Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática, Ingeniería del Software

TRABAJO DE FIN DE GRADO

ULAN

Diseño e implementación de un lenguaje de programación matemático

---

Primera convocatoria

Curso 2018/2019

Autor: Javier Castillo Delgado

Tutor: José Antonio Parejo Maestre

Dedicado a mi familia, especialmente a mi madre, por haber creído en mí en todo momento y haberme ayudado a convertir lo que parecía una idea nebulosa y alocada en una realidad palpable.

También le dedico este trabajo a mis amigos, pues siempre han estado ahí para apoyarme de esa manera tan irreverente que en ocasiones los caracteriza. Como bien diría Walo: "Chu, Gabie, chu…".

Por todo esto,

Gracias de corazón.

Índice

Sección I

Introducción

Antes de empezar

Contexto y motivación

El diseño de lenguajes es una rama que ha avanzado mucho en las últimas décadas. Tan solo hay que realizar una búsqueda rápida para darse cuenta de que hay cientos de lenguajes de programación desarrollados en numerosos ámbitos y herramientas que permiten hacer el proceso de implementación más sencillo (generadores de *parsers*, por ejemplo).

En la actualidad hay muchos tipos de lenguajes orientados a diferentes fines, lo cual genera una gran variabilidad de paradigmas. Haciendo una distinción por estos paradigmas, podríamos decir que la mayoría son lenguajes imperativos con funcionalidades estándar, pero existen otros muchos que se alejan de esta forma de ver la programación:

- **Programación funcional:** quizás esta sea el paradigma más famoso tras el imperativo. Este basa su estructura en funciones modeladas a un nivel más matemático y puro, por lo que quedan prohibidos los "efectos externos". Gracias a esta manera de ver un programa, podemos realizar optimizaciones basadas en sustituciones o en paralelismo [1]. Algunos ejemplos de lenguajes funcionales podrían ser *Haskell* [2] o *Scheme*.
- **Programación lógica:** este paradigma consiste en programar definiendo restricciones que acaban siendo modeladas o traducidas a predicados de lógica de primer orden. Tras esto, estas restricciones guían a un algoritmo de resolución genérico a obtener una respuesta correcta. Este tipo de código hace que tareas como resolver un sudoku adquieran una perspectiva completamente distinta. Uno de los máximos exponentes de este paradigma es *Prolog*, extendiéndose hasta el punto de tener diferentes variantes.
- **Programación simbólica:** se dice que un lenguaje de programación es simbólico cuando puede manejar *símbolos* además de valores. Si bien es cierto que es complejo de explicar y hay diferentes interpretaciones, podría decirse que estos lenguajes permiten manejar sus propias estructuras internas y/o pueden manejar la incertidumbre mediante el uso de expresiones semievaluadas. Un claro ejemplo es *Wolfram Language* [3] (lenguaje utilizado en *Mathematica*), el cual es ampliamente utilizado en entornos de investigación.

Estas son las principales influencias que se han tenido a la hora de diseñar Ulan, siendo el desencadenante el descubrimiento de la **programación de conceptos**, una filosofía a la hora de programar que consiste en hacer estructuras en el código que se asemejen lo máximo posible a lo que representan en el mundo real, intentando en todo momento reducir todo lo posible la ambigüedad y el *ruido sintáctico y semántico* [4]. El lenguaje de programación XL [5] fue desarrollado con esta intención en mente, por lo que en él se pueden describir toda clase de entidades de una manera relativamente intuitiva.

Todo el diseño de Ulan fue centrado en seguir esta filosofía y hacer que el programador pudiera definir todo lo relativo a las estructuras del lenguaje a varios niveles, por lo que se podría decir que este es el segundo lenguaje creado específicamente con la intención de implementar una programación de conceptos eficiente. Dicho esto, en este documento no se describe un clon de XL, sino que se pretende darle un enfoque diferente a esta filosofía a la que le dio Christophe de Dinechin con su lenguaje.

La principal diferencia entre estos dos lenguajes es la facilidad a la hora de definir artefactos de código o *tokens* a diferentes niveles. Mientras XL podría necesitar plugins para el compilador, Ulan tan solo necesitaría algunas líneas de código en el propio lenguaje. Asimismo, los lenguajes de extracción de datos que Ulan utiliza permiten una expresividad que muchas veces es complicada de conseguir únicamente con homoiconicidad (manejo de la estructura del código usando el propio lenguaje).

Utilizando los conceptos de la versión actual de intérprete se puede asignar una gran cantidad de funcionalidad, pero versiones futuras podrían permitir asignaciones semánticas a prácticamente cualquier árbol de sintaxis abstracta.

Objetivos

El objetivo principal de Ulan es demostrar que es posible reunir características interesantes de los lenguajes anteriormente mencionados en un lenguaje imperativo eficiente y seguro. También se pretende que estos conceptos estén bien modularizados para que sea posible fragmentar el aprendizaje sin comprometer otros aspectos por ello. Además, se irá un paso más allá para permitir definir la propia sintaxis del lenguaje mediante el uso de un lenguaje interno, USDL.

Esta última es la característica distintiva de este lenguaje, la definición de conceptos de a diferentes niveles. Se pretende que, en sus versiones finales, se permita definir la sintaxis y semántica exactas de una gran cantidad de conceptos representables mediante código. Además, se pretende que definir estas estructuras sea algo natural y que no hagan falta herramientas externas. Todo lenguaje auxiliar estará implementado en el propio intérprete.

El diseño de este lenguaje tiene una clara orientación matemática, hablándose de tipos nativos de manejo de expresiones algebraicas, números con precisión arbitraria por defecto y asimilación sencilla de dialectos nuevos. Ver Ulan como un **lenguaje para prototipar lenguajes** no es ser descabellado, ya que su sintaxis extensible lo hace ideal para estos propósitos.

