Introducción al ΔΔLang

Matias Gobbi

10 de enero de 2020

Fundamentos

Ya introducido el lenguaje, es hora de comenzar propiamente con la descripción informal e intuitiva de nuestro intérprete. Por lo tanto, en la siguiente sección nos dedicaremos a dar un vistazo general de las distintas construcciones que ofrece el ΔΔLang. Realizaremos una comparación entre el pseudocódigo utilizado en la materia, y nuestra implementación del lenguaje.

1. Introducción

El ΔΔLang es la implementación del pseudocódigo utilizado en la materia *Algoritmos y Estructura de Datos II*. Su objetivo es poder complementar la enseñanza de los contenidos de la asignatura, ofreciendo otro recurso para el aprendizaje de los conceptos estudiados en la misma.

La idea del lenguaje es poder permitir a los estudiantes implementar los distintos algoritmos vistos en la materia, utilizando una sintaxis más simple, y una semántica más transparente en comparación a la que se utiliza actualmente en la asignatura, basada en el lenguaje de programación *C*. De esta forma, se ofrece una transición más amena entre el diseñado de programas y la implementación de los mismos.

Debido a que el pseudocódigo de la materia no presenta ninguna especificación precisa, y permite diversos tipos de ambigüedades, se tomaron distintas decisiones de diseño a la hora de la implementación del ΔΔLang. A lo largo del informe se irán detallando las mismas, junto con las comparativas apropiadas entre ambos lenguajes.

ΔΔLang es implementado utilizando el lenguaje de programación funcional, *Haskell*. Actualmente, solo se encuentran definidas la sintaxis del lenguaje y el parser del intérprete. En la siguiente iteración, se planea agregar los chequeos estáticos correspondientes al lenguaje. Para la etapa del parser, se utilizó la librería *Parsec*.

2. Características Principales

En esencia, el lenguaje implementado busca ser lo más próximo posible al pseudocódigo empleado en la materia. A pesar de esto, se presentan varias diferencias entre uno y otro por diversos motivos. Algunas cambios se deben a la falta de especificación, por lo que se decidió proceder de la forma más apropiada según el contexto. Otras decisiones se tomaron para facilitar la implementación del intérprete, y algunas pensando en futuras expansiones del lenguaje.

Por mencionar algunas de las modificaciones, a continuación se enumeran algunos de los cambios más importantes que se realizaron a la hora de la implementación:

1. Se definieron cuales son los tipos nativos del lenguaje, y por lo tanto, también los que deberían ser implementados por el programador para su uso. De esta forma queda especificado que tipos, básicos o estructurados, ofrece el lenguaje. También se estructuró la forma de definir nuevos tipos de datos, aspecto que se manejaba de forma ambigua en el pseudocódigo. De esta forma, se limita que clase de valores se pueden representar en el lenguaje.

2. Se estableció una forma precisa de especificar instrucciones. Anteriormente, había múltiples formas de definir las mismas, lo que complicaba la implementación del parser del intérprete. También se definieron las distintas versiones de la instrucción **for**, simplificando la semántica del comando. La sintaxis de las sentencias queda abierta a posibles futuras expansiones.

3. En el caso de las expresiones, se fijaron los operadores definidos que ofrece el lenguaje. En el pseudocódigo se trabaja con una enorme cantidad de los mismos, por lo que está decisión busca simplificar la sintaxis del lenguaje. Una posible expansión de las expresiones es la inclusión de los *strings* como constantes.

4. Finalmente, se definió la forma de especificar identificadores. Ya sea para variables, métodos, o incluso tipos, existe una forma precisa de nombrar estas construcciones. Anteriormente en el pseudocódigo, este aspecto se trataba con libertad, lo que ocasionaba confusión sobre que representaba cada identificador.

El intérprete, al igual que el lenguaje, todavía se encuentra en la fase de desarrollo, por lo que muchas de las características del mismo se encuentran sujetas a posibles cambios futuros. Actualmente, la sintaxis del lenguaje está definida, al igual que el parser para el intérprete. Para las siguientes iteraciones tenemos pendientes la implementación de los chequeos estáticos correspondientes, la semántica *one-step* para la ejecución de programas, y la interfaz de usuario para el uso del intérprete.

3. Tokens

Comencemos propiamente con la descripción del intérprete, más precisamente con el parser. Para la construcción del código, los bloques léxicos básicos se denominan tokens. Se podrían definir como las palabras del lenguaje. Los caracteres ingresados por el programador se combinan en tokens de acuerdo a las reglas del lenguaje de programación. La tarea del parser es entonces, tomar la cadena de tokens ingresados y transformarlo en el **AST** (*abstract syntax tree*) correspondiente. Hay distintas clases de tokens en ΔΔLang:

1. **palabras reservadas** son las que poseen un significado fijo en el lenguaje. Se utilizan principalmente como delimitadores para las distintas construcciones. Se escriben en minúscula.
2. **identificadores** son los nombres que el programador define para las variables, métodos, tipos, entre otras construcciones. Hay distintas clases según que se desea representar con los mismos.
3. **operadores** son los símbolos para las operaciones. Se utilizan principalmente en las expresiones. A diferencia del pseudocódigo, todos los operadores se especifican mediante combinaciones de símbolos.
4. **separadores** comúnmente son los espacios en blanco. También pueden ser las tabulaciones o saltos de línea. No tienen mayor uso que mantener el código prolijo.
5. **constantes** se utilizan para denotar valores específicos en el código. Ejemplos de estos son los valores numéricos, los booleanos, los caracteres, y las constantes definidas.

