# $\Delta\Delta$Lang

Implementación de un lenguaje *Pascal*-Like

**Matías Federico Gobbi**


Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba
30 de mayo de 2020

# Resumen

Este trabajo consiste en el diseño e implementación de un intérprete para un lenguaje de programación estructurado basado en el lenguaje *Pascal*, orientado al aprendizaje de algoritmos y estructura de datos. El mismo es utilizado actualmente en una materia de la facultad, contando con una definición informal. Existe una sintaxis concreta relativamente consolidada aunque no especificada, y la semántica está definida de manera intuitiva. Siguiendo un modelo de desarrollo en cascada, analizamos la información disponible a partir del dictado de la materia obteniendo una definición formal de la sintaxis abstracta. Luego diseñamos una sucesión de chequeos estáticos, como el sistema de tipos. Finalmente definimos una semántica *small step* a partir de la cual implementamos un intérprete interactivo en el lenguaje *Haskell*.

# Índice general

# Capítulo 1

# Introducción

En el siguiente trabajo, desarrollaremos un lenguaje de programación para la materia *Algoritmos y Estructura de Datos II*. Antes de comenzar propiamente con la definición formal e implementación del mismo, lo correcto es presentar los objetivos que nos hemos impuesto para la creación del lenguaje, y las motivaciones que nos han impulsado al desarrollo de este proyecto. Por lo tanto, la siguiente sección introducirá al lector los distintos aspectos que influyeron la formación del lenguaje, y justificaron la realización de este trabajo.

## 1.1. Motivación

En la materia *Algoritmos y Estructura de Datos II*, durante varios años, se ha utilizado un pseudocódigo para la enseñanza de los distintos conceptos que se estudian en la misma. Debido a esto, el lenguaje que diseñamos (basado en este pseudocódigo) tendrá un fin didáctico, y busca ser otra fuente de aprendizaje para auxiliar el dictado de la asignatura. Los ejes principales de la materia consisten en el análisis de algoritmos, la definición de tipos abstractos de datos, y la comprensión de diversas técnicas de programación.

El objetivo del pseudocódigo es poder introducir a los estudiantes a nuevos conceptos y fomentar buenas prácticas de programación. Se utiliza para describir de forma precisa principios operacionales de los distintos algoritmos estudiados en la materia. Típicamente, se omiten detalles esenciales para la implementación de los algoritmos para favorecer el entendimiento de los mismos. No existe ningún estándar para la sintaxis o semántica del pseudocódigo, por lo que un programa en este pseudo-lenguaje no es ejecutable en el sentido que no puede ser traducido a una serie de instrucciones máquina.

Nuestra meta final con este proyecto, es poder tomar la totalidad de los fragmentos de pseudocódigo que se encuentran dispersos en los diversos contenidos de la materia, y transformarlos en un lenguaje completamente implementado, con todo lo que esto implica. Obviamente, nuestro principal desafío para cumplir nuestra tarea, es poder resolver las distintas ambigüedades y la falta de especificación que el actual pseudocódigo presenta.

### 1.1.1. Objetivos de la Asignatura

Durante el desarrollo de la materia, se pretende que el alumno adquiera diversos conceptos relacionados con los distintos temas estudiados en la asignatura. Algunos de los mismos son listados a continuación:

- Capacidad para comprender y describir el problema que resuelve un algoritmo (el *qué*), y diferenciarlo de la manera en que lo resuelve (el *cómo*).
- Suficiencia para analizar algoritmos, compararlos según su eficiencia en tiempo de ejecución y en espacio de almacenamiento.
- Hábito de identificar abstracciones relevantes al abordar un problema computacional, y aptitud para la especificación e implementación de las mismas.
- Familiaridad con técnicas de diseño de algoritmos de uso frecuente, y comprensión de diversos algoritmos conocidos.
- Contacto con la programación (principalmente en el lenguaje $C$) de algoritmos y estructura de datos.
- Aptitud para la utilización de diversos niveles de abstracción y adaptación a distintos lenguajes de programación.

### 1.1.2. Programa de la Asignatura

El contenido de la materia se puede dividir en tres unidades. En cada una de estas, se introducen nuevos conceptos que luego se reflejan en diversos fragmentos de pseudocódigo empleados para facilitar la comprensión de los mismos. En el diseño del futuro lenguaje, se deberán tener en cuenta todas estas cuestiones para poder crear una herramienta útil para complementar la enseñanza de la asignatura.

#### Análisis de Algoritmos

La primer unidad de la materia, se basa en el análisis de algoritmos. Inicialmente, se estudian distintas maneras de ordenar arreglos utilizando diversas técnicas, como *ordenación por selección*, *ordenación por inserción*, *ordenación por intercalación*, *ordenación rápida*, entra otras. Con estos contenidos básicos presentes, se enseña al alumno a contar operaciones de un programa, introduciendo de esta forma los conceptos de *orden* y *jerarquía* sobre la complejidad de un algoritmo. Para terminar, se introducen las recurrencias *divide y vencerás*, y se presentan otros algoritmos como la *búsqueda lineal*, y la *búsqueda binaria*.

#### Estructura de Datos

La segunda parte de la materia, presenta la noción de estructuras de datos. Se describen a los *tipos concretos* como un concepto relativo a un lenguaje de programación, donde se estudian elementos como los arreglos, las listas, los registros, y los tipos enumerados. Mientras, los *tipos abstractos* se presentan como una idea asociada a un problema que se quiere resolver. Se describe la diferencia entre la *especificación*, y la *implementación* de los mismos, y se enseña la importancia de la elección adecuada para estos. Se examinan diseños distintos para varios de los diversos *TAD's*, como el *contador*, la *pila*, la *cola*, y el *árbol*, junto con la eficiencia en tiempo o espacio de sus distintas operaciones. Además, se introduce el concepto de *manejo dinámico de memoria* de un programa mediante el uso de punteros.

**Algoritmos Avanzados**

La última unidad, presenta distintas estrategias conocidas para la resolución de problemas algorítmicos. Se introduce el esquema general de los *algoritmos voraces*, y se enseñan diversos algoritmos que se basan en esta idea como el de *Dijkstra*, *Prim*, y *Kruskal*. También se introduce al concepto de *backtracking*, resolviendo los problemas de la *moneda*, y la *mochila*, entre otros. Luego, se ve *programación dinámica* donde se visitan problemas previos, además de ver nuevos conceptos como el algoritmo de *Floyd*. Un último tema que se enseña en la materia, es la recorrida de grafos, y las distintas variantes para realizar la misma.

## 1.2. Características del Lenguaje

Una vez detallados los fundamentos en los que se basa el lenguaje, es hora de describir sus características más importantes. ΔΔLang es una formalización del pseudocódigo utilizado en la materia *Algoritmos y Estructura de Datos II*, por lo que está diseñado para enseñar conceptos fundamentales de forma clara y natural. Es un lenguaje imperativo similar a *Pascal*. Este último fue diseñado por *Niklaus Wirth* cerca de 1970 [1]. Algunos elementos básicos que comparten son:

- Una sintaxis verbosa, pero fácil de leer.
- Un tipado fuerte para las expresiones.
- Un formato estructurado del código.

Al solo contar con una definición informal e incompleta del pseudocódigo, tuvimos que enfrentarnos a problemas de ambigüedad y falta de especificación para la creación del lenguaje. Debido a esto, la transición de uno a otro puede no ser inmediata. De todas formas, la esencia de ambos es la misma. La sintaxis del lenguaje es rigurosa, en comparación al código de la materia que era flexible en este aspecto. La semántica del primero está definida de forma precisa, a diferencia del pseudocódigo que solo se contaba con la intuición del lector para interpretar la misma.

### 1.2.1. Tipado

Como mencionamos previamente, ΔΔLang posee tipado fuerte. Esto significa que no se permiten violaciones de los tipos de datos, es decir, dado el valor de una variable de un tipo determinado, no se puede usar la misma como si fuera de otro tipo distinto al especificado en el programa. En una primera instancia, no hay ninguna especie de conversión, implícita o explícita, para los tipos de las expresiones del lenguaje.

En ΔΔLang se ofrecen una serie de tipos nativos un poco más limitada a la utilizada en el pseudocódigo de la materia. Los mismos, a su vez, se pueden dividir en tipos básicos y en tipos estructurados. Al mismo tiempo, existe la posibilidad de definir nuevos tipos de datos en el lenguaje; aspecto fundamental para el desarrollo de la segunda unidad de la asignatura.

Los tipos nativos básicos del lenguaje son los enteros, los reales, los booleanos y los caracteres. Para manipular valores de estos tipos, se ofrecen las operaciones aritméticas y lógicas típicas. A su vez, también se encuentran definidas las operaciones de igualdad y orden para estos valores, a pesar que su aplicación no se limita solo a los mismos.

Por otro lado, los tipos estructurados incorporados son los arreglos y los punteros. Los primeros tendrán un funcionamiento similar a los especificados en el lenguaje *C*. Además, existirá la posibilidad de definir arreglos multidimensionales, y arreglos cuyos tamaños serán variables. Los segundos, permitirán el manejo dinámico de la memoria de un programa y serán útiles para la creación de nuevos tipos de datos.

Finalmente, el usuario podrá crear sus propios tipos de datos. Hay tres posibilidades para la declaración de estos. Los tipos enumerados representarán una enumeración de un conjunto finito de valores. Los sinónimos serán un renombrado de un tipo ya existente. Y las tuplas permitirán la creación de estructuras con múltiples campos. Todos estos elementos serán fundamentales para el estudio de los *TAD's* en la materia.

Similar a *Haskell*, en el lenguaje existen ciertas clases predefinidas que caracterizan el comportamiento de los tipos que las implementan. Estas son **Eq**, **Ord**, e **Iter**. La primera, será implementada por todos los tipos que se pueden igualar. La segunda, es satisfecha por las categorías de elementos que son ordenables. Finalmente, la última clase indica si cierta estructura puede ser recorrida de forma iterativa. Una vez que el usuario declara un tipo, puede implementar todas las operaciones que una determinada clase requiera, para convertir a su nueva estructura de datos en una instancia de la misma.

### 1.2.2. Polimorfismo

El lenguaje permite la declaración de funciones y procedimientos con polimorfismo paramétrico. Esto significa que con una única definición, independiente de los tipos específicos de las entradas polimórficas, las funciones y procedimientos tendrán la capacidad de ser aplicables a argumentos con valores de distinto tipo. A su vez, esto introduce la capacidad de trabajar con variables de tipo en el programa, cuyos tipos concretos serán resueltos en tiempo de ejecución.

Otra posibilidad que permite el lenguaje, es agregar restricciones de clases que deberán ser satisfechas por las variables de tipo que los procedimientos y funciones introducen en su prototipo. Con este refinamiento, uno puede abstraerse de los tipos específicos en la implementación, para solo considerar las operaciones básicas que los mismos proporcionan. De esta forma, podemos limitarnos a trabajar con valores que ofrecen ciertas propiedades como la de igualdad, la de orden, y la de ser iterables.

El polimorfismo que admiten las funciones y procedimientos del lenguaje no se limita solo a tipos. También existe una especie de polimorfismo para los tamaños de arreglos que se introducen en la declaración de los anteriores. Con esta posibilidad, se pueden definir funciones o procedimientos que operan sobre arreglos, independientemente del tamaño de estos. Esto introduce la capacidad de trabajar con tamaños variables de arreglos, cuyo tamaño concreto será resuelto durante la ejecución del programa.

### 1.2.3. Recursión

La recursión es otro de los temas fundamentales en el dictado de la materia. En particular, para la parte de conteo de operaciones, donde se estudia como calcular el *orden* de un algoritmo. Dentro del cuerpo de una función o procedimiento, se puede realizar una llamada recursiva a si mismo para continuar con la ejecución del programa. En estas situaciones, el lenguaje no presenta ninguna particularidad relevante como para ser mencionada en esta sección.