Página web

Documentación más orientada al aprendizaje de una manera amistosa ha sido recopilada en una página web construida tras la finalización del intérprete. Esta puede ser accedida utilizando el siguiente enlace:

https://ulan-language.herokuapp.com/

Requisitos

Dada la naturaleza del proyecto, se considera que utilizar las plantillas para requisitos orientadas a sistemas de información no proporcionará una solución óptima, por lo que se opta por redactarlos siguiendo una plantilla propia. La lista completa se muestra a continuación:

- **Capacidades básicas de leguajes de programación imperativos:** definición de variables, expresiones básicas de control de flujo, clases básicas para uso común, definición de estructuras y procedimientos propios…
- **Capacidades básicas de lenguajes funcionales:** pese a no ser un lenguaje funcional, se incluyen primitivas y librerías que permiten operar con expresiones funcionales (composición, binding…).
- **Capacidades simbólicas:** manejo interno de expresiones numéricas con incógnitas. Se incluyen algoritmos de simplificación, cálculo de raíces, representación gráfica, álgebra booleana y resolución lógica.
- **Comprensión de listas extendida:** capacidades similares a lenguajes como *Python* en lo referente a creación y manejo de estructuras serializadas, pero con una sintaxis más expresiva y más tipos de estructuras de datos optimizadas a bajo nivel.
- **Sintaxis extensible:** se permite extender la sintaxis del lenguaje utilizando un lenguaje interno llamado USDL, el cual fue diseñado con ese único objetivo en mente. También se permite utilizar este lenguaje para extracción de datos y parsing. Igualmente, es posible definir estructuras que normalmente suelen ser fijas, tales como los operadores.
- **Estructuras personalizables:** las estructuras en este lenguaje pueden poseer unas abstracciones algo diferentes a la de lenguajes comunes, incluyendo aspectos que acortan el tiempo de desarrollo. Un claro caso es el de la conmutatividad de operadores.
- **Tipado gradual:** se permite utilizar variables y parámetros no tipados para obtener un tiempo de desarrollo más corto. Un algoritmo de inferencia de tipos permite realizar optimizaciones incluso cuando se usa esta forma de programar.
- **Aritmética con precisión arbitraria:** la librería numérica estándar de este lenguaje permite hacer operaciones con números grandes sin limitación teórica. Esta librería es única para este lenguaje.
- **Orientación matemática:** se incluyen numerosas clases para cálculos matemáticos de ámbito científico. Si se incluyen las librerías estándar podemos hablar de operaciones con números complejos, manejo de grafos o transformaciones lineales para gráficos.
- **Capacidades gráficas:** se incluye una librería gráfica con unos algoritmos que permiten representar datos. Sin ir más lejos, el propio logo del lenguaje está definido como un programa Ulan.

Tecnologías utilizadas

Lenguaje de desarrollo

Todas las librerías básicas utilizadas en este proyecto están desarrolladas en C++17, utilizando el compilador GCC (MinGW-W64). Varias estructuras están basadas casi en su totalidad en el estándar que ofrece este lenguaje (String y Container, por ejemplo). Uno de los objetivos principales en cuanto a implementación era no utilizar nada fuera del estándar para evitar dependencias de librerías externas, por lo que no hay ninguna librería fuera del mismo que no haya desarrollado personalmente.

Para las secciones del documento que describen detalles de implementación se presupone un nivel avanzado de conocimientos sobre el lenguaje de implementación. Si el lector desea introducirse en este lenguaje, se recomienda el libro introductorio escrito por el propio creador del lenguaje [6]. Para detalles más técnicos, se recomienda la última versión del estándar oficial hasta la fecha del lenguaje [7].

Entorno

El entorno utilizado para desarrollar el intérprete es Code::Blocks [8], utilizando la última versión estable disponible hasta la fecha, 16.01 revisión 10702. Se utilizaron los plugins estándar del IDE. Se prescindió del uso de cualquier librería no perteneciente al estándar para evitar tener problemas futuros con cambios de implementación y para hacer un software a nivel más personal.

Se utilizó la herramienta CppCheck [9] para encontrar errores de estilo en el código y realizar optimizaciones sencillas. Pese a haber dedicado mucho tiempo a buscar una herramienta de *profiling*, no conseguí encontrar ninguna que funcionase correctamente para el contexto, por lo que no utilicé ninguna.

La herramienta utilizada para generar los diagramas de clases y cualquier otra documentación gráfica del código que aparezca en este documento fue Doxygen [10] con la ayuda de su conexión con Graphviz [11]. Su principal uso fue para la generación automática de diagramas de herencias.

La mayoría de la documentación gráfica no referente al código (grafos, diagramas aclarativos, etc.) fue diseñada con la herramienta web draw.io [12]. Siguiendo los términos de uso de esta página, cualquier imagen realizada con su ayuda es libre de aparecer en este documento.

Los fragmentos de código con coloreado de sintaxis fueron obtenidos utilizando un plugin estándar del IDE que exporta el código a formato rtf. Esto se hace para tener la mayor consonancia posible entre el código y el documento.

Las capturas de pantalla con código Ulan son del editor de texto Sublime Text 3 [13] con un módulo de sintaxis creado específicamente para este lenguaje. En este editor se desarrolló todo el código de las librerías básicas.

Estructura del documento

Secciones

Cada sección del documento contiene una parte del proyecto claramente diferenciada del resto. A continuación se describe brevemente cada una de ellas:

Sección I. **Introducción:** información básica sobre el trabajo.
Sección II. **Planificación:** descripción de la evolución temporal del proyecto.
Sección III. **Diseño:** detalles sobre las decisiones de diseño de Ulan.
Sección IV. **Implementación:** detalles sobre la implementación del intérprete.
Sección V. **Validación y resultados:** ejemplos de programas en Ulan y pruebas.
Sección VI. **Conclusiones:** comentarios finales sobre los resultados y lo aprendido.

La mayoría de secciones son pequeñas y fáciles de comprender, pero las secciones III y IV podrían plantear un problema, por lo que se comentan de manera más profunda a continuación.

Sobre el diseño

Esta sección está planteada para comenzar construyendo el lenguaje desde cero, pero asumiendo que el lector conoce conceptos básicos tales como *scopes*, *variables* o *tokens*. Todos estos serán utilizados durante la descripción de la sintaxis a diferentes niveles.

Tras definir la sintaxis se pasará a definir cada una de las estructuras internas que el lenguaje maneja como abstracciones (funciones, clases, casts…). Estos serán definidos en profundidad en función de la sintaxis construida anteriormente. También se incluirá una lista completa de las estructuras por defecto.

