lo posible, las variables de un tipo comienzan con la(s) letra(s) de dicho tipo.

- Los identificadores de las variables de un tipo constante son en letras mayúsculas.

- El nombre de los procedimientos y funciones comienzan con letra mayúscula.

- Los atributos o miembros de las clases tienen como prefijo la cadena “m_” para indicar que pertenecen a una clase.

Así como en los lenguajes de programación, la notación Alpha contiene tipos de datos y las operaciones entre estos tipos. Un tipo de dato es un conjunto de valores y una serie de operaciones que pueden ser aplicadas a estos valores. Básicamente, los tipos de datos se pueden clasificar en simples (también llamados primitivas o elementales) y compuestos (también llamados estructurados). La diferencia radica en el hecho de dividirse en términos de los valores que lo constituyen tal que constituyan otro tipo de dato. En la tabla 1 se muestra un resumen de los tipos de datos simples empleados.

Tipo de Dato	**Conjunto de Valores**	**Operadores**
Char	Caracteres imprimibles (95) del código ASCII	Relacionales
Boolean	True y false	Relacionales de álgebra booleana
Integer	Números naturales del conjunto *S*, tal que $S \in \mathbb{Z}$	Aritméticos y relacionales
Real	Números naturales del conjunto *S*, tal que $S \in \mathbb{R}$	Aritméticos y relacionales
Enum	Subconjunto de valores enteros especificados por el intervalo [1,n] donde n es el número de identificadores del tipo Enum	Heredados de Integer
Pointer	Direcciones de memoria donde se almacenen variables	Referenciación, derreferenciación, suma y resta

Tabla 1. Tipos de datos simples de la notación Alpha. Fuente: Propia

Más adelante, en la sección 3.3, se discutirá sobre los tipos de datos compuestos con mayor detalle.

3.2 Operaciones básicas

Las operaciones básicas del pseudocódigo se definen como la declaración de variables, asignación de valores a variables, la lectura estándar y escritura estándar.

Declaración de variables

La declaración de una variable de nombre *<identificador>* del tipo de dato *<tipo>* se declara como:

<tipo> <identificador> [;]

Del mismo modo, es posible definir múltiples variables como una lista de identificadores en su declaración:

$<tipo> <identificador_1, identificador_2, ..., identificador_k> [;]$

Nótese que la presencia del símbolo *‘;’* al final de la declaración es opcional. Si se aplica dicho operador, se considera como de secuenciamiento, es decir, permite declarar más instrucciones en una misma línea. Se recomienda emplear una instrucción por línea solo para hacer más legible el pseudocódigo a escribir. Esta convención de secuenciamiento aplica de la misma forma para las instrucciones de un programa empleando el

pseudocódigo. En los ejemplos presentados en este artículo, no se empleará el uso del símbolo *';'* al final de cada instrucción si contiene una sola sentencia.

Asignación Simple

La asignación simple permite guardar un valor de su conjunto de valores en una variable declarada. Para la declaración de un *<valor>* a una variable *<identificador>* se realiza de las siguientes formas:

<identificador> = <valor> [;]

<identificador> = <identificador> [;]

Se emplea el operador = para la operación de asignación en la notación Alpha. La asignación no solamente permite darle un valor a las variables, sino también una expresión. Algunos ejemplos de declaración de variables de tipo de dato simple, se muestran en la figura 1.

```
Integer iMonth = 6
Char cGender = ' g '
Boolean bFlag = iMonth > 8
Real rTaxes = 12.5
Enum eDays = [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]
Integer * pPointer = ref iMonth
```

Figura 1. Ejemplo de declaración de variables simples. Fuente: Propia.

En el caso de las expresiones, se intenta evaluar de izquierda a derecha, mientras sea posible. Si existen diversos operadores, entonces se debe aplicar prioridad de operadores. La prioridad de operadores define un orden de ejecución de los operadores dentro de una expresión. Si en la expresión existen operadores con el mismo nivel de prioridad, se evalúa de izquierda a derecha.

La tabla 2 muestra un conjunto de operadores aritméticos, lógicos, relacionales y de asignación bajo un orden de prioridad (mayor prioridad de arriba a abajo y de izquierda a derecha para una expresión).

Tipo	**Operador**
Aritmético	*[ ] ( ) *(derreferenciado) ref* *-(unario) ^(potencia) * / div(división entera)* *mod(módulo)* *+ -*
Relacional	*> < ≥ ≤* *== !=*
Lógico	*not and or*
Asignación	*=*

Tabla 2. Prioridad de operadores a considerar en la notación. Fuente: Propia

Un ejemplo de la prioridad de operadores se puede observar en la figura 2.

```
Boolean bFlag = iMonth > 8
bValue = -4^2 < 8.05 + 6 * 3 and not 3 == 8
```

Figura 2. Ejemplo demostrativo de la prioridad de operadores. Fuente: Propia.

