

$\Delta\Delta$ Lang

Implementación de un lenguaje *Pascal*-Like

Matías Federico Gobbi



Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba
18 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

1. Introducción	4
1.1. Motivación	4
1.1.1. Objetivos de la Asignatura	4
1.1.2. Programa de la Asignatura	5
1.2. Características del Lenguaje	6
1.2.1. Tipado	6
1.2.2. Polimorfismo	7
1.2.3. Recursión	7
1.2.4. Manejo de Memoria	8
1.2.5. Encapsulamiento	8
1.3. Desarrollo del Intérprete	8
1.3.1. Análisis	9
1.3.2. Síntesis	10
2. Sobre el lenguaje	11
3. Sobre el parser	12
3.1. Sintaxis Abstracta	12
3.1.1. Expresiones	12
3.1.2. Sentencias	13
3.1.3. Tipos	14
3.1.4. Programas	15
3.2. Sintaxis Concreta	16
3.2.1. Identificadores	16
3.2.2. Azúcar Sintáctico	17
3.3. Parser	17
3.3.1. Librerías	18
3.3.2. Información de Posición	19
3.3.3. Módulos	20
4. Sobre la semántica	22
5. Conclusión	23
5.1. Trabajos Futuros	23
5.1.1. Síntesis	23
5.1.2. Generación de Múltiples Errores	23
5.1.3. Funcionalidades Adicionales	24

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*, entre 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. $\Delta\Delta$ Lang 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. 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, $\Delta\Delta$ Lang 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 $\Delta\Delta$ Lang 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 1988.

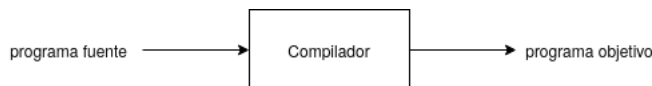


Figura 1.1: Un compilador

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.

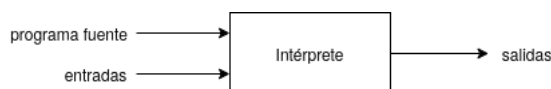


Figura 1.2: Un intérprete

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 *árboles 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.

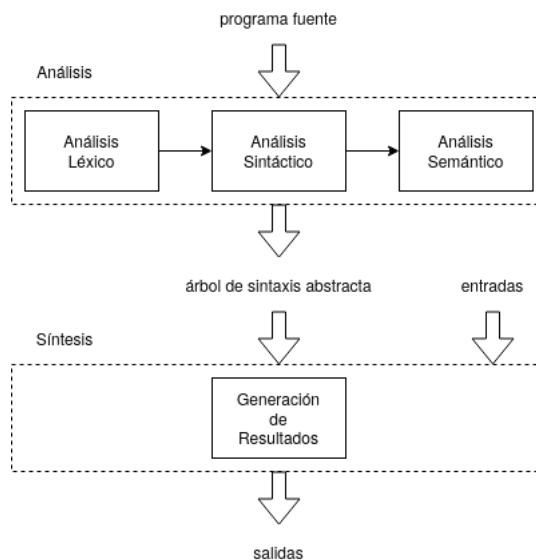


Figura 1.3: Estructura de un intérprete

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 $\Delta\Delta\text{Lang}$. 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 $\Delta\Delta\text{Lang}$ 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.

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

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.

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

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.

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

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.

```

 $\langle operation \rangle ::= \langle expression \rangle \langle binary \rangle \langle expression \rangle \mid \langle unary \rangle \langle expression \rangle$ 
 $\langle binary \rangle ::= + \mid - \mid * \mid / \mid \% \mid || \mid \&\& \mid <= \mid >= \mid < \mid > \mid == \mid !=$ 
 $\langle unary \rangle ::= - \mid !$ 

```

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.

```

 $\langle variable \rangle ::= \langle id \rangle$ 
 $\mid \langle variable \rangle [ \langle expression \rangle \dots \langle expression \rangle ]$ 
 $\mid \langle variable \rangle . \langle fname \rangle$ 
 $\mid * \langle variable \rangle$ 

```

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.

```

 $\langle sentence \rangle ::= \text{skip} \mid \langle assignment \rangle \mid \langle procedurecall \rangle \mid \langle if \rangle \mid \langle while \rangle \mid \langle for \rangle$ 
 $\langle assignment \rangle ::= \langle variable \rangle := \langle expression \rangle$ 
 $\langle while \rangle ::= \text{while } \langle expression \rangle \text{ do } \langle sentences \rangle$ 
 $\langle sentences \rangle ::= \langle sentence \rangle \dots \langle sentence \rangle$ 

```

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.

```

 $\langle procedurecall \rangle ::= \langle id \rangle ( \langle expression \rangle \dots \langle expression \rangle )$ 
 $\mid \text{alloc } \langle variable \rangle$ 
 $\mid \text{free } \langle variable \rangle$ 

```

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*).

```

 $\langle if \rangle ::= \text{if } \langle expression \rangle \text{ then } \langle sentences \rangle \text{ else } \langle sentences \rangle$ 

```

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 $\Delta\Delta$ Lang 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.