Una parte importante es la de USDL, ya que tiene su propia estructura. Aquí se define por completo la estructura del sublenguaje siguiendo una estructura similar a la del propio documento.

Sobre la implementación

En esta sección se habla sobre las técnicas utilizadas durante el desarrollo a nivel de programación y sobre el modelado de los datos que maneja el intérprete. También se dará una descripción superficial de las librerías básicas del lenguaje, las cuales se asumirá que fueron desarrolladas con anterioridad.

Cabe destacar el apartado de USDL, que sigue el mismo patrón que el apartado con el mismo nombre en la sección anterior. Todo lo definido en esta parte tendrá una estructura similar a la del mismo documento que la contiene.

Sección II

Planificación

Metodología

Desarrollo

Para realizar el proyecto se ha optado por utilizar metodologías ágiles por la flexibilidad que permiten. Concretamente, se ha utilizado Scrum para la construcción del código del proyecto. Por otra parte, el diseño del lenguaje se hizo con anterioridad a la fecha de inicio del proyecto y se considera no cuantificable por su complejidad temporal.

Dada la especial naturaleza del equipo de proyecto (está compuesto por un único miembro) y las dificultades que plantea que el Product Owner sea el propio director, se ha adaptado la metodología de trabajo consecuentemente.

Las reuniones diarias han sido, por tanto, omitidas y las reuniones de *Sprint Planning* y *Sprint Review* han sido unificadas, realizándose durante las reuniones de seguimiento del trabajo. Esto ayuda a reducir la frecuencia de las mismas y a agilizar un desarrollo que no tiene por qué ser tan enrevesado en este caso.

Control de versiones

La gestión de código se ha hecho en un repositorio privado de gitlab, procurando en todo momento obtener una versión completamente funcional cada vez que se añade una característica nueva. Siguiendo esta norma, al final de cada sprint se tiene una *release* que se puede someter a un proceso de pruebas automáticas.

Planificación temporal

Duración y fechas

El proyecto comienza el día 15 de febrero de 2018 y tiene como fin programado el día 15 de octubre de 2018, momento en el que el código debería estar terminado y se debería poder acceder a una versión estable del intérprete. Este tiempo fue dividido en cuatro iteraciones de dos meses de duración:

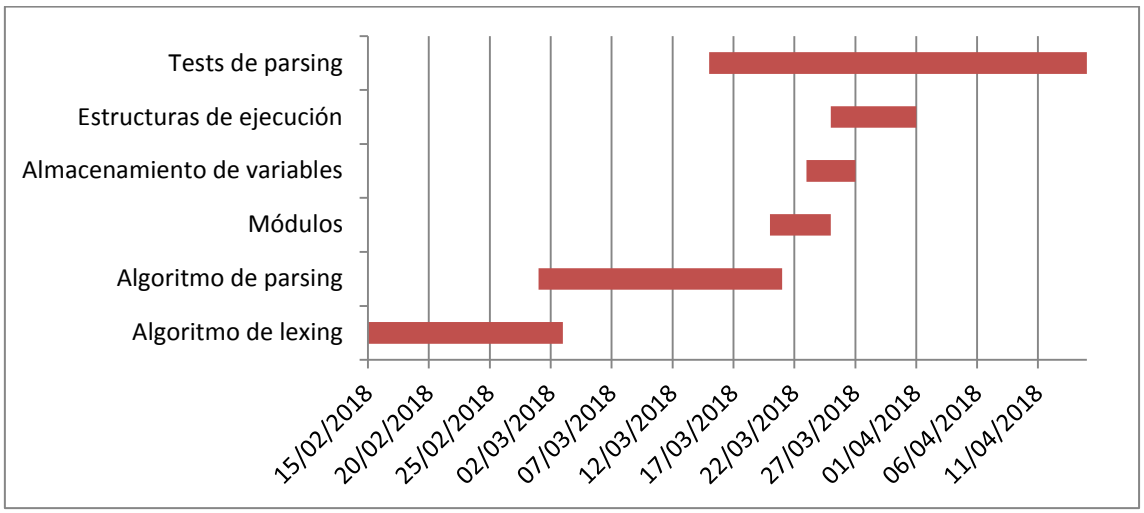
Iteración	Inicio	Fin
Primera	15 de febrero	15 de abril
Segunda	15 de abril	15 de junio
Tercera	15 de junio	15 de agosto
Cuarta	15 de agosto	15 de octubre

Se puede observar que la duración de las iteraciones es superior a lo habitual para esta metodología, esto se debe a la particular naturaleza del proyecto desarrollado. Fueron necesarios periodos largos para asegurar la calidad de las *releases*.

Cronogramas

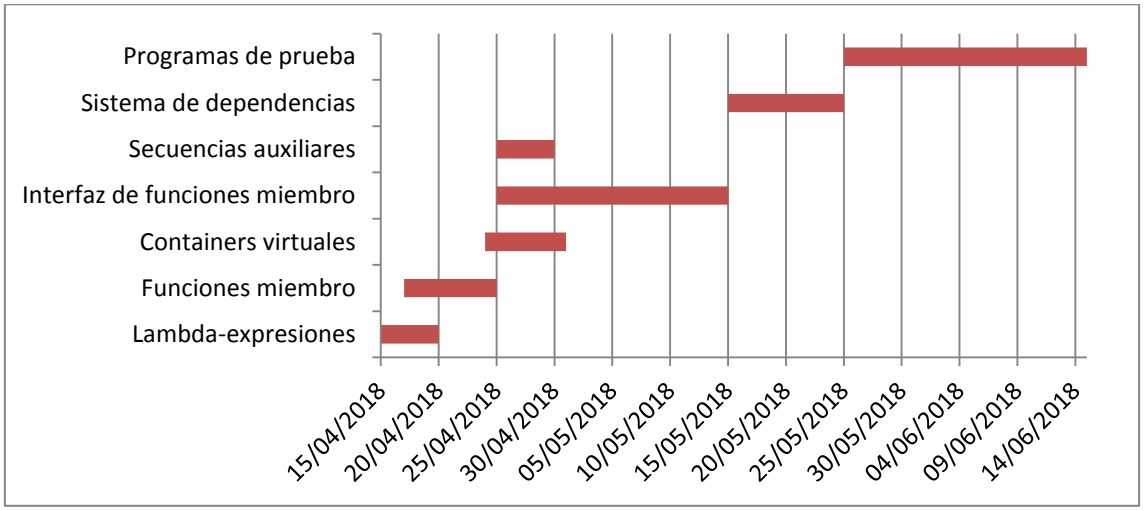
Primera iteración

Desarrollo de las estructuras básicas de interpretación y almacenamiento de código y de los contenedores de abstracciones del lenguaje. Al final se dedicará un tiempo al desarrollo de una suite de pruebas automáticas.