En el ejemplo, para obtener el valor de *bValue* primero se evalúa el símbolo - unario en el *4* y luego *-4^2* dando un valor de -16. Luego, se evalúa *6 * 3* y posterior se suma el valor de *8.05* resultando el valor de *26.05*. Después, se realiza la comparación *-16 < 26.05* dando el valor de *true*. La próxima evaluación es *3 == 8* resultando en *false*, y *not* de *false* es *true*. Finalmente, la evaluación final es *true and true* en la asignación de *bValue*.

El mecanismo para violar la prioridad existente es el uso de paréntesis *( )*.

Declaración de Constantes

Una constante *<identificador>* se define de acuerdo a su *<tipo>* y a su *<valorDeclarado>* de la siguiente forma:

const <tipo> <identificador> = <valorDeclarado> [;]

El *<valorDeclarado>* debe pertenecer al conjunto definido para el *<tipo>* de forma explícita.

Comentarios

La notación Alpha soporta la utilización de comentarios (i.e. información blablala). Los comentarios ocupan solo una línea y se identifican por el símbolo *//*. Todo texto que se encuentre después de dicho símbolo en una línea, se considera un comentario y no afecta la ejecución del programa en pseudocódigo. Su utilidad radica en documentar y etiquetar el código escrito para hacerlo más legible.

Entrada y Salida

La entrada o lectura simple permite obtener desde el dispositivo de entrada estándar definido (e.g. teclado) y asignarlo a una ó más variables del mismo tipo definidas como *<identificador$_i$>* previamente declaradas. Su forma sentencial es:

Read (<identificador$_1$>, <identificador$_2$>, ..., <identificador$_k$>) [;]

La salida o escritura simple permite mostrar los resultados de una expresión, identificador o valor por el dispositivo de salida estándar (e.g. monitor, impresora, etc.). Se emplea de las siguientes formas:

Print (<identificador-1>, <identificador-2>, ..., <identificador-k>) [;] //lista de identificadores

Print (<valor>) [;] //un solo valor

3.3 Tipo de Dato Compuesto

Un tipo de dato compuesto consiste en una colección de otros tipos de datos con el objetivo de construir algoritmos más complejos que solo con tipos elementales sería complicado. El tipo de dato compuesto se puede dividir en componentes y formar otro tipo de dato, cosa que no se puede hacer con el tipo de dato simple. La notación define el tipo de dato cadena (*String*), arreglo (*Array*), registro (*Register*) y archivo (*File*).

Tipo Cadena - String

El tipo *String* se puede definir como una secuencia de caracteres.

- Definición de tipo: *String <identificador> [;]*

- Conjunto de valores: Secuencia de valores del tipo *Char* sin límite definido

- Operadores: concatenación (+), relacionales (==,≠), de acceso (*[<pos>]*), y de construcción de substring (*[iLi .. iLs]*).

- El operador de acceso permite obtener un valor del tipo *Char* de un tipo *String*. El valor de *<pos>* corresponde a un valor del tipo Integer en el rango *[1, n]* donde *n* representa la longitud del *String*. En la figura 3 se muestra un ejemplo del uso del tipo *String*.

```
String sName = "Springfield" //contiene 11 posiciones
Integer iPos = 4
Char c = sName[iPos]  //es equivalente a sName[4]
Print (c)  //la salida es 'i'
String sSubName = sName[2..5]
Print (sSubName)  //la salida es "prin"
```

Figura 3. Utilización del tipo compuesto *String*. Fuente: Propia.

Tipo Arreglo - Array

Para el arreglo se realiza la definición del tipo de dato, la operación selectora que permite obtener un valor del arreglo y la operación constructora para inicializar los valores de un arreglo.

- Definición de tipo: *Array <identificador> of <tipo> [Li..Ls] [;]*

- Selector: *<identificador> [<pos>],* donde *pos* está en el rango *[Li,Ls]*

- Constructor: *Array <identificador> of <tipo> [ ] = {valor$_1$, valor$_2$,..., valor$_k$} [;]*

El tipo *Array* soporta construir estructuras que sean k-dimensionales empleando k-pares de *[Li$_k$ .. Ls$_k$]*. Por ejemplo, para *k=2* se define como:

Array <identificador> of <tipo> [Li$_1$ .. Ls$_1$][Li$_2$ .. Ls$_2$] [;]

donde *Li$_1$* y *Ls$_1$* definen los límites inferior y superior de la primera dimensión, y *Li$_2$* y *Ls$_2$* de la segunda dimensión.