A lo largo del desarrollo de este informe iremos haciendo mención de cada uno de los mismos a medida que sea necesario. A continuación, ilustramos los conceptos introducidos con un ejemplo análogo al dado en la introducción al pseudocódigo. Este fragmento de código pasaría la etapa de parsing del intérprete, pero no es un programa válido ya que no satisface todos los chequeos estáticos. En particular, utiliza un tipo no definido *nat*.

```
proc insert (in/out a: array [1..n] of T, in i: nat)
  var j: nat
  j := i
  while j > 1 && a[j] < a[j-1] do
    swap(a, j - 1, j)
    j := j - 1
  od
end proc
```

Los significados de cada una de las construcciones especificadas en el ejemplo serán explicadas más adelante en sus respectivas secciones. De todas formas, la semántica de cada una de las mismas es idéntica a la del pseudocódigo introducido anteriormente.

3.1. Comentarios

Los comentarios son piezas de código que son descartadas por el intérprete. Solo existen para el beneficio del programador, para poder documentar ciertos fragmentos del programa. En el lenguaje solo se permiten realizar comentarios en bloque utilizando llaves, como en el siguiente ejemplo.

```
{ Este procedimiento inicializa un arreglo }
```

4. Tipos

Un tipo provee un conjunto de valores los cuales pueden ser tomados por una expresión al ser evaluada. En el lenguaje se ofrecen tipos básicos, tipos estructurados, e incluso la posibilidad de definir nuevos tipos de datos.

4.1. Tipos Básicos

Un tipo básico comprende un conjunto ordenable de valores. Existe un token para cada valor del conjunto. Poseen operaciones propias que los manipulan. A excepción de los *números naturales*, ΔΔLang ofrece los mismo tipos básicos que el pseudocódigo.

4.1.1. Tipos Numéricos

Los tipos numéricos son los enteros **int**, y los reales **real**. Los números naturales no fueron implementados ya que se consideró que no poseían ninguna utilidad práctica a la hora del diseñado de programas. En el lenguaje los números enteros se especifican en notación decimal, mientras que los números flotantes no permiten exponentes.

A diferencia de su definición matemática, estos conjuntos están acotados. En el lenguaje se encuentra definida la constante **inf**, que representa el límite superior de ambos conjuntos numéricos. Asumiendo que n y m son expresiones numéricas del mismo tipo. Los operadores aritméticos que se ofrecen son los siguientes. Los mismos evalúan a un valor numérico.

Operador	Operación
− n	Resta Unaria
n − m	Resta Binaria
n + m	Sumatoria
n ∗ m	Multiplicación
n / m	División
n % m	Módulo

Notar que los operadores para números están sobrecargados. Esto significa que las operaciones numéricas están definidas para enteros y reales. Todavía queda pendiente la decisión respecto a si el lenguaje soporta subtipado de enteros en reales. Otra posibilidad también considerada, es la implementación de conversiones implícitas de tipo entre los conjuntos numéricos.

4.1.2. Tipos Booleanos

El tipo **bool** representa al conjunto de valores booleanos. Puede adoptar los valores de verdad **true** y **false**. También se ofrecen las operaciones clásicas para manejo de booleanos. Asumiendo que p y q son expresiones de tipo booleano, los operadores son los siguientes.

Operador	Operación
! p	Negación
p || q	Disyunción
p && q	Conjunción

Los operadores de orden e igualdad pueden ser utilizados para comparar valores del mismo tipo. Sean a y b expresiones del mismo tipo, los operadores de comparación son los siguientes. Los mismos evalúan a un valor booleano.

Operador	Operación
a == b	Igualdad
a ! = b	Desigualdad
a < b	Menor
a > b	Mayor
a <= b	Menor o Igual
a >= b	Mayor o Igual

Todavía no se ha determinado de forma precisa cuales tipos implementan estas operaciones. Idealmente, todos los tipos básicos deberían poder ser comparados al igual que los tipos constantes definidos por el usuario. Esta es otra decisión pendiente a ser implementada.

4.1.3. Tipos Caracteres

El tipo **char** representa al conjunto de caracteres. Los valores que puede adoptar se representan encerrados entre comillas simples, de la forma *'a'*. Al igual que en el pseudocódigo, este tipo de dato no presenta mucha utilidad en el lenguaje. Inicialmente, las únicas operaciones que se podrían realizar entre caracteres son las comparativas.

En el siguiente ejemplo, análogo al dado en la introducción al pseudocódigo, se detalla una función que utiliza los tipos básicos recientemente introducidos. La misma determina si un arreglo de caracteres posee llaves balanceadas, utilizando un contador entero.

```
fun balanceados (a: array [1..n] of char) ret b: bool
  var count: int
  count := 0
  b := true
  for i := 1 to n do
    if a[i] == '[' then
      count := count + 1
```

```
    elif a[i] == ']' && count != 0 then
      count := count - 1
    elif a[i] == ']' && count == 0 then
      b := false
    fi
  od
  b := b && count == 0
end fun
```

4.2. Tipos Estructurados

Un tipo estructurado es caracterizado por el tipo de sus componentes y por su método de estructuración. Al igual que los tipos básicos, también hay operaciones específicas definidas para los mismos. A diferencia del pseudocódigo, ΔΔLang solo ofrece arreglos y punteros como tipos estructurados. Las listas y los conjuntos podrían ser definidos en base a estos tipos nativos.

4.2.1. Tipo Arreglo

Un tipo **array** es una estructura que consiste de una cantidad fija de componentes del mismo tipo. Los elementos de un arreglo son designados por índices. Los mismos deben representar un conjunto de valores enumerables. Para definir un arreglo se necesita especificar el tipo de los componentes y los índices de los mismos.

```
var a: array [1..4] of int
a[1] := 1
a[2] := a[1] + 2
a[3] := a[2] + 3
a[4] := a[3] + 4
```

En este ejemplo, creamos un arreglo *a* que estará definido para los índices *1*, *2*, *3*, y *4*, y almacenará en cada posición valores de tipo entero. Luego, en cada posición *i* del arreglo se guardará la sumatoria de los primeros *i* números naturales.