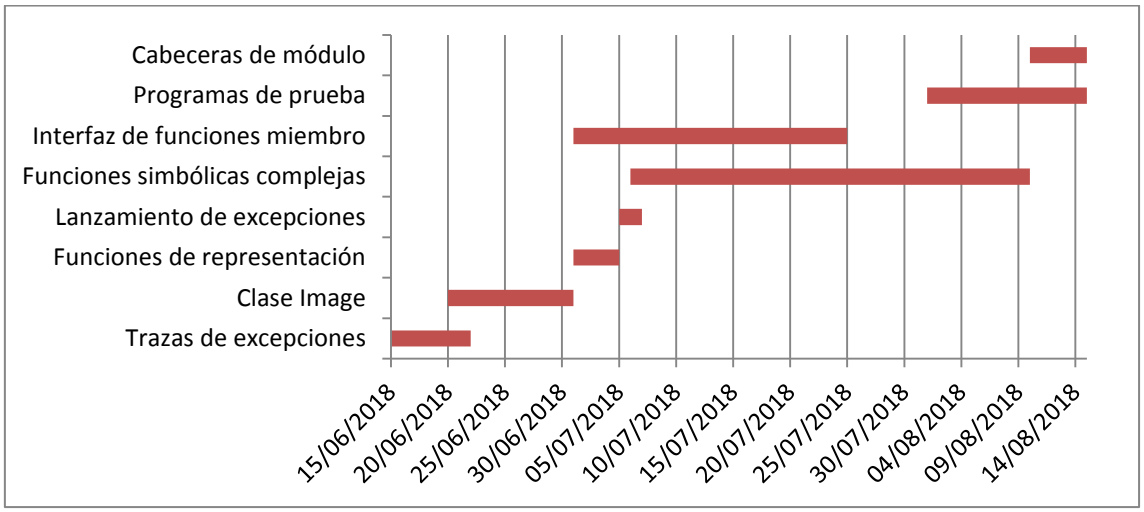
Segunda iteración

Extensión de las estructuras definidas en la iteración anterior e interfaz de funciones de la mayoría de las clases. Al final se dedicará un tiempo a realizar programas que servirán como prueba de que el lenguaje funciona correctamente.



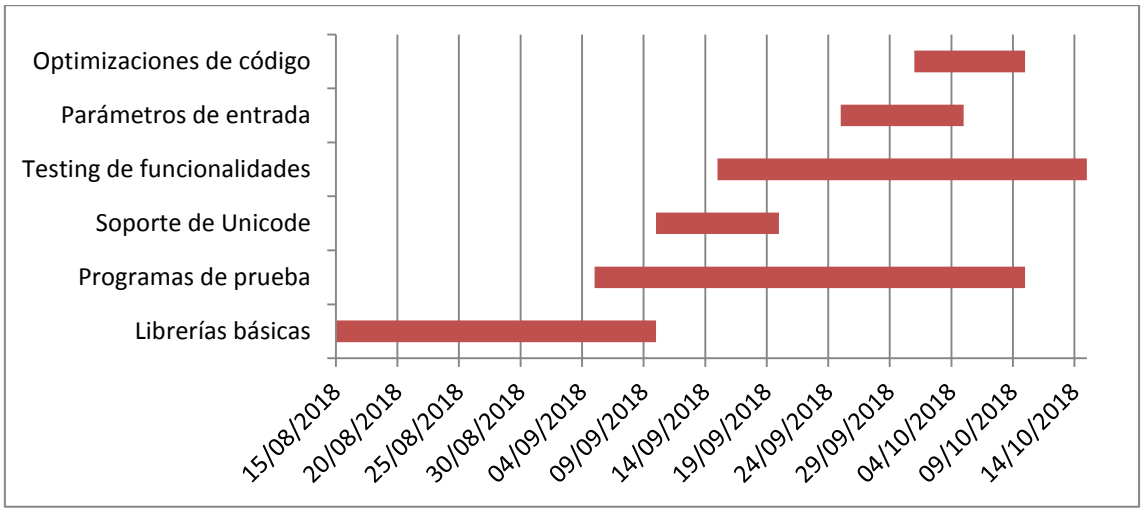
Tercera iteración

Finalización de las interfaces de funciones de las clases definidas y realización de funciones con connotaciones matemáticas más complejas. El tiempo dedicado a los programas de prueba es menor.



Cuarta iteración

Desarrollo de las librerías básicas restantes y de los programas de prueba que constituirán la sección de resultados, finalización de la interfaz del ejecutable, modificaciones de código y test final de funcionalidades.



Costes

Horas imputadas

Este proyecto empezó como una iniciativa propia, por lo que su desarrollo se sale parcialmente del entorno académico y ha resultado en un número mayor de horas de las que supuestamente un trabajo de final de grado debería tener. Este sobreesfuerzo fue realizado de manera completamente voluntaria, sin ningún tipo de presión por parte de la dirección del proyecto.

El desarrollo, si bien es cierto que tenía fechas de entrega debido a las iteraciones, ha transcurrido de manera bastante fragmentada, por lo que es complicado saber el tiempo exacto que se le ha dedicado. Además, este tiempo excluye el desarrollo de librerías tales como *Number* o *Symbolic*.

Dicho esto, se considera que las horas imputadas a la parte del desarrollo del intérprete y la integración de las librerías ya desarrolladas son aproximadamente 500, pero se trata de una medida imprecisa.