Tipo Registro - Register

En la definición del tipo de dato *Register* se muestra la forma de su definición, el operador de acceso y el conjunto de valores.

- Definición de tipo:

Register <identificador>
<tipoDato$_1$> <identificador$_1$> [;]
<tipoDato$_2$> <identificador$_2$> [;]
...
<tipoDato$_k$> <identificador$_k$> [;]
end

- Selector: El operador '.' para tener acceso a un campo del registro. La asignación '=' seguida de todos los literales asociados a cada campo, separados por el símbolo ',' entre llaves *{ }*

- Conjunto de Valores: Los asociados a cada tipo de dato de los campos.

Un ejemplo de definición y utilización de un tipo *Register* que almacena los datos de una persona se muestra en la figura 4.

```
Register rgPerson
  String sName
  Integer iId
  Char cGender
  String sAddress
end
rgPerson.sName = "Homer Simpson"
rgPerson.iId = 911
rgPerson.cGender = 'M'
rgPerson.sAddress = "742 Evergreen Terrace"
rgPerson = {"Homer Simpson", 911, 'M', "742 Evergreen Terrace"}  //asignación como lista de valores
```

Figura 4. Ejemplo del uso de un tipo compuesto *Register*. Fuente: Propia.

Tipo Archivo - File

Un tipo de dato *File* representa a un archivo ó fichero. Este tipo de dato se construye como una clase, y sus variables definidas como objetos. La declaración de un archivo secuencial es:

File <objetoF> [;]

Las operaciones básicas de un tipo *File* son:

- Abrir Archivo: La definición para abrir un archivo en una variable *<identificadorF>* del tipo *File* es como sigue:

Boolean <identificador> = <objetoF>.OpenFile (<nombreArchivo>, <parámetros>) [;]

donde la función retorna *true* si fue exitosa la operación, *false* en caso contrario; *<objetoF>* es el nombre del objeto de la clase *File*; *<nombreArchivo>* es un tipo *String* que representa el nombre del archivo a abrir y, *<parámetros>* se refiere a la forma de operar con el archivo y puede tomar los siguientes valores:

- Create: Indica que se creará el archivo
- Write: Indica que el archivo se abre de solo escritura
- Read: Indica que el archivo se abre de solo lectura
- Text: Indica que el archivo a abrir es de formato texto
- Binary: Indica que el archivo a abrir es de formato binario
- Append: Indica que el archivo se abre de escritura pero se añade al final de archivo

La notación Alpha permite combinar el campo *<parámetros>* con el operador lógico *and*.

- Cerrar Archivo: Para cerrar o finalizar de operar sobre un archivo se hace ejecuta una función que retorna *true* si se logró exitosamente, y *false* en caso contrario:

Boolean <identificador>= <objetoF>.CloseFile ( ) [;]

- Fin de Archivo: La instrucción para indicar la finalización de archivo retorna *true* si no es posible obtener más datos en la lectura del archivo y *false* en caso contrario, y se define como:

Boolean <identificador> = <objetoF>.EndOfFile ( ) [;]

- Leer y Escribir del Archivo

Para la lectura del contenido del archivo se debe emplear una variable *<identificador>* que almacene los valores leídos. La definición es de la siguiente forma:

Boolean <identificador>= <objetoF>.ReadFile (<identificador>) [;]

De forma similar, la escritura en un archivo se realiza como:

Boolean <identificador>= <objetoF>.WriteFile (<identificador>) [;]

3.4 Tipo de Dato Definido

La notación Alpha soporta la definición de nuevos tipos de datos. Para ello, se debe colocar previo a la definición de una variable la palabra reservada *Type*. De esta forma, se crea un nuevo tipo de dato que puede ser utilizado para la creación de nuevas variables. La figura 5 presenta un ejemplo de declaración válida empleando un tipo de dato definido.

```
Type Array aWeek of Boolean [1..7]
aWeek tLaboral, tHolidays
```

Figura 5. Un ejemplo simple de la utilización de un tipo de dato definido. Fuente: Propia.

3.5 Tipo Pointer

El pseudocódigo soporta el uso de apuntadores ó punteros los cuales almacenan una dirección de memoria en donde por lo general se encuentra un dato, sea elemental o estructurado.