La declaración de índices de arreglos no está solamente limitada a valores numéricos. De hecho, el lenguaje soporta cuatro formas distintas de definir límites para los mismos.

- *Enteros*: Como en el ejemplo anterior, los índices pueden tomar cualquier valor entero, incluso valores negativos.

  ```
  var intArray: array [-5..8] of bool
  intArray[-1] := true
  ```

- *Caracteres*: También se pueden utilizar caracteres para definir los límites de un arreglo.

```
var charArray: array ['a'..'z'] of real
charArray['m'] := 3.14
```

- *Constantes*: Una vez definido un tipo constante, se pueden utilizar sus valores para marcar los índices válidos de un arreglo.

```
var constArray: array[Lunes..Viernes] of char
constArray[Miercoles] := 'p'
```

- *Variables*: Para definir métodos de forma genérica, se pueden definir límites variables para arreglos. Inicialmente uno desconoce el rango de valores para los cuales el arreglo a recibir como parámetro estará definido, por lo que se pueden marcar estos límites como variables.

```
var varArray: array[1..n] of int
varArray[n] := 10
```

Por último, los arreglos en ΔΔLang no están limitados a una dimensión. En particular, uno puede crear arreglos multidimensionales simplemente especificando los rangos de las dimensiones necesarias al momento de declarar un nuevo arreglo.

```
var x: array [1..5, 1..5] of int
var y: array [1..3, 'a'..'e', n..m] of real
x[1, 1] := 4
y[2, 'b', n] := 4.0
```

El arreglo x posee dos dimensiones, ambas indexadas desde el número *1* al *5*. El arreglo y, en cambio, tiene 3 dimensiones. La primera está indexada por enteros, la segunda por caracteres, y la última por índices variables. Para la asignación de valores del arreglo simplemente se separan con comas las distintas coordenadas que se quieren acceder o actualizar.

4.2.2. Tipo Puntero

Un tipo **pointer** es una estructura compuesta por un solo componente. Para el manejo dinámico de memoria se utilizan punteros. Con los mismos se reserva, aloja y libera memoria para el componente apuntado por el puntero, a lo largo de la ejecución del programa.

```
var p: pointer of int
```

La declaración anterior crea una variable p que será un puntero. La misma servirá para direccionar un valor numérico entero en un futuro. Antes de realizar cualquier operación sobre p, se deberá reservar memoria para el componente señalado, y una vez finalizado el cómputo se deberá liberar la memoria utilizada.

```
alloc(p)
#p := someComplexFunction()
```

```
someComplexProcedure(p)
free(p)
p := null
```

En el ejemplo anterior se ilustran algunas operaciones que se pueden realizar cuando se trabajan con punteros. El procedimiento **alloc** se encarga de reservar espacio en memoria para almacenar el componente a apuntar por el puntero *p*. En la siguiente sentencia, se inicializa el mismo para después realizar alguna computación. Luego, **free** libera el espacio en memoria apuntado por *p*. Finalmente, la constante **null** simboliza un puntero que no señala a ninguna posición de memoria. Asumiendo que *a* es un alias correspondiente a una tupla señalada por *p*. Las operaciones que manipulan punteros son las siguientes.

Operador	Descripción
#p	Devuelve el componente en memoria
p -> a	Devuelve el campo asociado a la tupla en memoria

Como se puede observar, los punteros permiten manejar explícitamente direcciones de memoria y sus contenidos, por lo que permiten programar a bajo nivel. Los mismos tienen gran importancia a la hora de definir tipos de datos abstractos. Como ΔΔLang no posee *garbage collector* es responsabilidad del programador administrar el uso de memoria del programa.

4.3. Definición de Tipos

Una característica muy importante de ΔΔLang es la posibilidad de crear nuevos tipos de datos. El lenguaje hace uso del tipado fuerte, es decir, toda expresión posee un tipo definido. Debido a esto, se simplifica la interpretación del código y se fomenta la generación de código prolijo. Pero al momento de querer resolver problemas de mayor complejidad, es necesario poder extender los tipos del lenguaje con otro más sofisticados y robustos. Por lo tanto, ΔΔLang ofrece ciertas construcciones con estos fines.

Para poder crear un nuevo tipo de datos se utiliza la palabra clave **type** seguida del identificador del nuevo tipo. El mismo debe comenzar en minúsculas y seguir con una combinación de letras, números y guiones. Algunas definiciones permiten el agregado de parámetros de tipo. Finalmente, en base a que clase de tipo se quiere definir, se especifica el cuerpo de la declaración correspondiente.

4.3.1. Tipo Constante

Los tipos constantes, también llamados tipos enumerados, representan uno de los tipos que pueden ser definidos por el programador en el lenguaje. Consisten en una enumeración de identificadores para constantes que luego podrán ser utilizadas en el resto del programa como valores definidos.

```
type dia = ( Domingo,
             Lunes,
```

```
            Martes,
            Miercoles,
            Jueves,
            Viernes,
            Sabado
        )
```

Para definir un tipo enumerado, se listan todos los valores posibles que puede adoptar, entre paréntesis. Los identificadores de constantes deben comenzar con mayúsculas y luego seguir con una combinación de letras, números, y guiones. Las únicas operaciones disponibles para este tipo de datos serían las comparativas.