En cambio, donde si haremos una salvedad, es en la declaración de tipos de datos. En el lenguaje, solo se permite una clase de recursión muy limitada para los mismos. Para crear un tipo recursivo, solo hay una posibilidad bastante restrictiva, y es en la definición de una tupla que posea un campo de tipo puntero. Esta especie de recursión es lo suficientemente expresiva como para permitir la implementación de listas enlazadas en el lenguaje, concepto fundamental en el programa de la asignatura.

### 1.2.4. Manejo de Memoria

Una característica muy importante del pseudocódigo, que se sigue manteniendo en el lenguaje, es el manejo dinámico de memoria. Durante la segunda y tercera parte de la materia, este es un concepto central que acompaña al desarrollo de la asignatura. Para la implementación de *TAD's*, resulta un tema recurrente que sirve para comparar distintos diseños en base a su claridad, portabilidad, y eficiencia.

Mediante el uso de punteros, y la llamada de los procedimientos especiales **alloc** y **free**, el usuario puede hacer un uso explícito sobre la memoria utilizada por el programa. Con algunos conceptos similares a *C*, uno puede reservar memoria que será accesible mediante el uso de punteros. Durante la ejecución del programa, se podrá manipular la memoria reservada y, cuando ya no sea necesaria, se podrá liberar la misma.

### 1.2.5. Encapsulamiento

Una última característica relevante que fue tomada de los contenidos de la materia y sentó las bases para el desarrollo del lenguaje, es el encapsulamiento. En el pseudocódigo esta particularidad muchas veces era omitida, debido que un programa en este pseudo-lenguaje solo se podía ejecutar *en el aire*. Debido a esto, se contaba con una intuición sobre que era parte de la especificación y que era parte de la implementación de un tipo de dato, de manera informal.

Cuando pasamos al lenguaje formal, hay una evidente diferenciación entre el código que se utiliza para especificar un *TAD*, y el código empleado para su implementación. En $\Delta\Delta$Lang se busca tener una clara separación en módulos para los elementos definidos de un programa. De esta forma, podemos abstraernos de los detalles propios de la implementación de una estructura de datos, y basarnos solo en su especificación para resolver cierto problema algorítmico.

## 1.3. Desarrollo del Intérprete

Debido que el objetivo de este proyecto es la creación de un lenguaje de programación, nuestro ideal es que el producto final del trabajo sea la implementación de un intérprete para el mismo. El desarrollo de esta herramienta no es una actividad trivial, por el contrario, su avance requerirá de múltiples iteraciones y se dividirá en distintas etapas donde, a medida que se progrese en el proyecto, habrá una retroalimentación mutua entre las mismas. Con respecto a este trabajo de tesis en particular, realizaremos la implementación para la primera versión de la fase de *análisis* del intérprete. En la misma, luego de definir la sintaxis del lenguaje, desarrollaremos tanto el parser como los chequeos estáticos que conforman esta etapa.

A continuación, daremos un breve marco teórico sobre distintas cuestiones que consideramos importante mencionar sobre el diseño del intérprete en general. Para el mismo, nos basaremos en los primeros capítulos de la bibliografía de Aho, Sethi y Ullman [2].


Figura 1.1: Un compilador

Un lenguaje de programación es una notación para describir computaciones a personas y a máquinas. Pero para que un programa sea ejecutable, antes debe ser traducido a un formato comprensible para una máquina. Los sistemas de software que se encargan de esta traducción son denominados *compiladores*. De forma simple, un compilador es un programa que puede leer

un programa especificado en un *lenguaje fuente* y traducirlo en un programa equivalente en un *lenguaje objetivo*. En la imagen 1.1 se puede observar un esquema simplificado de esta idea.

Un intérprete es otra clase común de procesador de lenguajes. En lugar de producir un programa objetivo como resultado de una traducción, un intérprete simula ejecutar directamente las operaciones especificadas en el programa fuente, en base a las entradas suministradas por el usuario y retornando las salidas producidas como resultado de la ejecución. En la figura 1.2 se puede apreciar la diferencia esencial entre ambas clases de procesadores de lenguajes.


Figura 1.2: Un intérprete

Si entramos un poco más en detalle, podemos observar que el proceso de transformación está compuesto por dos etapas: *análisis* y *síntesis*. En el caso de la primera etapa, su desarrollo puede realizarse de manera idéntica tanto para compiladores como para intérpretes. En cambio, la segunda presenta diferencias sustanciales entre ambos. Debido que nuestro objetivo es implementar un intérprete para el lenguaje, nos concentraremos solo en el estudio de este.

### 1.3.1. Análisis

La parte del análisis divide el programa fuente en distintas piezas e impone una estructura gramatical a las mismas. Luego, utiliza esta estructura para crear una representación intermedia del programa fuente. Si la etapa de análisis detecta que el programa presenta errores sintácticos o incoherencias semánticas, entonces deberá proveer mensajes informativos para que el usuario pueda aplicar las correcciones adecuadas. En nuestro caso, la representación intermedia que utilizaremos serán los *arboles de sintaxis abstracta*. La transformación del programa fuente a nuestra representación intermedia, no es trivial. Para facilitar la misma, comúnmente se divide esta tarea en varias fases.

#### Análisis Léxico

En esta etapa, también llamada *fase de escaneo*, se analiza la entrada carácter por carácter y se divide la misma en una serie de unidades elementales denominadas *componentes léxicos*. Por cada componente léxico, la fase de escaneo produce como salida un *token* que pertenece a cierta categoría gramatical y posee una cantidad determinada de atributos con información relevante para las siguientes fases de análisis. En esta etapa, además, se filtran elementos como los espacios en blanco y los comentarios.

#### Análisis Sintáctico

Esta es la fase que comúnmente se denomina *parser*. El parser utiliza los tokens obtenidos en la etapa previa, para crear una representación intermedia de la estructura gramatical del flujo total de tokens. Como mencionamos anteriormente, nosotros utilizaremos un *árbol de sintaxis abstracta* como representación. Las fases posteriores del intérprete emplearán esta estructura gramatical para continuar el análisis del programa fuente.

#### Análisis Semántico

La última etapa del análisis es la de *chequeos estáticos*. La misma se encarga de verificar si las restricciones semánticas impuestas en la definición del lenguaje son respetadas. Una parte importante de este análisis es el llamado chequeo de tipos (*typecheck*). Comúnmente, esta fase toma como entrada la representación intermedia obtenida en la etapa previa, y le agrega las anotaciones de tipos adecuadas, necesarias para continuar el análisis en etapas posteriores.

### 1.3.2. Síntesis

La parte de síntesis de un compilador es muy diferente a la de un intérprete. Para el primero, tenemos que construir el programa objetivo utilizando la representación intermedia, junto con toda la información adicional recopilada en las etapas previas. Habitualmente, esta fase se divide en otras dos partes, la *generación de código intermedio* y la *generación de código objeto*. En la *generación de código intermedio* se obtiene una representación independiente de la máquina, pero fácilmente traducible a lenguaje ensamblador. En cambio, la *generación de código objeto* es totalmente dependiente de la arquitectura concreta para la que se esté desarrollando el compilador. Además, durante estas fases comúnmente se aplica algún proceso de optimización sobre el código generado.

Del otro lado, como el objetivo de un intérprete difiere con el de un compilador, sus etapas de síntesis también lo hacen en la misma manera. Para obtener los resultados del programa, debemos partir del *árbol de sintaxis abstracta* obtenido luego de la fase de *análisis* del intérprete y, junto con los datos de entrada sustentados por el usuario, simular la ejecución del programa en base a las acciones especificadas en el código del mismo. De esta forma, una vez finalizada la ejecución, se obtienen las salidas del programa. En la imagen 1.3 se puede observar la estructura de nuestro intérprete.

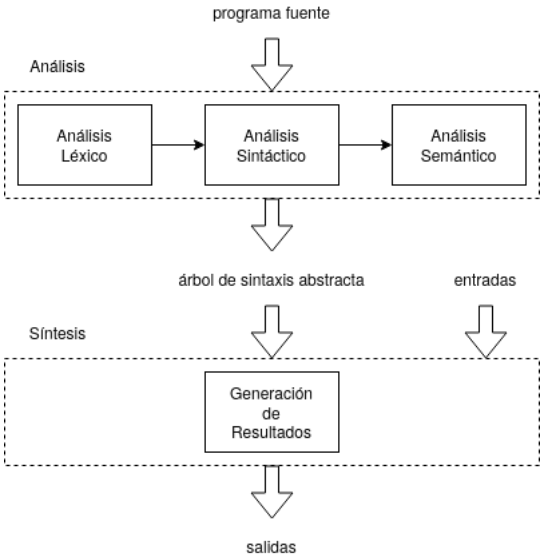


Figura 1.3: Estructura de un intérprete

# Capítulo 2

# Sobre el lenguaje

# Capítulo 3

# Sobre el parser

En el siguiente capítulo, nos dedicaremos principalmente al desarrollo del parser para nuestro intérprete. Lo primero a realizar, será precisar formalmente la sintaxis de nuestro lenguaje. Una vez definida, comenzaremos con la descripción de los aspectos principales sobre la implementación del parser de ΔΔLang. Idealmente, esta sección del trabajo será utilizada para la documentación del futuro intérprete.

## 3.1. Sintaxis Abstracta

La sintaxis del lenguaje ya se expuso de manera informal en capítulos anteriores, mediante los distintos ejemplos que se fueron ilustrando. A pesar de esto, si se quisiera hacer un estudio formal del lenguaje, lo correcto sería dar una definición precisa de la misma. Por lo tanto, a continuación se describirá la *sintaxis abstracta* de ΔΔLang de forma matemática.

### 3.1.1. Expresiones

Una expresión puede adoptar distintas formas; puede ser un valor constante, una llamada a función, una operación sobre otras expresiones o una variable con sus respectivos operadores. Su composición se describe a continuación.

$$\langle expression \rangle \ ::= \ \langle constant \rangle \ | \ \langle functioncall \rangle \ | \ \langle operation \rangle \ | \ \langle variable \rangle$$

A su vez, una constante puede tomar alguno de los siguientes valores. Los no terminales *integer*, *real*, *bool*, y *character* denotan los conjuntos de valores esperados, mientras que *cname* hace referencia a los identificadores de constantes definidas por el usuario. Los terminales **inf** y **null**, representan al infinito y al puntero nulo respectivamente.

$$\langle constant \rangle \ ::= \ \langle integer \rangle \ | \ \langle real \rangle \ | \ \langle bool \rangle \ | \ \langle character \rangle \ | \ \langle cname \rangle \ | \ \textbf{inf} \ | \ \textbf{null}$$

Una llamada a función está compuesta por su nombre y la lista de parámetros que recibe. La misma puede tener una cantidad arbitraria de entradas. Notar que se utilizará la misma clase de identificadores tanto para funciones y procedimientos como para variables.

$$\langle functioncall \rangle \ ::= \ \langle id \rangle \ ( \ \langle expression \rangle \ \ldots \ \langle expression \rangle \ )$$

Los operadores del lenguaje están conformados por operados numéricos, booleanos, y de orden e igualdad. Observar que será necesaria la implementación de un chequeo de tipos para asegurar el uso apropiado de los mismos.

⟨*operation*⟩ ::= ⟨*expression*⟩ ⟨*binary*⟩ ⟨*expression*⟩ | ⟨*unary*⟩ ⟨*expression*⟩

⟨*binary*⟩ ::= + | − | ∗ | / | % | || | && | <= | >= | < | > | == | != 

⟨*unary*⟩ ::= − | !

Finalmente, describimos las variables con sus respectivos operadores. Las mismas pueden simbolizar un único valor, un arreglo de varias dimensiones, una tupla con múltiples campos, o un puntero a otra estructura en memoria. El no terminal *fname* representa el alias de una componente para una estructura de tipo tupla definida por el usuario.

⟨*variable*⟩ ::= ⟨*id*⟩
| ⟨*variable*⟩ [ ⟨*expression*⟩ ... ⟨*expression*⟩ ]
| ⟨*variable*⟩ . ⟨*fname*⟩
| ⋆ ⟨*variable*⟩

### 3.1.2. Sentencias

Las sentencias del lenguaje se dividen en las siguientes instrucciones. La composición de la *asignación* y el *while* es bastante simple, por lo que se detallan también a continuación. Notar que el no terminal *sentences* se utiliza para representar la secuencia de instrucciones.