El tipo Pointer es la base para la creación de la mayoría de las estructuras dinámicas (e.g. listas, árboles, etc.). Entre los operadores empleados por este tipo está la derreferenciación que retorna el dato referenciado por el apuntador, y se debe colocar antes del identificador que se desea derreferenciar. Para ello, se emplea el símbolo *'*'* para derreferenciar y se recomienda colocarlo entre paréntesis junto con el identificador para mayor legibilidad. Por otro lado, este tipo emplea el operador de referencia el cual retorna una dirección de memoria de una variable u objeto, y se simboliza mediante la palabra *ref*. Se puede observar en la figura un ejemplo del uso de este tipo.

```
Type Register Point
  Real x,y
end
  Real x,
Point ptMyPoint
Point * pPointer
pPointer = ref ptMyPoint  //pPointer toma la dirección en donde está almacenada ptMyPoint
*pPointer.x = 4           //se asigna 4 al campo de x de pPointer, equivale a ptMyPoint = 4
*(ref ptMyPoint).y = 3    //equivalente a ptMyPoint.y = 3
*(*(ref pPointer)).x = 1  //equivalente a ptMyPoint.x = 1
```

Figura 5. Utilización del tipo Pointer con los operadores referenciación y derreferenciación. Fuente: Propia.

El operador *'*'* tiene prioridad sobre el operador punto *'.'* utilizado para acceder miembros de registros y objetos, y éste sobre el operador *ref*. Otros operadores de interés para los apuntadores son el operador *new* y *delete*.

El operador *new* reserva el espacio en memoria suficiente para el tipo de dato especificado como parámetro y retorna la dirección de memoria que hace referencia a dicho espacio. Si no se logró reservar el espacio entonces retorna la constante *NULL* o

NIL. Del mismo modo, el operador *delete* libera la memoria dinámica reservada mediante el operador *new*. Se debe colocar *delete* seguido del identificador del apuntador. Como efecto de la operación, la memoria reservada será liberada y estará disponible para alojar nuevos datos, pero por estar libre, no puede ser accedida, pues generaría un fallo de protección de memoria.

3.6 Estructuras de control

Del mismo modo que en diversos lenguajes de programación, las estructuras de control permiten componer operaciones basadas en un conjunto de condiciones que hacen que se ejecute de forma excluyente un código. La notación propuesta define la selección ó condicional simple, condicional doble y selección múltiple.

Condicional Simple

Dada una condición <*condicionB*> que representa a una expresión que retorna un valor del tipo *Boolean*, un condicional simple se escribe en pseudocódigo como:

if <*condicionB*> *then*
  <*instruccion*$_1$>
  <*instruccion*$_2$>
  ...
  <*instruccion*$_k$>
end

Dentro del cuerpo del condicional puede haber 1 ó más instrucciones que a su vez puede ser un condicional simple.

Condicional Doble

Dada una condición <*condicionB*> que representa a una expresión que retorna un valor del tipo *Boolean*, se puede definir un condicional doble como:

if <*condicionB*> *then*
  <*instruccion*$^a_1$>
  <*instruccion*$^a_2$>
  ...
  <*instruccion*^a_k>
else
  <*instruccion*$^b_1$>
  <*instruccion*$^b_2$>
  ...
  <*instruccion*^b_k>
end

Si <*condicionB*> toma el valor de true entonces se ejecutan el conjunto de <*instruccion*^a_i>, en caso contrario el conjunto de <*instruccion*^b_i>. Como parte de las instrucciones puede estar uno ó más condicionales simples o dobles.

Cuando luego de la sentencia *else*, existen otras condiciones (simple o doble) es posible emplear la sentencia *elseif*. Su utilidad radica en hacer más legible el código escrito en pseudocódigo dentro de la notación Alpha. Su sintaxis es como sigue:

if <*condicionB*^{1}> *then*
  <*instruccion*$^a_1$>

<instruccion$^{a}_{2}$>
...
<instruccion$^{a}_{k}$>
elseif <condicionB^{2}> then
<instruccion$^{b}_{1}$>
<instruccion$^{b}_{2}$>
...
<instruccion$^{b}_{k}$>
end

Selección Múltiple

Dado un conjunto de condiciones que son excluyentes entre sí, se puede definir la estructura de selección múltiple como:

select
<condición$_1$> : <instruccion$_1$>
<condición$_2$> : <instruccion$_2$>
...
<condición$_k$> : <instruccion$_k$>
end

Cabe destacar que la estructura *select* solo es válido si contempla una partición de las condiciones posibles, es decir, solo satisface una de las condiciones *<condicionk>*. La estructura *select* se basa en la partición del rango de una variable y la verificación de las condiciones solo se hace con valores literales.

3.7 Estructuras iterativas

Las estructuras iterativas permiten ejecutar un conjunto de instrucciones de forma condicionada, en la notación se definen 4 tipos de estructuras iterativas: *while*, *do-while*, *for* y *foreach*.