4.3.2. Tipo Tupla

Otro tipo definible por el usuario son las tuplas. Las mismas, son una estructura compuesta por una cantidad fija de componentes, posiblemente de distintos tipos. Cada componente se denomina *campo*, y el identificador asociado a cada campo se llama *alias*. Son utilizadas para empaquetar una serie de valores que se relacionan semánticamente. La única operación disponible para trabajar con valores de tipo tupla es la de obtener **.** un dato almacenado en la misma. Dada una tupla t y un alias a, se puede acceder al campo asociado a ese alias con $t.a$.

```
type persona = tuple
               inicial: char,
               edad: int,
               peso: real
               end tuple
```

En el ejemplo, se declara un nuevo tipo *persona*. Los alias son *inicial*, que almacena un carácter, *edad*, que almacena un entero, y *peso*, que almacena un real. Notar el uso de las palabras claves **tuple** y **end tuple** para definir la misma. Los identificadores de componentes deben comenzar en minúsculas y luego seguir con una combinación de letras, números y guiones. En el ejemplo siguiente, se declara una variable con el tipo recientemente definido y se inicializan sus campos.

```
var t: persona
t.inicial := 'm'
t.edad := 18
t.peso := 60.0
```

La creación de tuplas también permite una definición con tipos paramétricos. De esta forma, se pueden crear tipos más genéricos. A continuación se define un tipo *par*, cuyos dos componentes pueden tener tipos distintos en base a como se instancie la tupla.

```
type par of (A, B) = tuple
                     fst: A,
```

```
                        snd: B
                        end tuple

fun div (a, b: int) ret res: par of (int, int)
  res.fst := a / b
  res.snd := a % b
end fun
```

Finalmente, las tuplas junto con los punteros permiten la creación de tipos recursivos. Los mismos, son aquellos donde el nombre del nuevo tipo figura también en la definición del mismo. Permitiendo definir así estructuras teóricamente infinitas. Todavía no está definido hasta que punto se permite la recursividad en la definición de tipos tupla.

```
type node = tuple
            value: int,
            next: pointer of node
            end tuple
```

Para ilustrar, daremos un ejemplo donde se implementa una función que trabaja con el nuevo tipo de dato definido. Se puede pensar a la estructura *node* como una celda que almacena un valor entero y señala a otro *node*, conformando así una posible lista abstracta.

```
fun crearListaAbs (n: int) ret l: pointer of node
  var aux: pointer of node
  l := null
  for i := n downto 1 do
    alloc(aux)
    aux->value := i
    aux->next := l
    l := aux
  od
end fun
```

4.3.3. Sinónimos de Tipos

Los últimos tipos definibles son los sinónimos de tipos. Los mismos representan una abreviatura de un tipo ya existente. La principal funcionalidad de los mismos es permitir un mayor nivel de abstracción en la implementación de programas, y facilitar la lectura del código. Un posible uso de los sinónimos es el siguiente.

```
type matriz = array [1..5, 1..5] of int
```

En el ejemplo, se creó un nuevo tipo *matriz*. Este sinónimo representa simplemente un arreglo bidimensional de enteros. Una vez definido, se puede utilizar el nuevo tipo de forma intercambiable con el tipo de su cuerpo.

Los sinónimos de tipos también permiten una definición paramétrica, al igual que las tuplas. De esta forma, se pueden crear sinónimos mucho más generales. Por ejemplo, se puede crear un sinónimo para un arreglo tridimensional donde el tipo de sus elementos podrá ser instanciado en base a lo que se necesite en el momento.

```
type tensor of (T) = array [1..10, 1..10, 1..10] of T
```

5. Expresiones

Las expresiones son utilizadas en asignaciones, guardas, y en pasaje de parámetros. Están compuestas por operadores y operandos. Producen un valor de cierto tipo, mediante la aplicación de los distintos operadores a sus respectivos operandos. La evaluación de una expresión no altera el estado del programa, ya que no produce efectos secundarios.

El lenguaje se basa en el tipado fuerte. Esto significado que toda expresión posee un tipo, y el mismo debe ser adecuado al contexto en el que se utiliza. Idealmente, una vez implementado los chequeos estáticos, no debería ser posible interpretar un programa que utilice expresiones con tipos inadecuados.

5.1. Constantes

Los valores constantes son una de las posibles clases de expresiones. Su evaluación es inmediata. En el lenguaje hay distintas categorías de constantes. Se pueden especificar valores numéricos, enteros o reales, valores booleanos, caracteres, constantes definidas por el usuario e incluso constantes predefinidas en el lenguaje. Para dar una idea, a continuación se muestra un ejemplo de juguete.

```
var b: bool
b := true && Lunes == Martes && 4 < inf
```

5.2. Llamadas a Funciones

Las llamadas a funciones comprenden otra parte de las expresiones. En las mismas se denota el identificador correspondiente a la función que se desea ejecutar, y luego se listan entre paréntesis todos los parámetros actuales de la misma. El nombre de la función y sus paréntesis deben ser especificados todos juntos, sin espacios entre medio. Obviamente, los tipos de los parámetros y el del resultado devuelto por la función deben ser los adecuados al contexto de ejecución. Un ejemplo de llamada a función puede ser el siguiente.

```
var b: bool
b := esPrimo(1493)
```

5.3. Locations

Las variables, o mejor llamadas *locations*, representan otra de las distintas clases de expresiones. Se utilizan para almacenar información, permitiendo etiquetar estos datos con nombres descriptivos. El identificador de una variable debe comenzar en minúscula, y seguir con una combinación de letras, números y guiones. Hay distintos operadores para cada una de las clases de locations disponibles en el lenguaje. Los mismos deben ser acompañados inmediatamente por la variable que modifican, sin espacios entre medio. Un ejemplo de juguete para ilustrar puede ser el siguiente.

```
var arreglo: array [1..5] of int
var puntero: pointer of node
arreglo[1] := 10
alloc(puntero)
puntero->value := arreglo[1]
puntero->next := null
```

5.4. Precedencia

Los distintos operadores para cada uno de los tipos básicos y estructurados ya se vieron en sus respectivas secciones. Cuando se emplean múltiples operadores en una expresión se utilizan las reglas de precedencia para determinar el orden de evaluación. Ya definido el parser del lenguaje, la precedencia de los operadores implementados es la siguiente. De más alta a más baja.