Asunciones

Dada la naturaleza del proyecto, se asumirá que de los 242 días que ha durado la construcción, alrededor de 60 fueron días laborables comunes. Esto se debe a que vamos a realizar los cálculos como si se tratara de jornadas completas de 8 horas, y normalmente el desarrollo no se ha realizado de esa manera tan constante.

Suponemos que el desarrollo de este proyecto lo ha hecho un equipo compuesto por un único programador junior para ser consistentes con la realidad del desarrollo. Asimismo, también se obviarán gastos de gestión y se estimarán únicamente los gastos por sueldos, ya que se trata de un proyecto con un equipo algo singular y difícil de medir.

Resultados

Utilizando datos de diferentes empresas podemos establecer un sueldo para un programador junior de alrededor de $40{,}000€$ al año si suponemos que trabaja a jornada completa. Esto se traduce en alrededor de $3{,}200€$ al mes. Siguiendo esta cadena podemos llegar a la conclusión de que un programador junior cobra alrededor de $20€/h$. Si bien es cierto que estos cálculos no son especialmente precisos, se debe tener en cuenta que es complicado realizar estas estimaciones.

Siguiendo estos números especificados anteriormente podemos llegar a la conclusión de que el precio aproximado del proyecto es $60\ días \cdot 8h \cdot 20€/h = 9600€$.

Sección III

Diseño

Sintaxis a nivel de texto

Introducción

Es importante distinguir los diferentes tipos de sintaxis en Ulan, ya que esta puede ser extendida de diferentes maneras. Cuando hablamos de sintaxis a nivel de texto nos referimos a la información necesaria para realizar un *lexing* o *tokenización* correctos del lenguaje.

Aspectos de nivel superior

A la hora de interpretar una línea de código es necesario aplicar un algoritmo de lexing, pero también es necesario tener ciertas consideraciones que funcionan a un nivel superior que este algoritmo. A continuación se listan estos aspectos.

Espacios

Es necesario especificar a qué nos referimos con un espacio en Ulan. En este caso, se define un espacio como cualquier carácter que la función estándar *isspace* detecte. Los espacios encadenados serán interpretados como uno solo por la sintaxis estándar, pero es posible definir sintaxis propias que necesiten varios espacios encadenados para ser detectadas correctamente.

Separación de tokens

Se dice que dos tokens están separados y se detectarán como artefactos diferentes cuando ocurre uno de los dos casos siguientes:

- En el caso de que un token tenga un borde (carácter del principio y/o el final) alfanumérico:
  - Si se quiere separar de otro borde alfanumérico se deberá introducir al menos un espacio entre los dos.
  - Si se quiere separar de un borde no alfanumérico no es necesario introducir espacios.
- En el caso de que se quiera separar un token que tenga un borde no alfanumérico de otro con borde no alfanumérico, estaremos hablando de separar operadores. Estos tokens tienen un algoritmo de separación automática implementado, pero es posible separarlos con espacios si el resultado no es el esperado.

Expresiones multilínea

Pese a no ser algo particularmente idiomático, se permite definir expresiones que ocupen más de una línea utilizando una secuencia de caracteres especial. Cualquier línea será concatenada con la siguiente si termina en " \"(espacio, barra inversa).

Comentarios

Los comentarios en Ulan se definen de la misma manera que en la mayoría de lenguajes estándar. Cualquier carácter que se encuentre delante de "//" en el código será ignorado, haciendo de esa cadena un comentario.

Contextos

Definición

Uno de los conceptos más importantes de la sintaxis de Ulan son los contextos. Podríamos referirnos a los contextos como el trozo de código desde el que es referenciable una variable recién definida, siendo este concepto equivalente a los *scopes* o *ámbitos* en lenguajes como C o C++.

Al contrario que en estos lenguajes, Ulan delimita los contextos utilizando indentación. Cualquier bloque de código que esté más indentado que los demás pertenecerá a un contexto superior. Por supuesto, un contexto puede contener otros contextos de nivel superior, al igual que en cualquier otro lenguaje.

Es posible especificar un contexto que retorne un valor al contexto superior. Estos son llamados cuerpos de funciones y son los únicos contextos en los que se permite la sentencia *return*. Se podrá acceder a los datos de contextos superiores siempre que no haya un cuerpo de función en la cadena de la jerarquía. La única excepción de esta regla es el contexto primario, cuyos datos siempre son accesibles.

$$\boldsymbol{Ej}: Contexto\ primario \rightarrow Función\ f_1 \rightarrow Función\ f_2 \rightarrow Bucle\ for \rightarrow Sentencia\ if$$

En el ejemplo dado, tan solo los contextos rojos son accesibles desde la sentencia *if*.

Contexto primario

Se dice que el contexto primario es el contexto con nivel 0 o, dicho de otra forma, el contexto que no está indentado. Cualquier variable definida en este contexto será considerada global, por lo que no se borrará de la memoria hasta que finalice el programa o se fuerce su eliminación. Asimismo, una variable definida en este contexto será accesible desde cualquier parte del código.

Cuerpo de clase

Existe un tipo de contexto especial utilizado para definir los atributos de una clase tras una secuencia *Define class*. Estos contextos solo pueden contener una variable opcionalmente tipada por línea, representando cada una de estas un atributo de la nueva clase a definir. Si se intenta introducir un tipo de expresión diferente al mencionado, la interpretación resultará en un error de sintaxis.

Restricciones

Todas las variables que sean definidas en un contexto serán eliminadas de la memoria en el momento en el que el contexto se cierre. Si alguno de los objetos tiene una referencia hacia un bloque de datos fuera del contexto, se mantendrá en memoria hasta que ese bloque se quede sin referencias.

También es necesario destacar que la definición de estructuras está prohibida para cualquier contexto diferente al primario. Esto se hace para poder evitar la serialización de definiciones, ya que podrían sobrecargar el sistema si se abusa.