While y do-while

Dada una *<condicion>*, la estructura iterativa *while* se define como:

while <condicion> do
<instrucción$_1$>
<instrucción$_2$>
...
<instrucción$_k$>
end

La definición de la estructura *do-while* en la notación Alpha es como sigue:

do
<instrucción$_1$>
<instrucción$_2$>
...
<instrucción$_k$>
while <condicion>

En la figura 6 se muestra el cálculo del valor del factorial empleando la estructura *while* y *do-while*.

```
Integer iFactorial = 1, iNumber = 8            Integer iFactorial = 1, iNumber = 8
```

```
while iNumber > 1 do
  iFactorial = iFactorial * iNumber
  iNumber = iNumber - 1
end
Print (iNumber + "!=" + iFactorial)
```

```
do
  iFactorial = iFactorial * iNumber
  iNumber = iNumber - 1
while iNumber > 1
Print (iNumber + "!=" + iFactorial)
```

Figura 6. Cálculo del factorial empleando la estructura *while* y *do-while*. Fuente: Propia.

For y foreach

La estructura iterativa *for* se indica de forma similar a muchos lenguajes de programación, y se indica de forma opcional el incremento a realizar. Si dicho incremento no es indicado, se asume que es 1. Esta se define como:

for <identificador> = <vInicial> to <vFinal> [step <incremento>] do
<instrucción$_1$>
<instrucción$_2$>
...
<instrucción$_k$>
end

donde *<vInicial>* representa el valor inicial de iteración y *<vFinal>* el valor final; e *<incremento>* indica el número de pasos a incrementar en cada paso.

La estructura *foreach* es una estructura iterativa similar a *for* que mantiene un contador explícito y permite iterar sobre todos los elementos de una secuencia de elementos. Su definición es como sigue:

foreach <identificador> in <vCollection>
<instrucción$_1$>
<instrucción$_2$>
...
<instrucción$_k$>
end

donde *<identificador>* representa un elemento del conjunto *<vCollection>* en cada iteración de la estructura durante su ejecución.

```
Integer iFactorial = 1, iNumber
for iNumber = 2 to 8 do
  iFactorial = iFactorial * iNumber
end
Print (iNumber + "!=" + iFactorial)
```

```
Integer iFactorial = 1, iNumber
Array aValues of Integer [] = {2,3,4,5,6,7,8}
foreach iNumber in aValues
  iFactorial = iFactorial * iNumber
end
Print (iNumber + "!=" + iFactorial)
```

Figura 6. Cálculo del factorial empleando la estructura *for* y *foreach*. Fuente: Propia.

3.8 Acciones y Funciones

Las acciones y funciones permiten estructurar el código en segmentos con el fin de realizar tareas individuales y particulares. Una acción o función agrupa un conjunto de sentencias bajo un identificador (nombre de la acción/función) y que puede ser invocado en cualquier punto dentro de un algoritmo. La diferencia entre ambas es que la función retorna un tipo de dato que debe ser indicado en su definición, mientras que la acción no.

La definición de una acción viene dada por la palabra clave *void* de la siguiente forma:

void <identificador> ([<parámetro$_1$>[,<parámetro$_k$>]])
<definiciones>
<instrucciones>

end

Por otro lado, la definición de una función es con la palabra *function* como sigue:

void <identificador> ([<parámetro$_1$>[,<parámetro$_k$>]])
  <definiciones>
  <instrucciones>
end

En la figura 7 se muestra un ejemplo de acción y función empleando la notación Alpha donde se hace el cálculo del factorial y se imprime su valor.

```
void Factorial (Integer iN)
  Integer iFactorial = 1
  while iN > 1 do
    iFactorial = iFactorial * iN
    iN = iN - 1
  end
  Print (iN + "!=" + iFactorial)
end
Integer iNumber = 8
Factorial (iNumber)
```

```
function Factorial (Integer iN) : Integer
  Integer iFactorial = 1
  while iN > 1 do
    iFactorial = iFactorial * iN
    iN = iN - 1
  end
  return iFactorial
end
Integer iNumber = 8
Print (iNumber + "!=" + Factorial(iNumber))
```

Figura 7. Ejemplo para el cálculo del factorial en una acción y función. Fuente: Propia.

Nótese en la figura 7 que después de la definición de la acción y función, se ejecutan instrucciones dada las características del pseudocódigo propuesto.

El pase de parámetros tanto de la acción como función puede ser por valor o por referencia. Por defecto, los valores son pasados por valor, y utilizar un parámetro por referencia debe ser solamente un identificador y anteponer la palabra reservada *ref*.