Operadores	Categoría de Operador
# ->	Punteros
.	Tuplas
! −	Unarios
∗ / %	Multiplicativos
+ −	Aditivos
< > >= <=	Orden
== ! =	Igualdad
&& ||	Booleanos

A la hora de evaluar una expresión, el intérprete se basará en las siguientes reglas para calcular su valor.

- En operaciones con precedencias distintas, se evalúa primero la que tenga mayor precedencia.
- En operaciones con precedencias iguales, se evalúa primero la que se encuentra antes en el código.
- Si se utilizan paréntesis en expresiones, su contenido es evaluado primero al tener la precedencia más alta.

5.5. Coerción de Tipos

A veces es necesario cambiar o adecuar el tipo de una expresión para poder hacer que la evaluación de la misma sea coherente. Es decir, queremos realizar una transformación para que los tipos sean compatibles.

```
var i: real
i := 1 + 2
```

Un ejemplo posible puede ser el anterior. En el mismo tenemos una suma de números enteros, pero la variable a asignar es de tipo real. En estas situaciones hay varias opciones posibles. 1. Informamos al programador sobre un posible error de tipos durante el chequeo estático. 2. Realizamos una conversión *implícita* del valor de la expresión a un valor real. 3. Especificamos subtipado para los tipos numéricos, donde los números enteros pueden ser interpretados también como números reales. Esta es una de las decisiones pendientes a tomar durante la implementación de los chequeos estáticos.

6. Declaración de Variables

El lenguaje tiene una forma de declaración de variables idéntica a la del pseudocódigo. La única restricción es que la declaración solo está permitida al comienzo de un método, antes de cualquier tipo de sentencia. Para crear una variable nueva se utiliza la palabra clave **var**, seguida del identificador de variable y su tipo.

```
var num: int
```

En el lenguaje, las variable no son inicializadas luego de su declaración. Es responsabilidad del programador asignarles un valor inicial correspondientes a las mismas. La lectura de una variable sin previa inicialización detendrá la ejecución del programa produciendo un mensaje de error.

Al igual que en el pseudocódigo, ΔΔLang permite declarar múltiples variables del mismo tipo en una línea. En el siguiente ejemplo, se declaran dos variables nuevas, *aux* y *tmp* de tipo real.

```
var aux, tmp: real
```

Las variables obedecen las reglas de *scope*. Esto significa que la existencia de las mismas se extiende hasta el fin del bloque de código en el que fueron declaradas. Debido a esto, se puede utilizar el mismo identificador para variables diferentes, siempre y cuando el alcance de las mismas sea distinto. Otro detalle sobre esto es que en el lenguaje no se utilizan variables globales, por lo que todas las variables son locales al bloque de código en el que fueron declaradas.

Hay distintas categorías de locations en el lenguaje. Las mismas se pueden dividir en cuatro clases en base a como se accede a la memoria representada por ellas. Cada una de estas, tendrá su propio conjunto de operaciones que las manipulan.

- **Variables** son representadas en su totalidad por un identificador. Es decir, poseen un único lugar en memoria.

```
var pi: real
pi := 3.14
```

- **Variables Indexadas** para obtener un componente de un arreglo se debe indexar de forma apropiada el identificador que denota al mismo.

```
var a: array [1..5] of bool
a[1] := true
```

- **Variables de Campo** para obtener un componente de una estructura se especifica el alias correspondiente al campo requerido.

```
var yo: persona
yo.inicial := 'm'
yo.edad := 18
yo.peso := 60
```

- **Variables Referenciadas** para acceder al valor señalado por un puntero, se utiliza el operador de acceso con el identificador que denota al puntero.

```
var p: pointer of char
alloc(p)
#p := 'g'
```

7. Sentencias

La esencia de un algoritmo son las acciones que realiza. Estas acciones están contenidas en las sentencias del programa, las cuales se dicen ser ejecutables. Las mismas se dividen en sentencias simples, y sentencias estructuradas. $\Delta\Delta$Lang posee las mismas clases de construcciones del pseudocódigo, la diferencia entre ambos radica en que se fijó una única forma de especificar cada una de las mismas en la implementación del lenguaje.

7.1. Sentencias Simples

Una sentencia simple es una sentencia que no está conformada por sub-sentencias. Básicamente, son sentencias *atómicas* que forman la base para la construcción de las sentencias estructuradas.