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

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

3.2. Sintaxis Concreta

La *sintaxis concreta* de $\Delta\Delta$ Lang, 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.


```

⟨rest⟩ ::= ( ⟨letter⟩ | ⟨digit⟩ | ⟨other⟩ ) *
⟨letter⟩ ::= ⟨lower⟩ | ⟨upper⟩
⟨lower⟩ ::= a | b | ... | z
⟨upper⟩ ::= A | B | ... | Z
⟨digit⟩ ::= 0 | 1 | ... | 9
⟨other⟩ ::= _ | '

```

3.2.2. Azúcar Sintáctico

Existen una serie de construcciones en $\Delta\Delta\text{Lang}$ 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 (\cdot), uno puede hacer uso de esta abreviatura que resulta más conveniente.

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

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.

$$\text{fun } f (\dots a_1 : \theta, a_2 : \theta, \dots, a_n : \theta \dots) \dots \stackrel{def}{=} \text{fun } f (\dots a_1, a_2, \dots, a_n : \theta \dots) \dots$$

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 primera notación, permite omitir el último bloque de sentencias (*else*). La segunda, nos da la capacidad de agregar una cantidad arbitraria de condicionales adicionales (*elif*).

$$\begin{aligned} \text{if } b \text{ then } ss &\stackrel{def}{=} \text{if } b \text{ then } ss \text{ else skip} \\ \text{if } b_1 \text{ then } ss_1 \text{ elif } b_2 \text{ then } ss_2 \text{ else } ss_3 &\stackrel{def}{=} \text{if } b_1 \text{ then } ss_1 \text{ else if } b_2 \text{ then } ss_2 \text{ else } ss_3 \end{aligned}$$

3.3. Parser

Ya nos encontramos en condiciones para describir los detalles principales sobre el desarrollo del parser para $\Delta\Delta\text{Lang}$. 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* (Leijen 2019). 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* (Karpov 2019). 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 *dónde* 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* (GHC Team 2020). 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 $\Delta\Delta$ Lang. 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 $\Delta\Delta\text{Lang}$.

Capítulo 4

Sobre la semántica

(Reynolds 1998) y (Castegren y Reyes 2019).

Capítulo 5

Conclusión

A lo largo de este trabajo desarrollamos la etapa de *análisis* de nuestro intérprete. Las fases de *análisis léxico* y *análisis sintáctico* se implementaron con el parser, mientras que la de *análisis semántico* fue realizada junto con los chequeos estáticos del lenguaje...

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 distintos trabajos futuros relacionados con nuestro objetivo. Algunos estarán destinados a completar la especificación del intérprete, mientras que otros podrán ser realizados con el fin de mejorar y expandir el mismo.

5.1.1. Síntesis

Siguiendo con el desarrollo natural del intérprete, la siguiente fase a diseñar es la de *síntesis*. Una vez terminada la parte *estática*, es hora de comenzar la parte *dinámica*. Para esto, necesitaremos definir formalmente la semántica *small step* del lenguaje, junto con todos los mecanismos necesarios para poder ejecutar nuestros programas. Tendremos que diseñar los chequeos dinámicos del lenguaje, junto con la generación de fallas durante la interpretación del código. Además, se deberá proveer una interfaz de usuario para el uso interactivo del intérprete.

5.1.2. Generación de Múltiples Errores

Unos de los aspectos que se pueden 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 descrita en el mismo artículo en el que se basó el diseño de los chequeos estáticos del intérprete (Castegren y Reyes 2019). 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.

5.1.3. Funcionalidades Adicionales

Existe 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 diversas causas. Algunas debieron ser omitidas por falta de tiempo, otras porque no se llegó a un consenso en su diseño, e incluso algunas por las dificultades encontradas durante su implementación. A continuación daremos una breve descripción de cada una de estas.

Soporte para Múltiples Módulos

Inferencia de Tipos

Estructuras Iterables

Clases

Bibliografía

- Aho, Alfred, Ravi Sethi y Jeffrey Ullman (1988). *Compilers: Principles, Techniques, and Tools*. Addison Wesley.
- Castegren, Elias y Kiko Fernandez Reyes (2019). «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*.
- GHC Team (2020). *The Glorious Glasgow Haskell Compiler*. Ver. 8.10.1. URL: <https://gitlab.haskell.org/ghc/ghc>.
- Karpov, Mark (2019). *Megaparsec*. Ver. 8.0.0. URL: <https://github.com/mrkkp/megaparsec>.
- Leijen, Daan (2019). *Parsec*. Ver. 3.1.14.0. URL: <https://github.com/haskell/parsec>.
- Reynolds, John Charles (1998). *Theories of Programming Languages*. Cambridge University Press.