3.9 Clases

Una clase es una estructura fundamental en la programación orientada a objetos que representa una entidad o concepto, las cuales definen un conjunto de variables miembros y métodos apropiados para operar con dichas variables. Cada instancia de una clase de denomina objeto. La notación Alpha soporta el uso de las clases bajo la siguiente estructura:

class <NombreClase>[<Tipo>] [inherited <ModificadorDeAcceso> <NombreClasePadre>]
<ModificadorDeAcceso>:
  <atributo$_1$>
  <atributo $_2$>
  ...
  <atributo$_k$>
  [Constructor <NombreClase>
    <instrucciones>
  end]
  [Destructor <NombreClase>
    <instrucciones>
  end]
  [static | const | virtual] <Acción$_1$/Función$_1$>
  [static | const | virtual] <Acción$_2$/Función$_2$>
  ...
  [static | const | virtual] <Acción$_k$/Función$_k$>

end
Por convención en este artículo, se utilizará la primera letra del identificador de una clase (i.e. *<NombreClase>*) en mayúsculas. Los modificadores de acceso (i.e. *<ModificadordeAcceso>*) serán tres: *private*, *protected* y *public* para privado, protegido y público, y pueden ser aplicado tanto sobre los atributos como las acciones/funciones miembros. El constructor y destructor se indica explícitamente en la definición de la clase. Por último, la herencia es soportada al utilizar de forma opcional en la declaración de la clase la palabra reservada *inherited*, y el uso de plantillas o *templates* para un tipo de datos con el uso de un identificador de tipo entre los símbolos *'<'* y *'>'*. Los aspectos ligados a la programación orientada a objetos tales como conceptos de herencia, atributos, polimorfismo, ligadura dinámica, sobrecarga, entre otros pueden ser consultados en los textos adecuados (Gamma, Helm, Johnson, & Vlissides, 1994; Blaha, & Rumbaugh, 2004).

Para hacer referencia a los atributos y métodos de una clase se utiliza el operador punto '.' al igual que el tipo *Register*. Los objetos son declarados de la misma forma de las variables, y si contiene el Constructor entonces éste es invocado al momento de su declaración. Cuando se declara la instancia de la clase, se puede emplear el operador new para reservar su espacio en memoria. En caso de no emplear dicho operador, entonces se asume que el Constructor por defecto es invocado; en caso de tener parámetros dicho operador es de forma obligatoria.

En la notación Alpha, se asume que la herencia es simple, es decir, una clase deriva solamente de una clase base. Igualmente, es posible tener acceso a los atributos y métodos de la clase base empleando la palabra reservada *super* (e.g. *super.atributo*).

4. EXPERIMENTACIÓN Y RESULTADOS

Las pruebas realizadas con la notación Alpha se baso en utilizarla en diversas herramientas de la enseñanza de algoritmia tales como clases presenciales, guías teóricas y prácticas, y exámenes. Así, se realizaron experimentos cualitativos para medir el impacto de la notación Alpha en un grupo de estudiantes. El escenario de pruebas consistió en emplear la notación durante un período de 15 semanas a los estudiantes de una asignatura de programación, que forma parte del programa de la Licenciatura en Computación de la Universidad Central de Venezuela ubicado en Caracas, Venezuela.

En las clases impartidas en el salón de clases, la notación Alpha se empleó para construir los algoritmos y estructuras de datos requeridos para lograr las competencias de la asignatura. Un ejemplo de ello se observa en la figura 8 donde se escribe una implementación sencilla basada en apuntadores para una estructura de datos de Pila (i.e. *Stack*) empleando clases y siendo utilizada.

```
class Node <T>      //definición de una clase sin constructor/destructor y basado en plantilla
public:
  T tInfo
  Node * pNext
end
class Stack <T>     //definición de la clase Stack empleando en apuntadores
private:
  Node <T> * pTop
  Integer iN
public:
  Constructor Stack()
    iN = 0
    pTop = NIL
  end
  Destructor Stack()
    while pTop != NIL do
```

```
      Node<T> * pTemp = pTop
      pTop = *pTop.pNext
      delete pTemp
    end
  end
  function IsEmpty() : Boolean
    return iN == 0
  end
  void Push(ref T x)
    Node<T> * pNew = new Node
    *pNew.x = *x
    *pNew.pNext = pTop
    pTop = pNew
    iN = iN + 1
  end
  void Pop()
    Node<T> * pTemp = pTop
    pTop = *pTop.pNext
    *pNew.pNext = pTop
    delete pTemp
    iN = iN - 1
  end
  function Size() : Integer
    return iN
  end
  function Top() : T
    return *pTop.tInfo
  end
end
//a continuación se emplea la clase definida Stack
Stack myStack <Integer> = new Stack()
myStack.Push(1);  myStack.Push(3);  myStack.Push(5);  myStack.Push(8);
Print (myStack.Size())  //la salida es 4
Print (myStack.Top())  //la salida es 8
myStack.Pop()
myStack.Pop()
Print (myStack.Top())  //la salida es 3
```

Figura 8. Ejemplo de la creación de la clase Stack (que define una Pila) empleando apuntadores bajo la notación Alpha. Fuente: Propia.