⟨*sentence*⟩ ::= **skip** | ⟨*assignment*⟩ | ⟨*procedurecall*⟩ | ⟨*if*⟩ | ⟨*while*⟩ | ⟨*for*⟩

⟨*assignment*⟩ ::= ⟨*variable*⟩ := ⟨*expression*⟩

⟨*while*⟩ ::= **while** ⟨*expression*⟩ **do** ⟨*sentences*⟩

⟨*sentences*⟩ ::= ⟨*sentence*⟩ ... ⟨*sentence*⟩

Para la llamada de procedimientos, se utiliza una sintaxis similar a la empleada para funciones. Además de esto, se encuentran definidos dos métodos especiales exclusivamente para el manejo explícito de memoria del programa.

⟨*procedurecall*⟩ ::= ⟨*id*⟩ ( ⟨*expression*⟩ ... ⟨*expression*⟩ )
| **alloc** ⟨*variable*⟩
| **free** ⟨*variable*⟩

La instrucción *if* es bastante compleja en su composición. Para simplificar la sintaxis abstracta, nos limitaremos a solo permitir una única opción para su especificación. Notar que las otras versiones de esta sentencia, utilizadas en capítulos previos, se pueden obtener mediante azúcar sintáctico (*syntax sugar*).

⟨*if*⟩ ::= **if** ⟨*expression*⟩ **then** ⟨*sentences*⟩ **else** ⟨*sentences*⟩

Finalmente, otra instrucción que presenta varias opciones es el *for*. Debido que nos encontramos desarrollando la primer versión del intérprete, algunas funcionalidades deseadas aún no se encuentran definidas. Esto se ve reflejado en la actual instrucción. En la misma, se pueden especificar rangos ascendentes con **to**, o descendentes con **downto**, pero la versión que admite estructuras iterables aún no ha sido determinada.

⟨*for*⟩ ::= **for** ⟨*id*⟩ := ⟨*expression*⟩ **to** ⟨*expression*⟩ **do** ⟨*sentences*⟩
| **for** ⟨*id*⟩ := ⟨*expression*⟩ **downto** ⟨*expression*⟩ **do** ⟨*sentences*⟩

### 3.1.3. Tipos

Los tipos que soporta ΔΔLang pueden dividirse en dos categorías, los nativos del lenguaje y los definidos por el usuario. A su vez, los primeros puede separarse en básicos o estructurados. A continuación se detallan los mismos.

⟨*type*⟩ ::= **int** | **real** | **bool** | **char**
| ⟨*array*⟩
| ⟨*pointer*⟩
| ⟨*definedtype*⟩
| ⟨*typevariable*⟩

Del lado de los tipos nativos estructurados, se tienen a los arreglos y a los punteros. Para los primeros, hay que especificar como se definen los tamaños para sus dimensiones. El no terminal *sname* representa al tamaño polimórfico introducido en el prototipo de funciones y procedimientos. Para los segundos, solo se debe indicar cual es el tipo de valor que se va a referenciar.

⟨*array*⟩ ::= **array** ⟨*size*⟩ ... ⟨*size*⟩ **of** ⟨*type*⟩

⟨*size*⟩ ::= ⟨*natural*⟩ | ⟨*sname*⟩

⟨*pointer*⟩ ::= **pointer** ⟨*type*⟩

En el caso de las variables de tipo, las mismas poseen su propia clase de identificadores. En cambio, para los tipos definidos por el usuario, además de su nombre representado por el no terminal *tname*, se deben detallar los tipos en los cuales se instanciará. Si el mismo no posee argumentos, el terminal **of** no se deberá especificar.

⟨*typevariable*⟩ ::= ⟨*typeid*⟩

⟨*definedtype*⟩ ::= ⟨*tname*⟩ **of** ⟨*type*⟩ ... ⟨*type*⟩

Cuando se declara un procedimiento, es necesario especificar el rol que cumplirá cada una de sus entradas. Esto significa que se debe detallar, para todos los argumentos individualmente, si se emplearán para lectura (**in**), escritura (**out**), o ambas (**in/out**).

⟨*io*⟩ ::= **in** | **out** | **in/out**

También existe una serie de clases predefinidas para los tipos del programa. Las mismas representan una especie de interfaz que caracteriza las propiedades que cumplen cada uno de los tipos que las definen. Debido que aún no se ha precisado formalmente la naturaleza de la clase **Iter**, la misma se omitirá en esta primera versión del intérprete.

⟨*class*⟩ ::= **Eq** | **Ord**

Finalmente, para la declaración de nuevos tipos por parte del usuario hay tres posibilidades. Se pueden crear tipos enumerados, sinónimos de tipos y tuplas. Para los dos últimos, se pueden especificar parámetros de tipos que permiten crear estructuras más abstractas. Similar a lo dicho anteriormente, si los tipos declarados no poseen argumentos, entonces el terminal **of** se omite.

⟨*typedecl*⟩ ::= **enum** ⟨*tname*⟩ = ⟨*cname*⟩ ... ⟨*cname*⟩
| **syn** ⟨*tname*⟩ **of** ⟨*typearguments*⟩ = ⟨*type*⟩
| **tuple** ⟨*tname*⟩ **of** ⟨*typearguments*⟩ = ⟨*field*⟩ ... ⟨*field*⟩

⟨*typearguments*⟩ ::= ⟨*typevariable*⟩ ... ⟨*typevariable*⟩

⟨*field*⟩ ::= ⟨*fname*⟩ : ⟨*type*⟩

### 3.1.4. Programas

Para finalizar con la sintaxis del lenguaje, describiremos como se especifica un programa en el mismo. Un programa está compuesto por una serie de definiciones de tipo, seguidas de una serie de declaraciones de funciones y/o procedimientos.

⟨*program*⟩ ::= ⟨*typedecl*⟩ ... ⟨*typedecl*⟩ ⟨*funprocdecl*⟩ ... ⟨*funprocdecl*⟩

⟨*funprocdecl*⟩ ::= ⟨*function*⟩ | ⟨*procedure*⟩

A continuación, detallamos como se especifica el cuerpo de una función o procedimiento. El mismo está conformado primero por una lista de declaraciones de variables, y segundo por una lista de instrucciones. Para declarar una variable solo se tiene que especificar el identificador de la misma, junto con el tipo que posee. También existe la posibilidad de definir múltiples variables en una sola declaración.

⟨*body*⟩ ::= ⟨*variabledecl*⟩ ... ⟨*variabledecl*⟩ ⟨*sentences*⟩

⟨*variabledecl*⟩ ::= **var** ⟨*id*⟩ ... ⟨*id*⟩ : ⟨*type*⟩

Una función posee un identificador propio, una lista de argumentos, un retorno, y un bloque que conforma su cuerpo. Tanto para los argumentos, como para el retorno, solo se tienen que detallar sus identificadores junto con el tipo del valor que representarán.

⟨*function*⟩ ::= **fun** ⟨*id*⟩ ( ⟨*funargument*⟩ ... ⟨*funargument*⟩ ) **ret** ⟨*funreturn*⟩
**where** ⟨*constraints*⟩
**in** ⟨*body*⟩

⟨*funargument*⟩ ::= ⟨*id*⟩ : ⟨*type*⟩

⟨*funreturn*⟩ ::= ⟨*id*⟩ : ⟨*type*⟩

Un procedimiento posee una estructura muy similar a la de una función. Las dos diferencias fundamentales con esta, es que el primero no posee retorno ya que no producirá ningún valor como resultado, y que sus argumentos deben especificar que clase de uso se hará con los mismos.

⟨*procedure*⟩ ::= **proc** ⟨*id*⟩ ( ⟨*procargument*⟩ ... ⟨*procargument*⟩ )
**where** ⟨*constraints*⟩
**in** ⟨*body*⟩

⟨*procargument*⟩ ::= ⟨*io*⟩ ⟨*id*⟩ : ⟨*type*⟩

Para agregar a lo anterior, debido que los argumentos de una función o procedimiento pueden tener tipo variable, es conveniente poder imponer restricciones a los mismos. De esta forma, se pueden crear funciones y procedimientos más abstractos que funcionen para una gran variedad de tipos, y al mismo tiempo, requerir que los mismos sean instancias de ciertas clases.

$\langle constraints \rangle ::= \langle constraint \rangle \ \ldots \ \langle constraint \rangle$

$\langle constraint \rangle ::= \langle typevariable \rangle : \langle class \rangle \ \ldots \ \langle class \rangle$

## 3.2. Sintaxis Concreta

La *sintaxis concreta* de ΔΔLang, ya se presentó de manera informal en la introducción del lenguaje, y como para el estudio del mismo nos limitaremos a la *sintaxis abstracta*, no entraremos demasiado en detalle en este aspecto. De todas formas, consideramos importante mencionar algunas características que pueden no haber sido detalladas de forma clara en el informe.

### 3.2.1. Identificadores

En el lenguaje hay diversas categorías de identificadores. Los mismos se pueden separar en dos clases, los que comienzan con minúscula (*lower*), y los que comienzan con mayúscula (*upper*). Para el resto de su cuerpo, se tiene la misma estructura en ambos casos. En la primera clase, se encuentran los identificadores de variables, funciones y procedimientos, a los tamaños de arreglos, los alias para campos de tuplas, y por último, los tipos definidos por el usuario. Mientras, la segunda clase está conformada por los identificadores para variables de tipo, y las constantes enumeradas.

$\langle id \rangle ::= \langle lower \rangle \ \langle rest \rangle$

$\langle sname \rangle ::= \langle lower \rangle \ \langle rest \rangle$

$\langle fname \rangle ::= \langle lower \rangle \ \langle rest \rangle$

$\langle tname \rangle ::= \langle lower \rangle \ \langle rest \rangle$

$\langle typeid \rangle ::= \langle upper \rangle \ \langle rest \rangle$

$\langle cname \rangle ::= \langle upper \rangle \ \langle rest \rangle$

Con esto en mente, se puede observar que en base al contexto y al formato del identificador parseado, en la fase de *análisis sintáctico* del intérprete, somos capaces de distinguir a que clase de elemento se está haciendo referencia en el código, a excepción de un caso particular. Dentro de una expresión, no podemos diferenciar al nombre de una variable del identificador utilizado para llamar al tamaño de un arreglo. Esto nos obliga a tener una precaución adicional a la hora de los chequeos estáticos para ser capaces de reconocer a los mismos.

Volviendo a la sintaxis, podemos ser aún más concretos. A continuación describimos los distintos caracteres que pueden conformar un identificador del lenguaje. Para el resto de *componentes léxicos*, como los números o los caracteres literales, su estructura no presenta ninguna particularidad relevante por lo que serán omitidos.

$\langle rest \rangle$ ::= ( $\langle letter \rangle$ | $\langle digit \rangle$ | $\langle other \rangle$ )*

$\langle letter \rangle$ ::= $\langle lower \rangle$ | $\langle upper \rangle$

$\langle lower \rangle$ ::= $a$ | $b$ | ... | $z$

$\langle upper \rangle$ ::= $A$ | $B$ | ... | $z$

$\langle digit \rangle$ ::= $0$ | $1$ | ... | $9$

$\langle other \rangle$ ::= _ | $'$

### 3.2.2. Azúcar Sintáctico

Existen una serie de construcciones en ΔΔLang que son definidas mediante *syntax sugar*. Las mismas pueden ser expresadas utilizando elementos ya presentes en el lenguaje, y no expanden el poder expresivo del mismo. Pero su utilidad radica en que ofrecen una forma sucinta y legible de especificar ciertas operaciones comunes en los algoritmos.