7.1.1. Asignación

Las asignaciones modifican el valor almacenado en una location, remplazando cualquier dato previo que puede haber tenido. Se utiliza el símbolo :=, precedido por la location a asignar y sucedido por una expresión cuyo valor debe

ser del mismo tipo que la variable a asignar. En el ejemplo, se asigna el valor *80* a la variable de tipo entero *i*.

```
var i: int
i := 4 * 20
```

7.1.2. Llamada a Procedimientos

Las llamadas a subrutinas comprenden otra de las sentencias simples. En las mismas se denota el identificador correspondiente al procedimiento, y luego se listan entre paréntesis todos los parámetros actuales del mismo. El identificador de un procedimiento debe comenzar en minúsculas y seguir con una combinación de letras, números o guiones. El nombre del método y los paréntesis correspondientes deben ir seguidos, sin espacios entre medio de los mismos. Suponiendo que definimos un procedimiento para ordenar los valores de un arreglo *selectionSort*, y tenemos un arreglo *a* de enteros, podemos realizar lo siguiente para invocar al procedimiento.

```
selectionSort(a)
```

7.1.3. Skip

La sentencia *skip* se ha utilizado históricamente en distintos lenguajes de programación para indicar la no realización de ninguna computación. En nuestro lenguaje esta instrucción no presenta ninguna utilidad. Se emplea por fines de claridad en la descripción de distintos algoritmos. No realiza ninguna evaluación y tampoco tiene efectos secundarios.

```
skip
```

7.2. Sentencias Estructuradas

Las sentencias estructuradas son construcciones compuestas por otras sentencias que pueden ser ejecutadas en secuencia, condicionalmente o repetidamente. En el lenguaje, se fijaron las formas de especificar cada una de las mismas, pero manteniendo el poder expresivo del mismo.

7.2.1. Sentencias Secuenciales

Muchos lenguajes de programación permiten la creación de *bloques de código*, compuestos por una serie de sentencias. Las mismas se deben de ejecutar de forma secuencial en el orden que aparecen en el código. En $\Delta\Delta$Lang no existe ninguna construcción explícita para denotar estos bloques, pero si se pueden especificar secuencias de sentencias como en el siguiente ejemplo. El mismo también se podría escribir en una sola línea.

```
minp := minPosFrom(a, i)
swap(a, i, minp)
i := i + 1
```

El ejemplo anterior es un fragmento del algoritmo de ordenación, *selection-Sort*. En el mismo se realiza una asignación, una llamada a un procedimiento y finalmente otra asignación. Los identificadores *minp* e *i* son variables de tipo entero y *a* es un arreglo, mientras que *minPosFrom* y *swap* denotan una función y un procedimiento respectivamente.

7.2.2. Sentencias Condicionales

Todo lenguaje imperativo implementa algún tipo de sentencia condicional. La misma sirve para realizar ejecuciones particulares dependiendo del valor de una condición.

```
if i < n then
  minp := minPosFrom(a, i)
  swap(a, i, minp)
  i := i + 1
fi
```

En el ejemplo, si el valor de la variable *i* es menor al de *n* se ejecutará el bloque de código dentro del condicional. Caso contrario, las sentencias que se encuentran dentro del *if* no serán ejecutadas.

Para la sentencia condicional básica se usa la palabra clave **if** seguida de una expresión booleana (también denominada *guarda*), sucedida por la palabra clave **then** y finalmente las instrucciones a ejecutar en el caso que la condición se cumpla. El bloque condicional finaliza con la palabra clave **fi**.

Además de la sentencia condicional compacta que vimos anteriormente, el lenguaje posee otros tipos más complejos de construcciones condicionales. Para el caso de querer realizar una división del flujo de ejecución más refinada se pueden especificar múltiples guardas utilizando **elif** e incluso se puede hacer uso de la palabra clave **else** para ejecutar un bloque de código cuando todas las demás guardas resultaron falsas.

```
var menor, igual, mayor: bool
menor := False
igual := False
mayor := False
if i < j then
  menor := True
elif i > j then
  mayor := True
else
  igual := True
fi
```

En el ejemplo se evalúa si la variable i es menor, mayor o igual a la variable j. Debido a que la ejecución del programa es secuencial se ejecutará el bloque de código cuya guarda sea la primera en ser satisfecha. Luego de esto, el programa saltará al final del condicional ignorando todas las otras guardas con sus respectivas instrucciones.

7.2.3. Sentencias Iteradoras

Las sentencias más importantes de un lenguaje imperativo son las que permiten ejecutar un conjunto de instrucciones una cantidad finita o infinita de veces. ΔΔLang ofrece la instrucción **while**, para iterar en base al valor de verdad de una condición, y una variedad de instrucciones **for**, para ejecutar ciclos de instrucciones en un rango de valores determinados.

```
var a: array[1..5] of int
var i: int
i := 1
while i <= 5 do
  a[i] := 1
  i := i + 1
od
```

El ejemplo inicializa los valores del arreglo en *1*. Esto lo hace, creando una variable i que toma distintos valores en el conjunto de índices del arreglo a e irá inicializando los valores del mismo. Notar que la sintaxis de esta instrucción consiste de la palabra clave **while** seguida de una expresión booleana (también denominada *guarda*), y luego un bloque de código encerrado entre las palabras claves **do** y **od**.

En otras situaciones, interesa ejecutar un conjunto de sentencias para distintos valores de una variable en un rango determinado. Es decir, si se quiere realizar de forma reiterativa ciertos cambios de estados conociendo de antemano la cantidad de veces que queremos iterar.

```
var a: array[1..5] of int
for i := 1 to 5 do
  a[i] := 1
od
```

Notar que con la sentencia **for**, uno puede especificar los límites de valores que puede tomar cierta variable dentro de su bloque. Hay una serie de consideraciones a tener en cuenta al utilizar este tipo de instrucción.

- La variable a iterar es declarada de forma implícita en el mismo **for**, por lo que no es necesario especificar su tipo. Esta característica puede ser modificada en lo que queda de desarrollo del intérprete.
- El alcance (*scope*) estará restringido al bloque de la sentencia. Esto significa que una vez finalizada la ejecución de la instrucción la variable i dejará de ser accesible.

- En el cuerpo del **for** no se modificará el valor de la variable a iterar. La alteración del estado de i solo se realiza de forma implícita al finalizar cada iteración.