Con el objetivo de conocer la reacción ante el uso de la notación en los distintos cursos de la asignatura, se realizó una encuesta a un total de 94 personas para conocer el impacto la notación durante el curso y considerar cambios pertinentes en su estructuración. En la tabla 3 se muestra las preguntas realizadas.

Pregunta	**Escala**
¿La notación empleada en el curso la considera...?	Selección en el rango [1,6]: Excelente (1) a Deficiente (6)
Considera Ud. que se deba incorporar algún cambio pertinente en la notación empleada	Si la respuesta es afirmativa, se debe escribir de forma libre dicho cambio

Tabla 3. Preguntas realizada para evaluar la notación propuesta. Fuente: Propia

Cabe destacar que dichas preguntas formaron parte de una encuesta de evaluación de diversos aspectos de la asignatura impartida basada en la escala propuesta originalmente por Rensis Likert (1931). Como parte de los resultados se obtiene el gráfico de la figura 9 que indica las respuestas a la primera pregunta.

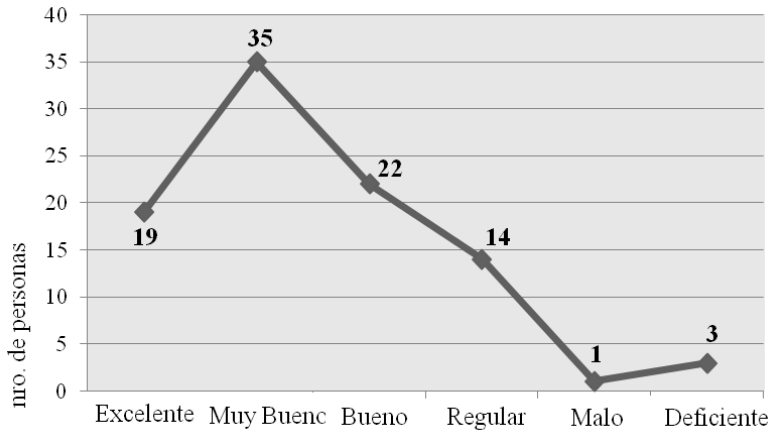

Figura 9. Resultados obtenidos de la encuesta realizada. Fuente: Propia.

El 80.85% de los encuestados consideraron la notación propuesta como excelente (20.21%), muy bueno (37.23%) y bueno (23.41%), lo cual se considera positivo de acuerdo a la escala utilizada. Sin embargo, un 19.15% no la considera de esta forma sino en el rango de regular (15.00%), malo (1.06%) y deficiente (3.09%). Es importante destacar que solamente el 23% de la población encuestada aprobó la asignatura la cual estaban cursando.

Con respecto a la segunda pregunta, el 4.25% pertenecientes al rango excelente, muy bueno y bueno, agregó que se deberían incluir cambios para que la notación se asemeje mucho más al lenguaje de programación empleado durante el curso (para esta investigación, fue el lenguaje C++). Dentro del rango regular, no se presentaron comentarios asociados a un cambio en concreto sino de forma muy vaga expresaron que no era adecuado. Ahora, el 50% perteneciente al rango malo y deficiente indicaron su total descontento con la notación sin ofrecer alguna sugerencia constructiva.

Con los resultados es posible analizar que la notación está cumpliendo su objetivo sin desestimar los comentarios proporcionados por la población encuestada. Al mismo tiempo, se debe considerar la presencia de discrepancias al momento de emplear la notación adecuadamente por parte del cuerpo docente presente en la asignatura, la cual consta de 5 docentes de cátedra y 4 estudiantes en labor de guías para la parte práctica. Así, se infiere que la inconsistencia presente al no cumplir a cabalidad la notación es un factor que pudo causar descontento y apatía con la asignatura así como con sus elementos asociados, incluyendo la notación propuesta.

5. CONCLUSIONES Y TRABAJOS FUTUROS

En este trabajo se presenta una notación basada en pseudocódigo para la construcción de algoritmos que emplea convenciones estructurales presentes en diversos lenguajes de programación, omitiendo los detalles específicos de implementación de dichos lenguajes. La notación, denominada Alpha, contiene estructuras que son útiles en el diseño de programas computacionales que facilitan la enseñanza de algoritmos y estructuras de datos.