Una notación conveniente para acceder a los campos de una tupla señalada por un puntero es la flecha ($\rightarrow$). De esta forma, en lugar de acceder a la memoria referenciada por un puntero con la estrella ($\star$), y luego consultar uno de los campos de la tupla con el punto (.), uno puede hacer uso de esta abreviatura que resulta más conveniente.

$$v \rightarrow fn \quad \stackrel{def}{=} \quad \star v.fn$$

Otra notación que resulta útil en el lenguaje, es la de agrupar argumentos. Dada la definición de una función o procedimiento, puede ocurrir que varias de sus entradas posean el mismo tipo. En estas ocasiones es conveniente unir todas sus declaraciones en una sola. De esta forma, queda más claro que todos los argumentos agrupados poseen el mismo tipo.

$$\textbf{fun } f \ (\ \ldots\ a_1 : \theta, a_2 : \theta, \ldots, a_n : \theta\ \ \ldots\ )\ \ldots \quad \stackrel{def}{=} \quad \textbf{fun } f \ (\ \ldots\ a_1, a_2, \ldots, a_n : \theta\ \ \ldots\ )\ \ldots$$

Las últimas construcciones utilizadas para escribir código de forma concisa en el lenguaje, son todas las variantes de la sentencia *if*. Anteriormente, solo se describió una única manera para especificar la instrucción. Con las siguientes definiciones, podemos hacer uso de los comandos introducidos en capítulos previos. La primer notación, permite omitir el último bloque de sentencias (*else*). La segunda, nos da la capacidad de agregar una cantidad arbitraria de condicionales adicionales (*elif*).

$$\textbf{if } b \textbf{ then } ss \quad \stackrel{def}{=} \quad \textbf{if } b \textbf{ then } ss \textbf{ else skip}$$

$$\textbf{if } b_1 \textbf{ then } ss_1 \textbf{ elif } b_2 \textbf{ then } ss_2 \textbf{ else } ss_3 \quad \stackrel{def}{=} \quad \textbf{if } b_1 \textbf{ then } ss_1 \textbf{ else if } b_2 \textbf{ then } ss_2 \textbf{ else } ss_3$$

## 3.3. Parser

Ya nos encontramos en condiciones para describir los detalles principales sobre el desarrollo del parser para ΔΔLang. Haremos mención de las decisiones más relevantes tomadas durante la implementación del mismo, las dificultades encontradas en el camino, y algunas limitaciones que debimos resolver.

### 3.3.1. Librerías

Inicialmente, se comenzó utilizando la librería *Parsec* [3]. Esta decisión se tomó debido que algunos de nosotros ya estábamos familiarizados con su uso por proyectos anteriores. Más adelante, en las etapas finales del desarrollo del parser, se decidió migrar el código a *Megaparsec* [4]. Esta transición fue justificada por las limitaciones que presentaba la primera opción frente a la segunda, que además de solucionar algunas de las dificultades de la implementación de forma sencilla, ofrece un rango de funcionalidades más diverso que puede beneficiar al desarrollo futuro del intérprete.

#### Parsec

*Parsec* es una librería para el diseño de un parser monádico, implementada en *Haskell*, y escrita por *Daan Leijen*. Es simple, segura, rápida, y posee buena documentación. Para la etapa inicial de desarrollo, resultó ser una herramienta intuitiva y fácil de manejar. A pesar de esto, para este proyecto en particular, la misma no ofrecía la suficiente flexibilidad y funcionalidad que buscábamos. La totalidad del parser (al menos en esta primera versión del intérprete) fue implementada usando esta librería.

#### Megaparsec

*Megaparsec* se puede considerar el sucesor extraoficial de *Parsec*, escrita por *Mark Karpov*. Partiendo de las bases definidas por esta última, la librería busca ofrecer mayor flexibilidad para la configuración del parser y una generación de mensajes de error más sofisticada respecto a su antecesor. La herramienta resulta familiar para todo el que tenga ciertos conocimientos básicos sobre *Parsec*. La transición de librerías se vio aliviada por esta característica, debido a la semejanza entre ambas.

#### Comparación

Como mencionamos previamente, debido que *Parsec* no resultó ser la herramienta ideal para el desarrollo de nuestro intérprete, se decidió comenzar a utilizar *Megaparsec*. Las razones puntuales que nos llevaron a tomar esta decisión se listan a continuación.

- Análisis Léxico: Ambas librerías implementan un mecanismo sencillo para definir un *analizador léxico*, dentro del mismo parser. De esta forma, se simplifica el diseño del intérprete al unificar estas dos fases fuertemente acopladas. La diferencia entre ambas, radica en el hecho que *Parsec* es demasiado inflexible en este aspecto. Para ciertas cuestiones, se tuvo que redefinir gran parte de la implementación de la librería para poder acomodarla a nuestras necesidades. En cambio, *Megaparsec* no impone ninguna estructura sobre el *analizador léxico*, y solo provee funcionalidades básicas elementales para su definición.

- Mensajes de Error: La generación de mensajes de error en el análisis sintáctico (al igual que en el análisis semántico) es una tarea sumamente importante para el intérprete. La primera herramienta utilizada, ofrecía una forma simple y concisa para el informe de errores. En la misma, se podían especificar cuales eran los *tokens* esperados (o inesperados) por el parser al momento de fallar, junto con el mensaje informativo asociado a esta. A pesar de esto, la segunda opción presenta una generación de errores mucho más desarrollada. Además de conservar las funcionalidades previas, se pueden configurar nuevas clases de errores junto con la forma que los mensajes de error son presentados al usuario.

- Desarrollo Futuro: Una vez que el desarrollo del intérprete se encuentre lo suficientemente avanzado, será necesario volver a esta etapa y adecuarla a las nuevas necesidades que hayan surgido en el camino. En este aspecto, *Megaparsec* ofrece una serie de funcionalidades adicionales que no se encuentran en *Parsec*. Una de ellas es el soporte para múltiples errores, junto con la capacidad de atrapar errores, lo que permite un control mucho más amplio sobre la información que recibe el usuario al analizar su código. También existe la posibilidad de agregar casos de test para aumentar la certeza que el funcionamiento del parser implementado es correcto. Una última característica que puede resultar útil en un futuro, es el parseo *sensible a la indentación* que ofrece la librería.

### 3.3.2. Información de Posición

Una tarea fundamental que debe realizar cualquier compilador, o en nuestro caso intérprete, es informar al usuario sobre los errores sintácticos o semánticos que se hayan detectado durante el análisis y procesamiento de su programa. Debido a esto, la generación de mensajes de error informativos y precisos es una cualidad deseada en esta clase de herramientas, ya que facilitan la corrección de los mismos por parte del programador. Una propiedad que se puede deducir de lo anterior, es que para que un mensaje de error sea adecuado es fundamental que el mismo pueda indicar puntualmente *donde* ocurre este error en el programa.

En este ciclo inicial de desarrollo del intérprete, tanto la fase de *análisis sintáctico* como la de *análisis semántico* podrán encontrar fallas en el código, y deberán informar sobre el problema detectado al usuario. Para la primer etapa, cuando se produce un error durante el parsing de algún elemento sintáctico, la librería *Megaparsec* ya incorpora un mecanismo para señalar la posición en el archivo donde se ha encontrado la falla. Haciendo uso de la misma, conseguimos tener una generación de mensajes de error claros para la fase de *análisis sintáctico*. Cuando avanzamos a etapas posteriores en el intérprete, necesitamos alguna forma de poder vincular las fallas detectadas durante las mismas con la ubicación en el archivo involucrado en el error.

Para solucionar este problema, se adaptó la sintaxis de una forma similar a como lo hace el compilador de *Haskell*, `GHC` [5]. Con esto nos referimos a que todo elemento sintáctico del lenguaje que puede ser relevante en la generación de errores, es envuelto con la información de posición correspondiente a su ocurrencia en el código. De esta forma, se definió en el módulo `Syntax.Located`, el siguiente *datatype* (3.1). Recordar que en esta primera versión del intérprete solo trabajamos con programas definidos en un único módulo, por lo que almacenar solamente las líneas y columnas de inicio y fin de un determinado elemento, es suficiente para generar un mensaje de error preciso.

```
1  -- Parsing Information
2  data Located e = L { info :: Info
3                     , item :: e
4                     }
5
6  -- Position Information
7  data Info = I { sLin :: Line
8                , sCol :: Column
9                , eLin :: Line
10               , eCol :: Column
11               }
```

Listing 3.1: Información de Posición en Parser

A medida que avanza el parser en el análisis de un archivo, cuando se obtiene un elemento sintáctico (por ejemplo, una expresión), se encapsula el mismo con la información de su posición y se almacena en el *árbol de sintaxis abstracta* que se genera como representación intermedia del código. Siendo un poco más puntuales, podemos ver el ejemplo de las expresiones del lenguaje. En `Syntax.Expr` se especifican las mismas de la siguiente forma (3.2). Con esta definición, podemos almacenar tanto la posición de una expresión particular como todas las de sus subexpresiones. Esto tiene como ventaja que en el caso de encontrarse una falla (por ejemplo, un error de tipos) en el código, se puede exhibir toda la *traza* de análisis junto con sus posiciones correspondientes.

```
1 -- Expressions
2 data Expr = Const LConstant
3           | Loc LLocation
4           | UOp UnOp LExpr
5           | BOp BinOp LExpr LExpr
6           | FCall Id [LExpr]
7
8 type LExpr = Located Expr
```

Listing 3.2: Expresiones del Lenguaje

### 3.3.3. Módulos

A continuación, describiremos brevemente los distintos módulos en los que se divide la implementación del parser. Mencionaremos los detalles más relevantes de cada uno de estos elementos, junto con el propósito de los mismos.

En `Parser.Position` se proveen todas las funcionalidades necesarias para poder calcular la ubicación en el archivo del elemento que se intenta parsear. Es utilizado en la mayoría de los módulos para el *análisis sintáctico* del intérprete. Como mencionamos previamente, esta tarea es fundamental para luego poder dar mensajes de errores precisos e informativos al usuario. Haciendo uso de la función `getSourcePos`, provista por la librería, podemos extraer la posición actual del parser y luego, ligarla con el elemento sintáctico correspondiente.

El módulo `Parser.Lexer` es uno de los más importantes del código. Comprende la totalidad del *analizador léxico* del lenguaje. En el mismo, se implementan funciones para parsear todas las clases de identificadores, los valores constantes (como los numéricos, por ejemplo), y las *palabras claves* y operadores de ΔΔLang. Inicialmente, cuando se utilizaba la librería *Parsec*, se tuvo que redefinir la mayoría de las funciones que implementaba debido que las mismas consumían automáticamente todos los *whitespaces* (espacios en blanco, comentarios, saltos de línea, etc.) al parsear un elemento. Esto impedía poder calcular de forma precisa la posición de las distintas estructuras sintácticas del lenguaje. Luego de la transición, debido que *Megaparsec* delega la responsabilidad del consumo de *whitespace* al usuario, se pudo hacer uso de las funciones auxiliares que brinda la librería, y se simplificó el módulo.

En `Parser.Expr` se parsean las diversas expresiones del lenguaje. Una particularidad interesante de este módulo, es el uso de la función `makeExprParser`. Dado un parser de términos, que serían los elementos básicos que conforman una expresión, y una tabla de operadores, donde se debe especificar la asociatividad y precedencia de cada uno, la función construye un parser para expresiones basado en los mismos. En nuestro caso, se hizo uso de este mecanismo para las expresiones y las variables del lenguaje. Para el primero, su implementación es directa debido que es una situación estándar. En cambio, para el segundo, se tuvo que interpretar a las distintas operaciones para el acceso de variables como operadores de expresiones para poder aprovechar la función especificada en la librería.

El módulo `Parser.Statement` se encarga de obtener las sentencias especificadas en el código. A diferencia de la *sintaxis abstracta*, en la implementación del lenguaje se permiten múltiples formas para detallar la instrucción condicional *if*. Para la misma, el componente *else* es opcional, y además, se pueden agregar una cantidad arbitraria de condicionales *elif*. Otra particularidad, es la forma de obtener la ubicación para la asignación. Debido que esta es la única sentencia que no posee un delimitador final, calcular su posición no es una tarea inmediata.

En `Parser.Decl` se parsean todas las declaraciones del lenguaje. Esto involucra diversas construcciones de distinto índole. Se obtienen las definiciones de tipo, los cuales abarcan a las tuplas, los sinónimos y las enumeraciones. También se especifican las declaraciones de instancias de clases para los mismos. Además, se parsean las definiciones de funciones y procedimientos, junto con todos los elementos que las conforman, como sus argumentos y restricciones.