La anterior no es la única versión de la sentencia **for**. A diferencia del pseudocódigo, la especificación de cada una de las mismas está definida de forma precisa. Las consideraciones anteriormente nombradas siguen aplicando a cada una de las versiones.

```
var a: array[1..5] of int
for i := 5 downto 1 do
  a[i] := 1
od
```

La semántica de este ejemplo es la misma del ejemplo anterior. La particularidad es que se puede especificar un rango descendente de valores para la variable de control haciendo uso de la palabra clave **downto**. Anteriormente, i tomaba valores de forma ascendente.

Otra característica muy interesante, es que la instrucción **for** permite definir cualquier tipo enumerable para la variable iteradora. Esto permite que los límites de valores para la variable de control no sean estrictamente numéricos. Todo conjunto de valores que pueda ser enumerado puede ser utilizado para especificar los límites de la iteración.

```
var b: array['a'..'e'] of int
for k := 'a' to 'e' do
  b[k] := 1
od
```

La otra versión de la instrucción **for**, es la que permite recorrer elementos *iterables*. Todavía queda pendiente debatir como se implementará el funcionamiento de esta construcción en el lenguaje. La idea, es que uno pueda especificar cierto tipo de dato que presente la capacidad de ser recorrido de forma progresiva en lugar de tener que detallar los límites de la iteración. De esta forma, uno podría hacer algo como en el siguiente ejemplo.

```
var sumatoria: int
sumatoria := 0
for x in a do
  sumatoria := sumatoria + x
od
```

Es muy común cuando trabajamos con arreglos, o algún otra construcción iterable, acceder a los diferentes valores del mismo y operar luego con estos. Asumiendo que a es un arreglo de enteros. Este ejemplo suma todos los elementos del mismo. Dado que solo nos interesan los valores que almacena el arreglo, podemos calcular la sumatoria iterando directamente sobre el arreglo, ignorando de esta manera los índices del mismo.

8. Funciones

Para definir rutinas determinísticas, es decir, secuencias de instrucciones que no dependen del estado del programa, ΔΔLang ofrece la posibilidad de especificar funciones. Las mismas realizan una computación, en base a un conjunto de parámetros, y devuelven un resultado. Las funciones son independientes del estado del programa, en el sentido que su comportamiento es determinado por los valores de entrada que recibe. Es importante notar que no modifican el estado de las variables que son pasadas como parámetros.

```
fun factorial (n: int) ret fact: int
  fact := 1
  for i := 1 to n do
    fact := fact * i
  od
end fun
```

En el ejemplo, se especifica una función que calcula el factorial de un número entero *n*. La variable *i* tomará distintos valores de *1* hasta *n*, mientras que en la variable *fact* se irá almacenando la productoria de estos números.

Para la sintaxis, se usa la palabra clave **fun** seguida del nombre de la función. Luego, entre paréntesis se especifican los parámetros separados por comas. Se tienen que detallar los tipos y los identificadores para cada una de las entradas que necesitará la función para su cómputo. También se suelen denominar *parámetros formales*. Con la palabra clave **ret** se especifica el valor de retorno, además del nombre y tipo de la variable asociada al mismo. Finalmente, en el cuerpo de la función se pueden usar todas las sentencias y expresiones que vimos hasta ahora para escribir el programa deseado. Para cerrar el bloque de la función, se utiliza **end fun**.

```
fun existePar (a: array[1..n] of int) ret b: bool
  b := False
  for x := 1 to n do
    b := b || a[x] % 2 == 0
  od
end fun
```

En este último ejemplo se ilustra la posibilidad de definir funciones que trabajan con arreglos cuyos límites conforman un parámetro extra. En el mismo, se utilizan los límites variables que permiten definir métodos más genéricos. Esta clase de parámetros, al igual que los demás, no pueden ser modificados en el cuerpo de la función.

Las funciones también pueden ser definidas de forma recursiva. Es decir, que en el mismo cuerpo de la función se puede llamar a sí misma para continuar con la computación. En el siguiente ejemplo se vuelve a implementar el *factorial*, pero ahora haciendo uso de la recursión. Se define el caso base, cuando la entrada es menor o igual a *1*, y el caso recursivo, que abarca las demás posibilidades.

```
fun factorial (n: int) ret fact: int
```

```
  fact := 1
  if n > 1 then
    fact := n * factorial(n - 1)
  fi
end fun
```

Finalmente, una última característica importante de las funciones es la definición polimórfica. En la misma, uno no especifica de manera concreta los tipos de las variables sino que utiliza una variable de tipo para poder crear una función cuya implementación pueda ser usada por más de un tipo de valor.

```
fun indiceMinimo (a: array[1..n] of T) ret min: int
  min := 1
  for i := 1 to n do
    if a[i] < a[min] then
      min := i
    fi
  od
end fun
```

La función devuelve el índice en el arreglo del valor más chico. Se puede ver que se utiliza la variable de tipo **T** para especificar que la función puede tomar valores de cualquier tipo. En particular, esta función puede ser usada para encontrar el índice del mínimo de un arreglo de enteros, caracteres, entre otros. Todo esto, siempre y cuando la comparación esté definida para el conjunto de valores del tipo variable.

Una salvedad importante sobre esta característica, pendiente a debatir durante la implementación del *type checker*, es que tan permisivo vamos a ser sobre las variables de tipo. En base al último ejemplo se puede ver que no siempre pueden tomar cualquier tipo concreto, por lo que debemos decidir si preferimos un lenguaje más restrictivo que no permita la interpretación de este tipo de programas; o uno que puede fallar a la hora de la evaluación de una expresión debido al uso de variables de tipos incorrectas.