Palabras reservadas

Listado completo

La lista de las principales palabras reservadas en Ulan se muestra a continuación en orden alfabético junto al ámbito general en el que se utiliza:

Nombre	Ámbito general de uso
As	Definición de estructuras y sobrecarga
Const	Definición de funciones miembro
Define	Definición de estructuras
Else	Análisis de casos contrarios a los condicionales especificados
For	Bucles y comprensión de listas
From	Definición de sintaxis propias
If	Análisis de casos condicionales
Is	Comprobación de tipos
In	Bucles y comprensión de listas
Return	Retornar un valor al contexto de llamada
While	Bucles

Aunque su ámbito de uso es completamente diferente, los subtipos de Container también son palabras reservadas. A continuación se especifica la lista de subtipos:

Nombre	Descripción
Deque	Cola doblemente terminada
List	Lista encadenada
Set	Conjunto
Vector	Array dinámico
SortedDeque	Cola doblemente terminada ordenada
SortedList	Lista encadenada ordenada
SortedSet	Conjunto ordenado
SortedVector	Array dinámico ordenado

Se considera que los nombres de las clases en Ulan son palabras reservadas. Estas tienen un significado sintáctico especial y van aumentando conforme se definen nuevas clases. El conjunto por defecto es el siguiente:

Nombre	Descripción
Number	Clase básica numérica
Matrix	Clase básica de matrices numéricas
String	Clase básica de cadenas de caracteres
Boolean	Clase básica de lógica booleana
Symbolic	Clase básica de expresiones algebraicas
Container	Clase básica de manejo de objetos serializados
Mapper	Clase básica de array asociativo
Comparator	Clase básica de criterios de comparación
Functional	Clase básica de expresiones funcionales
Image	Clase básica de manejo de gráficos

Pese a que no son clases internas, estrictamente hablando, los siguientes identificadores también pueden ser considerados tipos:

Nombre	Descripción
Void	Identificador de ausencia de retorno
Any	Identificador de cualquier tipo
None	Etiqueta de un objeto vacío (Uso interno)

También se utilizan palabras reservadas para identificar las estructuras que se están definiendo en las secuencias con *Define*. La lista de estas se muestra a continuación:

Nombre	Descripción
Class	Clase
Cast	Conversión de tipos
Function	Función estándar
Operator	Operador
Operation	Operación
Syntax	Sintaxis propia

Una serie de palabras reservadas de un uso muy cerrado son las categorías en las que se pueden separar los operadores y sus características internas. Estas son utilizadas estrictamente en las definiciones de este tipo de estructura y son las siguientes:

Nombre	Descripción
Full	Operador completamente conmutativo
Left	Operador tipo Left
Right	Operador tipo Right
Infix	Operador infijo
Prefix	Operador prefijo
Suffix	Operador sufijo
Prec	Precedencia

Es importante mencionar que la distinción que se ha hecho a la hora de listar las palabras reservadas es únicamente estética y no implica necesariamente que una secuencia real no pueda mezclar palabras de diferentes tablas. Un ejemplo claro de esto son las secuencias inline de comprensión de listas, las cuales pueden usar la palabra reservada *if* pese a no estar analizando un caso condicional de manera exacta.

Detección

El propósito principal de Ulan es ser un lenguaje potente con un gran poder expresivo, por lo que tiene sentido que se permita al programador utilizar tipos de sintaxis propios de lenguajes antiguos. Precisamente por esto, la detección de palabras reservadas es independiente de mayúsculas y minúsculas, con la excepción de los nombres de clases.

Elementos básicos

Para poder añadir sintaxis al lenguaje, primero hay que comprender la sintaxis básica del mismo. A continuación se especifican los elementos básicos que forman parte del código en Ulan, así como una breve descripción de cada uno.

Encapsuladores

Los encapsuladores de código son caracteres que, tal y como se puede inferir del nombre, encapsulan código para diversos fines. Cualquier código encapsulado se tratará de manera diferente al no encapsulado, variando este tratamiento según el encapsulador que se haya utilizado. A continuación se listan los encapsuladores definidos en Ulan:

Nombre	Apertura	Cierre	Descripción
Paréntesis	(	)	Los paréntesis agrupan código que debe ejecutarse con una prioridad diferente a la normal. Tienen un funcionamiento estándar y se usan sobre todo para alterar el orden de evaluación de operadores. También se utilizan para hacer *llamadas*
Llaves	{	}	Las llaves encapsulan los objetos que se encuentran dentro de un Container. Estos van separados por comas y pueden ser expresiones complejas. Existe la posibilidad de usar las llaves para mecanismos de comprensión de listas
Corchetes	[	]	Los corchetes encapsulan los argumentos de una tupla de alto nivel, estando estos están separados por comas. Se usan para representar capturas de lambda expresiones y argumentos en ciertas secuencias
Comillas	“	“	Las comillas encapsulan el texto perteneciente a un objeto de tipo String. Cualquier texto que se encuentre encapsulado de esta manera no será tratado como código, por lo que no estará sometido a la restricción de espacios múltiples

Operadores

Se define un operador a nivel de sintaxis como una subcadena que cumple unas ciertas restricciones de formato y que se ha definido anteriormente en el código. Estos son de los primeros elementos del código en detectarse.

Clases directas

Cuando hablamos de clases directas, nos referimos a clases con un parsing directo definido en el intérprete. Estas serán interpretadas como objetos y serán tratadas como un bloque de código que debe ser parseado con una función específica. Se podría decir que son el equivalente a los literales en lenguajes como C o Java, pero con la diferencia de que en Ulan se pueden definir nuevos elementos de este tipo utilizando USDL.

Elementos contextuales

Se define un elemento contextual como una subcadena que sigue ciertas normas de formato y que depende del código anterior para ser interpretado. Los tipos de elementos contextuales se listan a continuación:

- **Tipo:** cadena que coincide con el nombre exacto (sensible a mayúsculas y minúsculas) de una clase definida anteriormente en el código. Aparecen principalmente en definiciones de variables y parámetros tipados.
- **Palabra reservada:** cadena que coincide con cualquier palabra reservada que no sea una clase.
- **Nombre de función:** cadena que coincide con el nombre exacto dado a una función definida anteriormente en el código. Las expresiones funcionales quedan excluidas de esta definición.
- **Variable:** cadena que coincide con el nombre de una variable definida anteriormente en el código. Cualquier cadena que cumpla las restricciones de formato y no sea ningún otro elemento contextual se considerará una variable no definida.