Los últimos archivos no presentan ninguna complejidad adicional. El módulo `Parser.Type` obtiene los distintos tipos admitidos en el lenguaje. En `Parser.Class` se parsean todas las clases predefinidas en el mismo. Y finalmente, `Parser.Program` acepta los programas, sintácticamente válidos, de ΔΔLang.

# Capítulo 4

# Sobre los chequeos

Una vez especificada la sintaxis del lenguaje, junto con la implementación del parser para el intérprete, es hora de comenzar la etapa de *análisis semántico*. En este capítulo, describiremos los distintos chequeos estáticos que el intérprete deberá realizar para tener una mayor certeza que el programa a ejecutar es correcto. También haremos mención de algunos chequeos dinámicos que puede ser conveniente implementar, debido que la totalidad de los errores de un programa no puede ser capturada solo con un análisis estático.

Nuestra tarea entonces, consistirá del diseño e implementación de los distintos chequeos estáticos para nuestro lenguaje. La idea es que el intérprete sea más robusto, y pueda detectar errores en etapas tempranas del desarrollo de un programa. De esta manera, al ejecutar un programa previamente verificado de forma estática, habrá más posibilidades de que su ejecución finalice de forma exitosa. Del otro lado, para las validaciones que serán delegadas al análisis dinámico, solo haremos comentarios breves al respecto ya que la realización de las mismas será un trabajo futuro en el desarrollo del intérprete.

Para dar un formato estructurado a la sección, la misma se organizará en base al orden temporal en el que las distintas validaciones se efectúan en la implementación del intérprete. Se dará una descripción formal para cada uno de los chequeos a realizar, acompañada de una explicación informal para facilitar su comprensión. Idealmente, este capítulo formará parte de la documentación de $\Delta\Delta$Lang.

Los fundamentos teóricos utilizados en esta sección están basados en la bibliografía de Reynolds [6]. En particular, los capítulos sobre el sistema de tipos (`15`), el subtipado (`16`), y el polimorfismo (`18`), son de fundamental importancia para el desarrollo del trabajo. Por el otro lado, la implementación de los chequeos estáticos siguió la idea planteada en el artículo de Castegren y Reyes [7]. Las secciones más relevantes, para la realización de nuestro trabajo, comprenden la definición del *typechecker* (`2`), el soporte para *backtraces* (`4`), y el agregado de *warnings* (`5`).

## 4.1. Metavariables

Como mencionamos en capítulos previos, para el estudio del lenguaje nos apoyaremos en la especificación de su *sintaxis abstracta*. A lo largo de la sección, se utilizarán diversas metavariables (a veces acompañadas de superíndices, o subíndices) para representar distintas clases de construcciones sintácticas. A continuación se listan las mismas, junto con el elemento sintáctico que comúnmente simbolizarán, a menos que se especifique lo contrario en el momento.

| | | | |
|---|---|---|---|
| Expresiones | | | |
| $e$ | $\langle expression \rangle$ | $ct$ | $\langle constant \rangle$ |
| $v$ | $\langle variable \rangle$ | $n$ | $\langle integer \rangle$ |
| $x, a$ | $\langle id \rangle$ | $r$ | $\langle real \rangle$ |
| $\oplus$ | $\langle binary \rangle$ | $b$ | $\langle bool \rangle$ |
| $\ominus$ | $\langle unary \rangle$ | $c$ | $\langle character \rangle$ |
| Sentencias | | | |
| $s$ | $\langle sentence \rangle$ | $ss$ | $\langle sentences \rangle$ |
| Tipos | | | |
| $\theta$ | $\langle type \rangle$ | $as$ | $\langle size \rangle$ |
| $td$ | $\langle typedecl \rangle$ | $tn$ | $\langle tname \rangle$ |
| $tv$ | $\langle typevariable \rangle$ | $cn$ | $\langle cname \rangle$ |
| $cl$ | $\langle class \rangle$ | $fn$ | $\langle fname \rangle$ |
| $io$ | $\langle io \rangle$ | $fd$ | $\langle field \rangle$ |
| Programas | | | |
| $fpd$ | $\langle funprocdecl \rangle$ | $cs$ | $\langle constraints \rangle$ |
| $vd$ | $\langle variabledecl \rangle$ | | |

## 4.2. Notación

Antes de comenzar propiamente con la definición de los distintos chequeos, es necesario describir el significado de la notación que emplearemos a lo largo del capítulo. Cuando uno de los elementos verificados satisfaga todas las propiedades requeridas por el análisis, diremos que el mismo se encuentra *bien formado*. Comúnmente, utilizaremos la siguiente notación para decir que la construcción sintáctica $\chi$ está *bien formada* bajo el contexto $\pi$, en base a las reglas de la categoría $\gamma$.

$$\pi \vdash_\gamma \chi$$

También denominadas reglas para *juicios de tipado*, estas construcciones a veces producirán resultados luego de finalizado su análisis. Los mismos podrán extender los contextos involucrados en la verificación, a medida que se recolecta información del programa, o también podrán generar nuevos elementos sintácticos, como es el caso para los chequeos de tipos. En estas situaciones, se utilizará la siguiente notación, donde $\omega$ representa el resultado final obtenido.

$$\pi \vdash_\gamma \chi : \omega$$

Otra situación que se suele presentar a la hora del análisis, es el uso de una cantidad arbitraria de contextos. Debido que comúnmente se deberá almacenar información de distintos índoles para

efectuar la verificación, se hará uso de la siguiente notación para reflejar esta condición. Notar que también existe la posibilidad que no se necesite utilizar ninguna información contextual, por lo que en estos escenarios se omite el listado de contextos.

$$\pi_1, \ldots, \pi_n \vdash_\gamma \chi : \omega$$

A lo largo del informe, se darán distintos conjuntos de reglas $\gamma$ en base al elemento sintáctico que $\chi$ represente. Al mismo tiempo, se definirán diversos contextos $\pi$ que almacenarán la información recopilada a lo largo de la validación del programa. Sumado a todo esto, la clase de resultados $\omega$ obtenidos durante la verificación también dependerá del entorno de análisis en el que nos encontremos inmersos.

Una última notación que puede ser conveniente presentar, es la utilizada para expandir contextos. Los contextos que emplearemos a lo largo del trabajo serán conjuntos compuestos por *n-uplas* de elementos sintácticos del lenguaje. Debido que la operación principal sobre estas construcciones será el agregado de las distintas componentes que conforman el elemento sintáctico recientemente analizado, se hará uso de la siguiente abreviatura para simplificar la notación.

$$(\chi_1, \ldots, \chi_n) \triangleright \pi \stackrel{def}{=} \{(\chi_1, \ldots, \chi_n)\} \cup \pi$$

## 4.3. Chequeos

Comenzando con la especificación de los chequeos, avanzaremos progresivamente en el análisis de un programa a medida que las distintas propiedades sean enunciadas y verificadas. Para asegurar la corrección estática de un programa, se deben validar cada una de sus componentes, y en el caso que todas superen su respectiva verificación, se dirá que el mismo se encuentra *bien formado*. Según la sintaxis del lenguaje, un programa posee la siguiente estructura, donde vale que $n \geq 0$ y $m > 0$.

$$\begin{gathered} typedecl_1 \\ \ldots \\ typedecl_n \\ funprocdecl_1 \\ \ldots \\ funprocdecl_m \end{gathered}$$

### 4.3.1. Chequeos para Declaración de Tipos

Las primeras reglas que precisaremos serán sobre las declaraciones de tipo de un programa. Con las mismas, buscamos verificar una serie de propiedades tales como la unicidad de los identificadores empleados para representar determinadas construcciones, la validez de los tipos especificados dentro de las declaraciones, e incluso el uso adecuado de las variables de tipo introducidas como parámetros en las definiciones.

Una definición de tipo *typedecl* consiste en alguna de las siguientes tres construcciones sintácticas; un tipo enumerado, un sinónimo de tipo, o una estructura de tipo tupla. Cuando una de estas declaraciones se encuentre *bien formada* su información será almacenada en el contexto adecuado. Habrá un contexto diferente para cada una de las categorías de tipos que se pueden definir en el lenguaje. Además, estos conjuntos tendrán una serie de invariantes que deberán ser respetadas a lo largo del análisis de un programa.

- **enum** $tn = cn_1, cn_2, \ldots, cn_m$
- **syn** $tn$ **of** $tv_1, \ldots, tv_l = \theta$
- **tuple** $tn$ **of** $tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m$

En el caso de la declaración de un tipo enumerado, se debe almacenar el nombre del tipo definido junto con el listado de constantes enumeradas en su cuerpo. Una invariante que se debe respetar en este contexto, es que los nombres de constantes deben ser únicos. Esto significa que no pueden ser repetidos dentro de una misma definición, ni tampoco ocurrir en otras.

$$\pi_e = \{(tn, \{cn_1, \ldots, cn_m\}) \mid tn \in \langle tname \rangle \wedge cn_i \in \langle cname \rangle\}$$

Para los sinónimos, además del nombre, se deben guardar las variables de tipo utilizadas como argumentos, junto con el tipo que lo define. En este contexto, la invariante debe asegurar que dentro de una declaración no se repitan los identificadores empleados para representar a los parámetros de la misma.

$$\pi_s = \{(tn, \{tv_1, \ldots, tv_l\}, \theta) \mid tn \in \langle tname \rangle \wedge tv_i \in \langle typevar \rangle \wedge \theta \in \langle type \rangle\}$$

Finalmente, para las tuplas, tenemos que almacenar su nombre, sus argumentos de tipo, y los distintos campos especificados en su definición. En esta situación, además de evitar la repetición de variables de tipo, se tiene que asegurar que los identificadores de campo sean únicos dentro del cuerpo de la declaración.

$$\pi_t = \{(tn, \{tv_1, \ldots, tv_l\}, \{fd_1, \ldots, fd_m\}) \mid tn \in \langle tname \rangle \wedge tv_i \in \langle typevar \rangle \wedge fd_j \in \langle field \rangle\}$$

Estos tres contextos se encargarán de almacenar toda la información relacionada con los tipos declarados por el usuario en un programa. A todas las condiciones de consistencia mencionadas anteriormente se le tiene que sumar una última. Los nombres de tipos definidos deben ser únicos. Es decir, que no puede haber más de una definición para el mismo identificador de tipo entre los distintos contextos.

### Tipos en Declaración de Tipos

A continuación, especificamos cuando un tipo empleado dentro de una declaración de un tipo nuevo, es válido. Claramente esto nos permitirá determinar cuando una definición de tipo se encuentra *bien formada*. El siguiente conjunto de reglas será utilizado en diversas secciones del análisis de un programa; en cada una de estas situaciones se evidenciarán ligeras modificaciones realizadas al mismo con el fin de adecuarlo al chequeo vigente.

Cuando nos encontramos en la derivación de una declaración de tipo, a la hora de analizar propiamente un tipo, utilizamos la siguiente notación para denotar que el tipo representado por $\theta$ es válido en el contexto de los tipos definidos; enumerados $\pi_e$, sinónimos $\pi_s$, y tuplas $\pi_t$. Utilizando una notación más compacta, comúnmente haremos referencia al contexto $\pi_{\mathbf{T}}$ para representar a la anterior tripla de contextos. Esta salvedad la tendremos para facilitar la lectura de las reglas, y poder concentrarnos propiamente en las derivaciones.

$$\pi_e, \pi_s, \pi_t \vdash_t \theta$$
$$\pi_{\mathbf{T}} \vdash_t \theta$$

Para decidir si uno de estos *juicios* es válido, tenemos que proveer una derivación utilizando las reglas que definiremos a continuación. Con la aplicación sucesiva de las mismas, se pueden

construir pruebas que demuestran las distintas propiedades requeridas para que un tipo en un programa sea considerado estáticamente correcto.

Comenzaremos con los tipos básicos del lenguaje. La prueba de los mismos es inmediata, ya que su regla no presenta ninguna premisa. Por lo tanto, todo tipo básico es considerado un tipo correcto.