9. Procedimientos

Una construcción similar a las funciones son los procedimientos. Ambas representan formas de especificar rutinas reusables para realizar tareas particulares. Pero la diferencia fundamental entre ambos es que los procedimientos pueden modificar el estado del programa en su accionar. En base a un conjunto de parámetros, algunos de entrada y otros de salida, un procedimiento realiza una computación que modificará el entorno del proceso que lo llamó.

```
proc initArray (out a: array [n..m] of int)
  for i := n to m do
    a[i] := 0
  od
end proc
```

Este ejemplo, muestra un programa donde se define un procedimiento *initArray* que inicializa un arreglo de enteros con el valor cero. Mientras que una función solo devuelve un valor, un procedimiento modifica variables globales que fueron definidas fuera del mismo, en este caso, un arreglo.

La sintaxis de un procedimiento es similar al de una función salvo por el uso de las palabras claves **proc** y **end proc** que encierran al mismo, y la especificación de entradas y salidas. Para cada parámetro formal del procedimiento hay tres opciones:

- **in** determina que las variables de entrada serán utilizadas solo para lectura. Por lo tanto, su valor no será modificado pero si se utilizará para alguna evaluación.
- **out** significa que el valor asociado a la variable será alterado por la llamada al procedimiento. Su valor no será utilizado para la evaluación de una expresión.
- **in / out** establece que la variable va a ser empleada para lectura y escritura. Su valor inicial determinará la computación, pero el mismo será modificado a lo largo del procedimiento.

Al igual que en las funciones, los procedimientos permiten polimorfismo de la misma forma. En el siguiente ejemplo, se modifica un arreglo permutando los valores en las posiciones *i* y *j*. Notar que la variable de tipo **T**, se utiliza tanto en la especificación de parámetros como en la variable temporal dentro del procedimiento.

```
proc swap (in/out a: array[n..m] of T, in i, j: int)
  var tmp: T
  tmp := a[i]
  a[i] := a[j]
  a[j] := tmp
end proc
```

Notar que el uso de la variable de tipo no produce ninguna clase de conflicto en este ejemplo. A diferencia del código dado para funciones, no se aplica ninguna operación sobre un valor del tipo variable, por lo que el *type checker* podría aceptar este programa sin ningún tipo de advertencia.

Finalmente, y similar a las funciones, se permite la definición de procedimientos recursivos. A continuación especificaremos un fragmento de un algoritmo de ordenación, *mergeSort*. El algoritmo se encarga de dividir el arreglo en dos mitades, y ordenar cada una de forma independiente. Una vez realizado esto, une ambas partes para obtener el arreglo ordenado.

```
proc mergeR (in/out a: array[1..n] of T, in l, r: int)
  var m: int
  if r > l then
    m := (r + l) % 2
```

```
    mergeR(a, l, m)
    mergeR(a, m+1, r)
    merge(a, l, m, r)
  fi
end proc
```

10. Programas

Una vez definidas las construcciones anteriores solo resta describir como se especifica un programa en ΔΔLang. En el pseudocódigo este aspecto era tratado con libertad, permitiendo algunas ambigüedades. En la implementación elegimos un formato estructurado para facilitar la comprensión del código.

Un programa, esta conformado por dos partes. La primera es una serie de definiciones de tipo, donde se crean los tipos de datos con los que se desea trabajar. La segunda es una serie de métodos, ya sean procedimientos y/o funciones, que implementan los algoritmos deseados.

Para ejemplificar, vamos a implementar uno de los primeros algoritmos de ordenación vistos en la materia *selectionSort*. Algunas de las componentes del programa ya se ilustraron en secciones anteriores, levemente modificadas.

```
proc swap (in/out a: array[1..n] of T, in i, j: int)
  var tmp: T
  tmp := a[i]
  a[i] := a[j]
  a[j] := tmp
end proc

fun minPos (a: array[1..n] of T, i: int) ret mp: int
  mp := i
  for j := i + 1 to n do
    if a[j] < a[mp] then
      mp := j
    fi
  od
end fun

proc selectionSort (in/out a: array[1..n] of T)
  var minp: int
  for i := 1 to n-1 do
    minp := minPos(a, i)
    swap(a, i, minp)
  od
end proc
```

En el ejemplo se define el procedimiento *swap* que se encarga de intercambiar los valores de dos posiciones de un arreglo. La función *minPos* calcula la posición

del valor más chico entre un índice intermedio y el índice final del arreglo. Finalmente, el procedimiento *selectionSort* es el encargado de ordenar el arreglo llamando a los dos métodos auxiliares anteriores.

Inicialmente, no hay ninguna restricción sobre el orden en el que deben ser especificados los distintos métodos de un programa. Posiblemente, una vez comenzada la implementación de los chequeos estáticos se tomé la decisión de limitar la definición de métodos auxiliares antes de su llamada en el programa.

A continuación, un último ejemplo donde se muestra como se definen nuevos tipos y se trabajan con los mismos en un programa de ΔΔLang. En el mismo, se creó un tipo de dato *counter* el cual se utiliza justamente para contar. Posee los procedimientos para inicializarlo *init*, para aumentarlo *inc* y decrementarlo *dec*. También se define la función que pregunta si es inicial *isInit*.

```
type counter = int

proc init (out c: counter)
  c := 0
end proc

proc inc (in/out c: counter)
  c := c + 1
end proc

proc dec (in/out c: counter)
  c := c - 1
end proc

fun isInit (c: counter) ret b: bool
  b := c == 0
end fun
```

Otra vez, no hay ninguna limitación en el orden en el que deben ser definidos los nuevos tipos de datos. Aunque todavía resta por debatir esta característica. La única restricción en la creación de programas es que la especificación de tipos debe ser previa a cualquier definición de método.