Sintaxis propia

Se define una sintaxis propia como una sintaxis USDL utilizada en una definición especificada anteriormente en el código. Dicho de otra forma, son las funciones con sintaxis propia que se definieron utilizando este lenguaje antes de interpretar la línea de código actual.

Sintácticamente hablando, este elemento es idéntico a las clases directas, pero a nivel semántico son diferentes. Además, se distingue entre estos dos elementos a la hora de ejecutar el algoritmo de parsing por sus implicaciones.

Serialización

Se dice que una cadena es una serialización cuando se trata de una serie de subconjuntos de otros elementos básicos separados por comas. Estas tan solo tienen sentido cuando se trata de código encapsulado, pudiendo distinguirse varios tipos:

- **Argumentos de llamada (encapsulamiento en paréntesis):** se utilizan para hacer llamadas a funciones y operaciones de cálculo.
- **Tuplas de alto nivel (encapsulamiento en corchetes):** se utilizan en ciertos contextos en los que un paréntesis no es estéticamente correcto o es confuso. Se pueden ver en bucles multifor y capturas de lambda expresiones.
- **Definición de Container (encapsulamiento en llaves):** se utilizan para definir los valores de un Container de manera directa.

Una serialización de un único elemento es equivalente a la expresión interna y se tiene en cuenta a la hora de aplicar el reconocimiento de patrones. Asimismo, una serialización vacía es un concepto válido en la sintaxis de Ulan, pero tan solo tiene un valor simbólico en encapsulaciones vacías.

Formatos

Elementos contextuales

Las normas a la hora de decidir si una subcadena se trata de un elemento contextual son las especificadas a continuación:

- La cadena no debe contener espacios.
- Se permite cualquier carácter Unicode en la cadena (UTF-8).
- Solo se permiten caracteres alfanuméricos (letras latinas o letras griegas), con la excepción del carácter "_".
- Los nombres de clases deben empezar por una letra, sea esta mayúscula o minúscula

Ejemplos:

$$"BinaryFunction"$$

$$"date"$$

$$"Boolean\_array"$$

Operadores

La cadena de caracteres que contiene la representación de un operador debe seguir ciertas reglas de formato. Estas reglas se especifican a continuación:

- La cadena no debe contener ningún carácter encapsulador de código para evitar confundir al intérprete.
- Los operadores alfanuméricos están aceptados, pero solo se detectarán correctamente cuando se introduzcan entre espacios en el código.
- No se permiten espacios múltiples en la representación de un operador, ya que el propio intérprete eliminará de cualquier línea de código las sucesiones de espacios.
- Se permite cualquier carácter Unicode en la cadena (UTF-8).
- La representación del operador no puede ser la misma que la de otro operador, pero con espacios al principio y/o al final. La razón de esto es que podría provocar graves problemas de legibilidad.

Ejemplos:

$$" + + + "$$

$$"or"$$

$$" \$op\$ "$$

Sintaxis a nivel de tokens

Introducción

La representación como árbol de *tokens* obtenida durante la fase de *lexing* es tan solo una representación intermedia que es interpretada para obtener una expresión final con semántica clara. A continuación se definirán las estructuras básicas de la sintaxis de Ulan a un nivel ligeramente superior.

Expresiones

Retorno

Se dice que una expresión retorna un valor cuando se puede extraer un elemento perteneciente a alguna clase Ulan de su interpretación y ejecución. No todas las expresiones tienen retorno. Internamente, se considera que una expresión sin retorno devuelve un objeto de tipo Void.

Expresión cabecera

Se dice que una expresión es una cabecera cuando no tiene sentido por sí misma, sino que presenta un contexto para las líneas de código inferiores. Estas expresiones no tienen retorno. Las únicas líneas de código que presentan esta característica son las secuencias controladoras de flujo (*if*, *else*, *for*…) y las cabeceras de definición de estructuras.

Tipo

Una vez identificada una sintaxis correcta y la coherencia de elementos, se pueden distinguir tipos de expresiones a la hora de representarlas en forma de árbol:

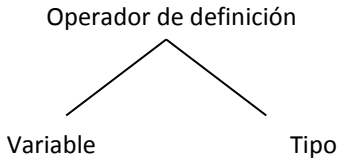
- **Valor directo:** se trata de una expresión simple que consiste en un único elemento semántico del que se puede extraer un valor directamente del parseo.
- **Valor indirecto:** se trata de una especie de plantilla de la que se puede extraer un valor tras hacer ciertos cálculos. Un ejemplo es la definición de Containers.
- **Variable:** la expresión es una referencia a una variable, esté definida o no.
- **Operación:** se trata de un conjunto de operadores iguales con expresiones intercaladas.
- **Secuencia:** se trata de un conjunto de palabras reservadas con expresiones intercaladas. Un claro ejemplo son los controladores de flujo o la sentencia *return*.
- **Funciones:** se dice que una expresión es de este tipo cuando consiste en una llamada a alguna función definida anteriormente.
- **Sintaxis propia:** se habla de expresiones de este tipo cuando su sintaxis coincide con una definida en USDL anteriormente. Se excluyen de esta definición las sintaxis directas de clases.

Elementos básicos

Variable

En Ulan, se define una variable como una zona de memoria con un nombre asociado en la que se pueden almacenar datos con ciertas estructuras predefinidas para su posterior actualización o recuperación. Este es un concepto básico de la mayoría de lenguajes de programación, no se ha cambiado su semántica, pero cabe destacar que vas variables en Ulan pueden no tener un tipo asociado.

Cualquier estructura cuya sintaxis sea identificada como el nombre de una variable (esté definida o no) será interpretada como una variable no tipada a nivel semántico. También pueden ser un nombre de función dependiendo del contexto. Asimismo, las variables tipadas se representan en el árbol de sintaxis de la siguiente forma:



Clase directa

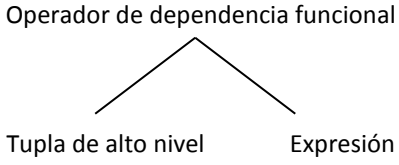
Se define como una clase directa a cualquier sintaxis directa de una clase (sea esta básica o propia). Por lo general, la relación con la sintaxis de este concepto semántico es bastante clara, ya que se considerarán clases directas a los elementos sintácticos con el mismo nombre.