**Regla DT para Tipos:** Básicos

$$\frac{}{\pi_{\mathbf{T}} \vdash_t \theta} \quad \text{cuando } \theta \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{char}\}$$

Un puntero será correcto, siempre que el tipo del valor al que hace referencia sea correcto. Notar que la premisa de la regla requiere de la prueba de un tipo estructuralmente menor al inicial.

**Regla DT para Tipos:** Punteros

$$\frac{\pi_{\mathbf{T}} \vdash_t \theta}{\pi_{\mathbf{T}} \vdash_t \mathbf{pointer}\ \theta}$$

Para los arreglos, solo se debe verificar el tipo de los valores que almacenará. Debido que en la declaración de tipos no se permite utilizar tamaños variables para las dimensiones de un arreglo, la regla se simplifica al no tener que realizar ninguna validación sobre los mismos. Si quisiéramos permitir el uso de tamaños polimórficos en esta instancia del programa, deberíamos adaptar la forma en que se declaran los nuevos tipos definidos por el usuario.

**Regla DT para Tipos:** Arreglos

$$\frac{\pi_{\mathbf{T}} \vdash_t \theta}{\pi_{\mathbf{T}} \vdash_t \mathbf{array}\ as_1, \ldots, as_n\ \mathbf{of}\ \theta} \quad \text{cuando } as_i \in \langle natural \rangle$$

En primera instancia, una variable de tipo es correcta de forma inmediata. Inicialmente, esto puede resultar inadecuado para las propiedades que queremos verificar en un programa. La razón de esta definición quedará clara más adelante, de momento adelantamos que la declaración de un tipo tiene restricciones bien duras con respecto a la introducción de variables de tipo y su utilización; de esta manera el chequeo de la corrección de un tipo en el contexto del chequeo de la declaración de uno tiende a simplificarse para el caso de las variables.

**Regla DT para Tipos:** Variables de Tipo

$$\frac{}{\pi_{\mathbf{T}} \vdash_t tv}$$

Las reglas para los tipos definidos, pueden separarse en dos categorías en base si los mismos poseen, o no, argumentos de tipo. La prueba de un *juicio de tipado* para un tipo definido no parametrizado, consiste simplemente de constatar que su nombre se encuentra declarado en algún contexto determinado. Evidentemente, hay que asegurar que en su definición no se haya especificado ningún argumento de tipo.

**Regla DT para Tipos:** Tipos Enumerados

$$\frac{(tn, \{cn_1, \ldots, cn_m\}) \in \pi_e}{\pi_e, \pi_s, \pi_t \vdash_t tn}$$

**Regla DT para Tipos:** Sinónimos sin Argumentos

$$\frac{(tn, \emptyset, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t \vdash_t tn}$$

**Regla DT para Tipos:** Tuplas sin Argumentos

$$\frac{(tn, \emptyset, \{fd_1, \ldots, fd_m\}) \in \pi_t}{\pi_e, \pi_s, \pi_t \vdash_t tn}$$

En cambio, para un tipo parametrizado, es necesario realizar unas verificaciones adicionales. En particular, se deben validar todos los tipos especificados como argumentos del mismo, y que la cantidad de estos coincida con los declarados en su definición.

**Regla DT para Tipos:** Sinónimos con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_l\}, \theta) \in \pi_s \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_l}$$

**Regla DT para Tipos:** Tuplas con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_l\}, \{fd_1, \ldots, fd_m\}) \in \pi_t \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_l}$$

A continuación, presentamos como es una derivación para probar la corrección de un tipo determinado del lenguaje. Supongamos los siguientes contextos donde no hay declarados tipos enumerados, ni sinónimos de tipo, y solo tenemos definido el tipo tupla *node* parametrizado en *Z*. Notar que el campo *elem* es de tipo variable *Z*, y *next* es un puntero al mismo tipo.

$$\pi_e = \emptyset$$
$$\pi_s = \emptyset$$
$$\pi_t = \{(node, \{Z\}, \{elem : Z, next : \textbf{pointer } node \textbf{ of } Z\})\}$$

Si nos adelantamos un poco, y quisiéramos probar la declaración de tipo **syn** *list* **of** $A =$ **pointer** *node* **of** $A$, entonces una parte de la prueba consistirá en demostrar la validez del tipo que ocurre dentro de la definición. Por lo tanto, con los contextos previos, se puede hacer lo siguiente. Notar el uso de las reglas para punteros, tuplas definidas, y variables de tipo.

**Prueba 1.** Demostración de corrección para el tipo *puntero a nodo*.

$$\frac{\dfrac{(node, \{Z\}, \{elem : Z, next : \textbf{pointer } node \textbf{ of } Z\}) \in \pi_t \qquad \dfrac{}{\pi_e, \pi_s, \pi_t \vdash_t A}}{\pi_e, \pi_s, \pi_t \vdash_t node \textbf{ of } A}}{\pi_e, \pi_s, \pi_t \vdash_t \textbf{pointer } node \textbf{ of } A}$$

### Variables de Tipo en Declaración de Tipos

Además de las reglas para el chequeo de tipos que hemos especificado, necesitamos definir cuando una variable de tipo es considerada *libre*. En el lenguaje no hay ninguna clase de cuantificación para estos elementos, pero nos referiremos de esta manera informal a todas las variables que ocurran dentro de una declaración de tipo, y que no estén en el alcance de un parámetro con el mismo identificador.

$$FTV : \langle type \rangle \to \{\ \langle typevar \rangle\ \}$$

El propósito de esta definición, es poder chequear que todos los argumentos de una declaración de tipo sean efectivamente utilizados en su cuerpo, y al mismo tiempo, que todas las ocurrencias de variables estén asociadas a un parámetro determinado. La función para calcular el conjunto de *variables de tipo* presentes en un tipo se describe a continuación.

$$\begin{aligned}
&FTV(\textbf{int}) && = \emptyset \\
&FTV(\textbf{real}) && = \emptyset \\
&FTV(\textbf{bool}) && = \emptyset \\
&FTV(\textbf{char}) && = \emptyset \\
&FTV(\textbf{pointer}\ \theta) && = FTV(\theta) \\
&FTV(\textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ \theta) && = FTV(\theta) \\
&FTV(tv) && = \{tv\} \\
&FTV(tn) && = \emptyset \\
&FTV(tn\ \textbf{of}\ \theta_1, \ldots, \theta_n) && = FTV(\theta_1) \cup \ldots \cup FTV(\theta_n)
\end{aligned}$$

### Declaración de Tipos

Definidas las reglas para chequear cuando un tipo es válido, y la función que calcula las variables de tipo que ocurren en el mismo, comenzaremos con la especificación de las reglas empleadas en la prueba de corrección para declaraciones de tipo. Cuando se determina que una definición esta *bien formada*, su información es añadida al contexto apropiado y se continua con el análisis del programa. Notar que la prueba de una serie de declaraciones de tipo responde al orden en que las mismas se encuentran especificadas, y aún más importante, que las reglas no permiten la definición mutua entre estas declaraciones. De esta forma, un tipo definido solo será accesible para las declaraciones posteriores al mismo.

El *juicio de tipado* que prueba la validez de una declaración de tipo es el siguiente. Luego del análisis, se producirá un nuevo contexto donde se agrega la información de la definición recientemente verificada al contexto inicial. Recordar que al realizarse estas extensiones, se deben seguir respetando las invariantes de consistencia para los conjuntos involucrados.

$$\pi_{\mathbf{T}} \vdash_{td} typedecl : \pi'_{\mathbf{T}}$$

La regla para la definición de tipos enumerados es simple. Debido a las invariantes de los contextos de tipos, la deducción es inmediata. Con la construcción del nuevo conjunto, uno puede asegurar la unicidad del nombre de tipo respecto a las otras definiciones, y que los constructores empleados en la misma no se repiten a lo largo del programa.

**Regla DT:** Enumerados

$$\frac{}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{enum}\ tn = cn_1, \ldots, cn_m : \pi'_e, \pi_s, \pi_t}$$

donde $\pi'_e = (tn, \{cn_1, \ldots, cn_m\}) \triangleright \pi_e$.

Para los sinónimos, hay que realizar un par de verificaciones. Primero, se tiene que comprobar que el conjunto de parámetros de la declaración coincida con el conjunto de variables de tipo utilizadas en la definición del mismo. Esta condición evita la ocurrencia de variables *libres* en el cuerpo de la declaración, y también obliga el uso de todos los argumentos de la misma. Segundo, el tipo que propiamente define al sinónimo tiene que ser válido. La invariante del contexto garantiza la unicidad de los identificadores empleados para los argumentos de la declaración.

**Regla DT:** Sinónimos sin Argumentos

$$\frac{FTV(\theta) = \emptyset \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{syn}\ tn = \theta : \pi_e, \pi'_s, \pi_t}$$

donde $\pi'_s = (tn, \emptyset, \theta) \triangleright \pi_s$.

**Regla DT:** Sinónimos con Argumentos

$$\frac{FTV(\theta) = \{tv_1, \ldots, tv_l\} \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{syn}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = \theta : \pi_e, \pi'_s, \pi_t}$$

donde $\pi'_s = (tn, \{tv_1, \ldots, tv_l\}, \theta) \triangleright \pi_s$.

Las verificaciones para tuplas son similares a las de sinónimos, salvo que se deben adecuar para los múltiples campos de la misma. Hay que asegurar la igualdad entre los argumentos de la definición, y las variables de tipo que ocurren en todos los campos. Además, se tienen que analizar todos los tipos para asegurar su corrección. Por último, se deben respetar las invariantes de unicidad tanto para los nombres de campos, como para los argumentos de tipo.

**Regla DT:** Tuplas sin Argumentos

$$\frac{FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) = \emptyset \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \emptyset, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$.

**Regla DT:** Tuplas con Argumentos

$$\frac{FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) = \{tv_1, \ldots, tv_l\} \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \{tv_1, \ldots, tv_l\}, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$.

La última regla es la que permite la definición de tipos recursivos. La única posibilidad de declarar un tipo que se define en términos de si mismo es mediante el uso de punteros dentro de tuplas. Por lo tanto, tiene sentido que esta regla sea una variante de las reglas previas. Notar que la definición de tipos recursivos es bastante restrictiva. Solo se permiten utilizar las mismas variables de tipo paramétricas que en la definición, e incluso se las debe especificar en el mismo orden. En particular, si un tipo representa una llamada recursiva al tipo declarado, entonces el mismo quedará exceptuado del chequeo de validez para tipos. Las invariantes de consistencia para los contextos se deben respetar al igual que en las reglas anteriores.

**Regla DT:** Recursión para Tuplas sin Argumentos

$$\frac{FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) = \emptyset \qquad \theta_i \neq \textbf{pointer}\ tn \implies \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \emptyset, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$.

**Regla DT:** Recursión para Tuplas con Argumentos

$$\frac{FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) = \{tv_1, \ldots, tv_l\} \qquad \theta_i \neq \textbf{pointer}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l \implies \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \{tv_1, \ldots, tv_l\}, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$.

Siguiendo con el ejemplo especificado previamente, a continuación presentamos la prueba de corrección de una declaración de tipo del lenguaje. Primero verificaremos una estructura de tipo tupla, y luego haremos lo mismo para un sinónimo de tipo. Supongamos los siguientes contextos, los cuales serán los resultados obtenidos luego de realizados los respectivos análisis.

$$\pi_s = \{(list, \{A\}, \textbf{pointer}\ node\ \textbf{of}\ A)\}$$
$$\pi_t = \{(node, \{Z\}, \{elem : Z, next : \textbf{pointer}\ node\ \textbf{of}\ Z\})\}$$

Comenzando con los contextos de tipos vacíos, se puede realizar la prueba para un tipo recursivo de la siguiente forma. Como se preciso anteriormente, debido que el campo *next* realiza una llamada recursiva al tipo que estamos definiendo, no se debe verificar su tipo. Notar el uso de las reglas para las variables de tipo, y la recursión en declaración de tuplas.