Plantear una propuesta que permita formalizar el pseudocódigo tal que se construyan algoritmos y estructuras de datos se hace necesario debido a la no estandarización de éste. La notación Alpha constituye una herramienta poderosa y sumamente útil ya que se

ajusta a las necesidades actuales al constituir una solución a ser empleada en textos, artículos científicos y en la enseñanza. Los tipos de datos y estructuras de datos ofrecidas por la notación permiten construir diversos programas computacionales simples y complejos que abarcan muchos aspectos dentro del área de Algoritmia.

La escritura de un código empleando la notación resulta sencilla y adecuada para la enseñanza de algoritmos y estructuras de datos, permitiendo estructurar las instrucciones y operaciones de forma tal que su conversión a un lenguaje de programación es natural. Así, el resultado de la encuesta aplicada ofrece una visión que permitió determinar su impacto en una población de estudiantes que utilizó la notación durante un curso completo de una asignatura de algoritmos y estructuras de datos.

En un futuro, se propone construir la notación BNF (*Backus-Naur Form*) (Knuth, 1964) para la notación Alpha propuesta y ser empleada como una gramática formal. En el mismo sentido, se propone la construcción de una herramienta computacional que interprete y ejecute el pseudocódigo de la notación Alpha, de forma similar a como el software PseInt (Novara, 2003) pero enfocado en el pseudocódigo propuesto.

REFERENCIAS

Bailey, T., & Lundgaard, K. (1989). Program Design With Pseudocode (Tercera ed.). Brooks/Cole Pub Co.

Blaha, M., & Rumbaugh, J. (2004). Object-Oriented Modeling and Design with UML (Segunda ed.). USA. Prentice Hall.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). Introduction to Algorithms (Tercera ed.). USA. The MIT Press.

Coto, E. (2002). Lenguaje Pseudoformal Para la Construcción de Algoritmos (Nota de Docencia ND 2002-08). Venezuela. Lecturas en Ciencias de la Computación, Universidad Central de Venezuela.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). Algorithms. USA. Mc Graw Hill.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. USA. Addison-Wesley Professional.

Gilberg, R., & Forouzan, B. (2004). Data Structures: A Pseudocode Approach with C (Segunda ed.). USA. Cengage Learning.

Jones, C. (1990). Systematic Software Development Using Vdm (Segunda ed.). USA. Prentice-Hall.

Joyanes, L. (2008). Fundamentos de Programación (Cuarta ed.). España. McGraw Hill Interamericana S.A.

Knuth, D. (1964). Backus Normal Form vs. Backus Naur Form. Communications of the ACM, 7(12), pp. 735-736.

Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (Tercera ed.). USA. Prentice Hall.

Likert, R. (1931). A technique for the measurement of attitudes. Archives of Psychology. 22(140), pp. 1-55.

Martínez, A., & Rosquete, D. (2009). NASPI: Una Notación Algorítmica Estándar para la Programación Imperativa. Télematique. 8(3). pp. 55-74.

Neapolitan, R. (2003). Foundations of Algorithms Using C++ Pseudocode (Tercera ed.). USA. Jones and Bartlett Publishers.

Neapolitan, R., & Naimipour, K. (2004). Foundations of Algorithms Using Java Pseudocode. USA. Jones & Bartlett Learning.

Novara, P. (2003). PSeInt. Consultado el 31 de Octubre 2014, extraído de http://pseint.sourceforge.net/

McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction (Segunda ed.). USA. Microsoft Press.

Ottogalli, K., Martínez, A., & León, L. (2011). NASPOO: Una Notación Algorítmica Estándar para Programación Orientada a Objetos. Télématique. 10(1). pp. 81-102

Peña, R. (2005). Diseño de programas. Formalismo y abstracción (Tercera ed.). España. Prentice Hall.

Pes, C. (2006). Empezar de Cero a Programar en Lenguaje C. España. Autoedición.

Sedgewick, R., & Wayne, K. (2011). Algorithms (Cuarta ed.). USA. Pearson Education.

Sharp, A. (2008). Workflow Modeling: Tools for Process Improvement and Application Development (Segunda ed.). USA. Artech House.

Skiena, S. (2010). The Algorithm Design Manual (Segunda ed.). USA. Springer.

Spivey, J. (1992). The Z Notation: A Reference Manual (Segunda ed.). USA. Prentice-Hall.

Zobel, J. (2004). Writing for Computer Science (Segunda ed.). USA. Springer