**Prueba 2.** Demostración de corrección para la declaración de tipo *nodo*.

$$\frac{FTV(Z) \cup FTV(\textbf{pointer}\ node\ \textbf{of}\ Z) = \{Z\} \qquad \overline{\emptyset_e, \emptyset_s, \emptyset_t \vdash_t Z}}{\emptyset_e, \emptyset_s, \emptyset_t \vdash_{td} \textbf{tuple}\ node\ \textbf{of}\ Z = elem : Z, next : \textbf{pointer}\ node\ \textbf{of}\ Z : \emptyset_e, \emptyset_s, \pi_t}$$

Utilizando el contexto obtenido en la derivación anterior, se puede realizar la siguiente prueba. En la misma, hacemos uso de una verificación previa donde se demostraba la corrección de un tipo puntero. Notar que empleando las distintas herramientas introducidas a lo largo del capítulo, podemos probar la validez de la *lista abstracta* en nuestro lenguaje.

**Prueba 3.** Demostración de corrección para la declaración de tipo *lista*.

$$\frac{FTV(\textbf{pointer}\ node\ \textbf{of}\ A) = \{A\} \qquad \dfrac{\text{Prueba 1}}{\emptyset_e, \emptyset_s, \pi_t \vdash_t \textbf{pointer}\ node\ \textbf{of}\ A}}{\emptyset_e, \emptyset_s, \pi_t \vdash_{td} \textbf{syn}\ list\ \textbf{of}\ A = \textbf{pointer}\ node\ \textbf{of}\ A : \emptyset_e, \pi_s, \pi_t}$$

# Capítulo 5

# Conclusión

En este trabajo de tesis hemos presentado un nuevo lenguaje, ΔΔLang. Se han descripto las motivaciones principales que justificaron su creación, y se hizo mención de los elementos más relevantes que lo definen. Junto a esto, con una serie de ejemplos sobre casos de uso y explicaciones informales, se han expuesto las diversas construcciones que ofrece el lenguaje. Desde un punto de vista teórico, las contribuciones de la tesis se pueden resumir en:

- La especificación de un lenguaje de programación que podrá ser utilizado para la formación de estudiantes en la carrera.
- La definición de su sintaxis, abstracta y concreta, lo cual permite su aplicación durante el desarrollo de la asignatura.
- La formulación de los chequeos estáticos, que determinan cuando un programa está bien formado antes de su ejecución.

Sumado a lo anterior, hemos realizado la primer iteración en el desarrollo para el respectivo intérprete del lenguaje. Utilizando la formalización previa como base, se ha implementado la etapa de *análisis* de nuestro programa. Desde un punto de vista práctico, algunas de las contribuciones de este trabajo son:

- La implementación de las fases de *análisis léxico* y *análisis sintáctico*, las cuales fueron desarrolladas junto al parser del intérprete.
- La implementación de la fase de *análisis semántico*, que comprende las verificaciones estáticas que realiza el intérprete previa la ejecución de un programa.

## 5.1. Trabajos Futuros

El diseño del intérprete aún está lejos de estar terminado. Este trabajo solo comprendió la primer etapa de desarrollo del mismo, y sentó las bases para su implementación. Por lo tanto, a continuación describiremos distintas tareas para realizar a futuro, con el fin de cumplir nuestro objetivo. Algunas estarán destinadas a completar la especificación restante del intérprete, siguiendo con las fases posteriores necesarias para su definición. Mientras, otras podrán ser realizadas con la finalidad de mejorar y expandir las funcionalidades actuales de nuestro programa.

### 5.1.1. Continuando el Desarrollo

Hasta ahora, solo hemos trabajado en la etapa de *análisis* del intérprete. Partiendo de un archivo de texto, somos capaces de obtener la estructura sintáctica de un programa, y verificar de forma *estática* las distintas propiedades semánticas requeridas por el lenguaje. Pero ahora debemos pasar a una nueva etapa, la fase de *síntesis*. Utilizando la representación intermedia obtenida previamente, queremos tener la capacidad de realizar una ejecución del código provisto, y durante la misma, también poder validar de forma *dinámica* ciertas condiciones necesarias para asegurar el correcto funcionamiento del programa. Sumado a todo esto, es indispensable diseñar un medio de interacción para que el programador pueda comunicarse con nuestro intérprete. El desarrollo de una *interfaz de usuario* es otro aspecto fundamental para la implementación del programa, y su posterior aplicación en el dictado de la asignatura.

#### Síntesis

La etapa de *síntesis* comprende todos los aspectos *dinámicos* sobre la interpretación de un programa. Comenzando con el *árbol de sintaxis abstracta* obtenido durante el *análisis* del código, debemos simular la ejecución del programa en base a los datos de entrada provistos por el usuario. Para poder realizar esta acción, necesitaremos definir formalmente la semántica *small step* del lenguaje; la cual facilitará los medios necesarios para examinar detalles particulares sobre el orden de evaluación de las distintas construcciones especificadas en el código. Esto nos permitirá efectuar una por una las instrucciones del programa, al mismo tiempo que se exhiben los estados intermedios obtenidos durante su computación.

Sumado a lo anterior, durante la ejecución del código, es necesario llevar a cabo la validación *dinámica* del mismo. Debido que la totalidad de errores de un programa no puede ser detectada de forma *estática*, hay ciertas verificaciones que son realizadas durante esta etapa. Las mismas consistirán en su mayoría de asegurar el uso adecuado de memoria por parte del usuario. Ya sea para liberar o reservar memoria mediante el empleo de punteros, o incluso el acceso a ciertas ubicaciones de memoria representadas por variables, es fundamental asegurar que la ejecución del programa se pueda realizar de forma consistente, y que no se presenten comportamientos inesperados durante la misma.

#### Interfaz de Usuario

Uno de los motivos por el que se inició el desarrollo del intérprete, fue para facilitar el estudio de los contenidos presentados en la materia *Algoritmos y Estructura de Datos II*. Debido que los usuarios finales de nuestro programa serán, en su mayoría, estudiantes que se están introduciendo en el ámbito de la implementación de algoritmos, es importante proveer una *interfaz de usuario* intuitiva y amigable para el uso del intérprete. Actualmente, no hay ningún aspecto completamente definido sobre el futuro diseño de la interfaz, sino más bien, se están valorando distintas alternativas y funcionalidades particulares para su próxima implementación.

En la versión presente del intérprete, solo se cuenta con una interfaz básica por línea de comando, la cual se asemeja más a una herramienta para la verificación del código implementado del programa que a un mecanismo para interactuar propiamente con el intérprete. Una de las opciones consideradas, es la creación de una interfaz gráfica de usuario la cual permita observar la evolución del estado de las variables del programa a medida que se avanza en su ejecución. También se analiza la posibilidad de utilizar una codificación de colores para resaltar los errores encontrados durante la interpretación del código. Obviamente, se proveería un editor de texto dentro del programa para facilitar esta característica. Incluso se discutió adicionar soporte web a la herramienta, lo que permitiría utilizar el intérprete desde el navegador.

### 5.1.2. Generación de Múltiples Errores

Uno de los aspectos que se puede mejorar del intérprete, es la generación de múltiples errores tanto en la etapa de *análisis sintáctico* como en la de *análisis semántico*. Actualmente, cuando se encuentra un error durante el parsing o la verificación de un programa se aborta por completo el análisis del mismo, generando un mensaje de error informativo sobre la causa de la falla. Una característica deseable en el intérprete, es poder capturar la mayor cantidad posible de errores encontrados en una determinada etapa, antes de fallar, y generar todos los mensajes necesarios correspondientes. Esto obviamente beneficiaría al usuario, ya que se tendría que dedicar menor tiempo a la corrección de errores, agilizando efectivamente el uso de la herramienta, y podría destinar mayor atención al diseño de algoritmos.

#### Análisis Sintáctico

La librería *Megaparsec* ofrece un mecanismo para señalar múltiples errores en una sola corrida del parser. Un requisito para poder utilizar esta funcionalidad, es que debe ser posible omitir una sección problemática de la entrada (donde comúnmente se generaría un error de parsing) y resumir el análisis en una posición que se considere estable. Esto se consigue con el uso de la primitiva `withRecovery`.

Si quisiéramos aprovechar esta funcionalidad, deberíamos adaptar nuestro parser. Debido que identificar cual puede ser un buen punto de recuperación es una tarea compleja, es necesario realizar algunas modificaciones al intérprete para poder facilitar la misma. Un cambio conveniente posible, sería utilizar el punto y coma (**;**) para separar la secuencia de instrucciones del lenguaje. De esta forma, al fallar el parser se podrían consumir *tokens* hasta encontrar este delimitador, punto donde se puede recuperar y continuar el análisis normal del programa.

#### Análisis Semántico

Una posibilidad para la generación de múltiples errores en esta etapa, está descripta en el artículo que se empleó como guía para el diseño de los chequeos estáticos del intérprete [7]. Esta opción consiste en la implementación de un combinador, que es invocado cada vez que se deben verificar una serie de elementos sintácticos. La idea es que se puedan combinar la lista de chequeos a realizar y, si todos tienen éxito, se devuelven sus resultados correspondientes. En caso contrario, se deberán acumular la totalidad de los errores producidos, y luego generar los mensajes informativos adecuados. Esta técnica *ad hoc*, resulta ser la más simple, y requiere pocas modificaciones del código actual. De todas formas, no aprovecha al máximo la estructura en la que se organiza un programa, y puede no ser trivial cuando se debe aplicar el combinador, y cuando no. Esto se debe que muchas veces la corrección semántica de cierta construcción, depende de la validez de los elementos previos a los que la misma hace referencia.

Otra opción, es rediseñar los chequeos estáticos actuales para realizar más de una recorrida al *árbol de sintaxis abstracta*. La idea es que en una primera pasada, se puedan analizar todos los prototipos de las definiciones de tipo, y las declaraciones de funciones y procedimientos del programa. Una vez finalizado este análisis, se deberá realizar una segunda pasada donde esta vez se deberán verificar los cuerpos de las construcciones anteriormente mencionadas. De esta manera, se pueden acumular la totalidad de errores encontrados en una de estas fases, antes de abortar el análisis del programa con un mensaje de error. Comparada con la técnica anterior, la opción actual resulta mucha más compleja y necesita modificar gran parte de la implementación. A pesar de esto, la misma aprovecha la idea que los prototipos son válidos de forma mutuamente independiente entre ellos, al igual que como ocurre con sus respectivos cuerpos. Otra ventaja de la técnica actual sobre la opción anterior, es que la misma permitiría invocar funciones y

procedimientos sin importar el lugar espacial donde hayan sido declaradas. Incluso se admitiría la posibilidad de definir de forma mutua las construcciones previamente mencionadas.

### 5.1.3. Funcionalidades Adicionales

Existen una serie de funcionalidades que fueron consideradas a lo largo del desarrollo de este trabajo, pero que no llegaron a ser incluidas en esta primera versión del intérprete. Las mismas fueron relegadas por diversos motivos. Algunas debieron ser omitidas por falta de tiempo, otras a causa de no poder llegar a un concenso en su diseño, e incluso algunas por las dificultades encontradas durante su implementación. A continuación daremos una breve descripción de cada una de estas, junto con sugerencias para facilitar su futura incorporación al intérprete.

#### Soporte para Múltiples Módulos

Nuestro intérprete solo permite definir programas en un único archivo. Esto puede no ser un limitante en el poder expresivo del lenguaje, pero si resulta un inconveniente para su incorporación en el dictado de la materia. Debido que durante la asignatura se hace hincapié en la importancia de la separación de los módulos para la especificación de *tipos abstracto de datos*, y los correspondientes a su implementación, es fundamental que el lenguaje provea las herramientas necesarias para mantener esa abstracción mediante el encapsulamiento de las construcciones involucradas.

Para incorporar esta funcionalidad, se deberá determinar la sintaxis adecuada para poder importar y exportar tipos, funciones, y procedimientos definidos dentro de un determinado módulo. Sumado a esto, se tendrán que adaptar los chequeos semánticos para soportar estas nuevas situaciones. En particular, los sinónimos de tipos actualmente son interpretados como una definición transparente, similar a como lo hace *Haskell* cuando se declaran con `type`. Idealmente, quisiéramos que su definición se vuelva opaca fuera del módulo en el que fueron declarados, o incluso, reproducir un comportamiento parecido a `newtype` en *Haskell*; lo cual también permitiría declarar nuevas instancias de clase para el tipo aludido. Adicionalmente, se deberá adaptar la información de posición `Info` para almacenar además, el nombre del archivo `File` correspondiente a la ocurrencia del elemento sintáctico analizado.

#### Coersiones y Subtipado

En el lenguaje, hay dos tipos numéricos; los valores enteros y los valores reales. En ambos casos, se encuentra definido un listado de operadores sobrecargados que pueden ser utilizados para trabajar con cualquiera de los valores previos. Una restricción de la implementación actual, es que ambos tipos son percibidos como valores completamente diferentes. Esto significa, por ejemplo, que la operación de sumar enteros con reales es detectada como un error de tipos. Al mismo tiempo, la constante **inf** es interpretada como un valor entero, al igual como sucede con los tamaños polimórficos. Esto imposibilita especificar ciertos algoritmos donde deseamos realizar operaciones donde se combinan ambos tipos de valores numéricos, como es el caso de querer calcular el promedio de un arreglo de números reales, el cual posee tamaño variable. El código presentado en (5.1) ilustra esta situación mencionada. Notar que al analizar la última instrucción de la función, el intérprete fallará debido a un error de tipos producido por la división de un número real por uno de tipo entero.

Para flexibilizar el intérprete, podríamos implementar una *coersión* implícita que convierta un número de tipo entero, a otro equivalente de tipo real, si fuese necesario. De esta forma, obtendríamos la capacidad de emplear valores enteros en contextos donde se esperan números reales, como es el caso del ejemplo previo, donde deseamos dividir la sumatoria de valores en un

arreglo de reales por su tamaño entero. Otras situaciones que también se permitirían, comprenden la asignación de enteros a variables reales, o la llamada de funciones con parámetros del primer tipo cuando se esperaban del segundo. Incluso podríamos ir un paso más adelante, y declarar a los números enteros como un *subtipo* de los valores reales. Esto implicaría una modificación estructural del actual sistema de tipos, donde deberíamos permitir situaciones más complejas que las contempladas anteriormente, como puede ser la de manipular un arreglo de enteros como si fuese un arreglo de reales. La propiedad aún debe ser discutida y evaluada, ya que es necesario determinar su verdadera utilidad para el dictado de la materia.

```
1 fun promedioReal ( a : array [n] of real ) ret promedio : real
2   var sumatoria : real
3   sumatoria := 0
4   for i := 1 to n do
5     sumatoria := a[i] + sumatoria
6   od
7   promedio := sumatoria / n { Error de Tipos }
8 end fun
```

Listing 5.1: Promedio en Arreglo de Reales

### Estructuras Iterables

Durante el desarrollo del intérprete, se analizó la posibilidad de incluir la clase **Iter** para caracterizar a todas las construcciones que posean la capacidad de ser iteradas. Ejemplos de la misma, podrían ser los arreglos y las listas. El primero formaría parte de esta clase de manera predefinida, mientras que el segundo necesita de su respectiva implementación. Estas estructuras representan una serie de elementos con algún orden determinado. Una vez que el usuario define la instancia correspondiente a esta clase, para un tipo en particular, se podrán utilizar valores del mismo dentro de la instrucción **for** $x$ **in** $e$ **do** $ss$, lo cual permitiría recorrer estas construcciones, e ir obteniendo uno por uno todos los elementos que las conforman. Un ejemplo posible se ilustra en (5.2), donde se calcula la sumatoria de elementos de un arreglo.

```
1 fun sumatoriaIter ( a : array [n] of int ) ret sumatoria : int
2   sumatoria := 0
3   for i in a do
4     sumatoria := i + sumatoria
5   od
6 end fun
```

Listing 5.2: Sumatoria de Arreglo Iterable

Se estudiaron los mecanismos que lenguajes como *Python* y *Java* emplean para crear objetos iterables. En base a esto, se discutieron distintas maneras para declarar instancias de la clase en nuestro lenguaje. Una alternativa considerada, consistía en definir tres métodos para la estructura iterable. El primero, inicializaba un cursor para referenciar al primer elemento de la construcción. El segundo, verificaba si existía un sucesor en base al puntero actual. Y finalmente, el tercero obtenía el siguiente elemento, desplazando el cursor hacia adelante. Debido que la declaración de estructuras iterables no se adecuaba a la manera utilizada para definir instancias en el lenguaje, se decidió aplazar el diseño de esta funcionalidad para la siguiente etapa de desarrollo del intérprete.

### Clases e Instancias

Actualmente, en ΔΔLang solo hay definidas dos clases, **Eq** y **Ord**. Las mismas representan a los tipos que permiten comparaciones entre sus valores, ya sea en base a su igualdad en el caso de la primera, o según su orden para la segunda. Hay una posibilidad que en un futuro resulte útil añadir nuevas clases al lenguaje. Para lograr esta tarea, solo habría que agregar los operadores, funciones y/o procedimientos correspondientes a la clase que se quiere incorporar, junto con la adición de los chequeos adecuados para los mismos. Sobre esta característica, una opción posible sería añadir la clase **Enum**. La cual se emplearía para representar a todos los tipos que pueden ser enumerados desde su primer valor hasta el último. Incluso, otra alternativa sería adicionar la clase **Num**. Para representar a cualquier conjunto de valores que admita las operaciones de suma, resta, multiplicación, y demás; se utilizaría esta última clase.

```
type nodo = tuple
              valor : int,
              sucesor : pointer of nodo
            end tuple

  inst Eq ( n1, n2 : nodo ) ret igual : bool
    igual := n1.valor == n2.valor && n1.sucesor == n2.sucesor
  end inst

end type
```

Listing 5.3: Declaración de Instancia para Nodo

Sobre como declarar una instancia para una clase determinada, en el lenguaje solo se ha definido una sintaxis provisoria en este aspecto. En el fragmento (5.3) se ilustra un ejemplo particular, en el cual se especifica la operación de igualdad para el tipo *nodo*. Inicialmente al limitarnos a solo dos clases, que caracterizan propiedades similares, la verificación de sus instancias se simplifica. A continuación, describimos informalmente las validaciones que se deberían efectuar para asegurar que una definición de instancia es correcta.

1. Para todo tipo definido, existe una única instancia para una determinada clase. Similar a *Haskell*, se pueden emplear tipos concretos, o agregar restricciones de clase a las variables de tipo, en las declaraciones de instancias para tipos parametrizados.

2. Las instancias solo pueden tomar dos argumentos, y deben ser del mismo tipo. En particular, la definición del tipo y sus declaraciones de instancias deben ir juntas. Esta verificación es propia de las únicas dos clases del lenguaje.

3. El retorno de una instancia debe ser un valor booleano. Esta propiedad, al igual que la anterior, es específica de las clases mencionadas previamente.

4. El cuerpo de la declaración de instancias deberá cumplir las mismas verificaciones que satisface el cuerpo de una función. Esto se debería mantener incluso para las validaciones realizadas durante el tiempo de ejecución.

Un último detalle importante a mencionar, es que la declaración de instancias se permite solamente para las estructuras de tipo tupla. Esto es debido que un tipo enumerado satisface ambas clases de forma natural. Mientras que un sinónimo de tipo, hereda todas las clases que implementa el tipo de su definición. Si en un futuro se adoptará otro juicio, la modificación de la implementación actual es sencilla.

### Inferencia de Tipos

El sistema de tipos que se encuentra actualmente implementado en el intérprete, puede resultar básico a la hora de resolver ciertas cuestiones. Debido a esto, se presentan algunas limitaciones cuando se utiliza el mismo. Un ejemplo particular, es la llamada de funciones y procedimientos con la constante **null**, la cual simboliza un puntero vacío. Supongamos que hemos declarado una función como en el código de (5.4); la cual obtiene el elemento señalado por un puntero si el mismo no es nulo, o devuelve su segundo argumento en caso contrario. Si quisiéramos invocar a la misma con la llamada `accesoPuntero(null,5)`, entonces se generaría un error en la etapa de *análisis semántico* del intérprete, debido a la imposibilidad de chequear los tipos involucrados. Esto es debido que la constante **null** tiene tipo polimórfico, lo que significa que puede ser utilizada como un puntero a un entero, como también un puntero a un arreglo, o alguna otra estructura diferente. Esta característica permite un sistema de tipos más flexible, lo que facilita el uso del intérprete; pero al mismo tiempo, imposibilita el chequeo de tipos cuando el *algoritmo de unificación* necesita de información adicional para inferir un tipo particular en la llamada de una función o procedimiento.

```
1 fun accesoPuntero ( p : pointer of T, t : T) ret valor : T
2   if p == null then
3     valor := t
4   else
5     valor := #p
6   fi
7 end fun
```

Listing 5.4: Ejemplo para Inferencia de Tipos

Debido a esta constante, es necesario adaptar nuestro sistema de tipos para poder contemplar esta clase de situaciones. En otros contextos donde se puede emplear el valor **null**, el algoritmo de *type check* es capaz de resolver esta particularidad mediante el uso de técnicas *ad hoc*. El conflicto sucede cuando es necesario realizar algún tipo de inferencia para esta constante polimórfica, combinada con el polimorfismo paramétrico que admite una función o un procedimiento, lo que nos fuerza a requerir una mayor cantidad de información contextual para poder realizar las validaciones correspondientes. Para resolver este limitante, se presentan dos opciones que requieren la modificación de la implementación actual.

La primera, orientada al tipado explícito, implica la adaptación de la sintaxis del lenguaje para forzar al usuario a anotar el tipo concreto que tendrá la constante en el momento que se utiliza. De esta forma, durante la verificación se tendrá la suficiente cantidad de información para que el *juicio de tipado* sea directo. Como consecuencia de esta alternativa, el sistema de tipos se simplifica a costa de una sintaxis más verbosa. Volviendo al ejemplo, si se efectuara la llamada `accesoPuntero(null[int],5)`, el *algoritmo de unificación* sería capaz de verificar todas las propiedades necesarias de forma exitosa, y se continuaría con el análisis del programa.

La segunda opción, orientada al tipado implícito, supone la implementación de un algoritmo de *inferencia de tipos*. Extrayendo ideas del *algoritmo Hindley-Milner* [8], se podría adecuar nuestro sistema de tipos para lograr que el mismo sea capaz de inferir los *juicios de tipado* apropiados, sin necesidad de anotar el tipo de nuestra constante. El ejemplo previo permanecería sin cambios, sin embargo se necesitaría de una modificación estructural del algoritmo de *type check* implementado. De manera opuesta a la alternativa anterior, la sintaxis no se debería ajustar pero el sistema de tipos resultaría más complejo. Una propiedad que se ganaría al elegir esta opción, es que se permitirían declarar otras clases de funciones como la del fragmento (5.5). En la misma, se inicializa una lista de forma genérica, sin conocer el tipo de sus elementos.

```
fun inicializarLista ( ) ret lista : lista of (T)
  lista := null
end fun
```

Listing 5.5: Inicialización de Lista con Inferencia de Tipos

# Bibliografía


[1] Niklaus Wirth. *The Programming Language Pascal (Revised Report)*. Springer Verlag, 1972.

[2] Alfred Aho, Ravi Sethi y Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[3] Daan Leijen. *Parsec*. Ver. 3.1.14.0. 2019. URL: `https://github.com/haskell/parsec`.

[4] Mark Karpov. *Megaparsec*. Ver. 8.0.0. 2019. URL: `https://github.com/mrkkrp/megaparsec`.

[5] GHC Team. *The Glorious Glasgow Haskell Compiler*. Ver. 8.10.1. 2020. URL: `https://gitlab.haskell.org/ghc/ghc`.

[6] John Charles Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[7] Elias Castegren y Kiko Fernandez Reyes. «Developing a Monadic Type Checker for an Object-Oriented Language: An Experience Report». En: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering* (2019).

[8] Luis Damas y Robin Milner. «Principal Type-Schemes for Functional Programs». En: *9th Symposium on Principles of Programming Languages* (1982).