Todos los elementos semánticos de este tipo serán considerados valores directos en el árbol de representación de código final, ya que se puede extraer un valor directamente de la propia sintaxis.

Clase indirecta

Se dice que una clase es indirecta cuando no tiene una sintaxis directa definida, por lo que se deben instanciar objetos de estos tipos de maneras especiales. Dependiendo del tipo de instanciación se pueden dividir en dos tipos:

- **Funcionales:** se instancian mediante el uso de una o varias funciones. Un claro ejemplo son los objetos de tipo Comparator.
- **Por patrón:** se instancian mediante el reconocimiento de patrones en el árbol de sintaxis abstracta del lenguaje. El ejemplo más ilustrativo es la clase Functional:



Llamada

Se habla de llamadas cuando se detecta un nombre de función o de variable junto a una serialización o elemento aislado encapsulado entre paréntesis. Se considera que la llamada es una función si el primer elemento es un nombre de función definido y se considera una operación de cálculo en cualquier otro caso. En el caso de que exista una variable definida con el nombre de una función, la llamada a función tendrá prioridad mayor.

Operación

Se habla de una operación cuando en el árbol de sintaxis abstracta se detecta una sucesión de operadores con expresiones cualesquiera intermedias. Puede ser que se considere que una expresión es una secuencia cuando se utilizan ciertas palabras reservadas, pese a seguir la definición anterior.

Se define el caso especial de la operación de índice, que consiste en un nombre de variable seguido de un elemento aislado encapsulado entre corchetes.

Secuencia

Definir este elemento sintáctico es complicado, ya que podría decirse que es una serie de palabras reservadas con expresiones intercaladas, pero eso podría chocar con la definición de operación. Distinguir una operación de secuencias de una secuencia de operaciones puede ser una tarea muy compleja, por lo que se establecen las siguientes normas para la identificación de expresiones:

- Ante un caso ambiguo, se considerará que la expresión es una secuencia de operaciones. Para hacer el caso análogo se deberán encapsular las secuencias que hacen de operandos entre paréntesis. Esto evita unos altos niveles de ambigüedad, pese a la inconveniencia.
- Cualquier expresión que tenga una palabra reservada que no sea detectada como parte de otro elemento semántico (variables tipadas, clases indirectas…) será sometida a un algoritmo de reconocimiento de patrones para buscar una secuencia que encaje con ella. Si no se encuentra ninguna secuencia que coincida, se considerará un fallo semántico.

Sintaxis propia

Al igual que las clases directas, se define un elemento semántico de este tipo como cualquier elemento sintáctico con el mismo nombre, por lo que es fácil entender la transición entre representaciones.

La diferencia principal entre una clase directa y una indirecta es que se sobreentiende que una sintaxis propia es una **función con sintaxis especial**, por lo que es una expresión con retorno de un tipo no especificado y no se trata de una clase (no es lo mismo un Number que una función que retorna un Number). Esto tiene como principal consecuencia que el momento en el que se ejecuta cada uno cambia.

Sintaxis de valores indirectos

Tal y como se mencionó anteriormente, existen una serie de expresiones en Ulan que se agrupan dentro de una categoría llamada *valores indirectos*. Estas expresiones devuelven un valor de una clase especificada utilizando una sintaxis especial y propia de la clase. A continuación se describe la sintaxis de cada valor indirecto en Ulan.

Containers

Los Containers en Ulan se definen utilizando código encapsulado en llaves, siguiendo el estilo de lenguajes como *Python*. La diferencia principal es que esta sintaxis debe ser expandida para permitir definiciones de numerosos subtipos de Container, por lo que no sirve la estrategia de *Python* de cambiar el encapsulador. La sintaxis final es la siguiente:

$$[Comparator]\{Elementos\ \ldots\} : Tipo$$

Siendo $Comparator$ un criterio complejo de ordenación de elementos encapsulado en corchetes, $Elementos$ las diferentes subexpresiones que componen al objeto final y $Tipo$ el subtipo al que pertenece. El primero es completamente opcional, así como el tipo, que será considerado un array dinámico por defecto. Si se introduce un criterio de comparación el tipo debe estar ordenado.

Mapper

Los objetos de este tipo tienen una definición similar a la de los containers, pero con la diferencia de que todos los elementos de su interior son operaciones de definición con tan solo dos operandos. Tan solo se detectará como definición de Mapper en el caso de que el árbol de tokens resultante tenga exactamente tres nodos en su nivel superior. Cabe destacar que unas llaves vacías sin más indicación serán detectadas como un Mapper vacío. La sintaxis es la siguiente:

$$\{Key1 : Value1, Key2 : Value2\ \ldots\}$$

Functional

Las expresiones funcionales en Ulan tienen una sintaxis minimalista con la intención de que sea completamente natural utilizarlas, pero introduciendo sintaxis que debería estar orientada a un nivel más bajo de lo habitual. La sintaxis es la siguiente:

$$[Captura](Params) : Expr$$

Siendo $Captura$ una serialización encapsulada en corchetes con los nombres de las variables que se desean capturar como datos estáticos en la ejecución, $Params$ los parámetros de la expresión funcional y $Expr$ la expresión que se ejecutará. Cabe destacar que la captura es completamente opcional, pero puede dejarse como unos corchetes vacíos por razones estéticas si se desea. Asimismo, los parámetros pueden ser paréntesis vacíos en el caso de que la función no tenga parámetros.

La expresión que sigue al operador de definición no puede ser una cabecera, pero puede ser una expresión que no retorne ningún valor.

Unidades de código

Estructura de un programa en Ulan

Cuando hablamos de un programa en cualquier lenguaje de programación debemos ser concretos, ya que no es correcto decir que se pueden escribir líneas de código en un archivo de texto plano y eso generará un programa válido. La estructura de código principal en Ulan es el módulo, el cual consiste en un fragmento de código que se ejecuta teniendo en cuenta ciertos aspectos de configuración individuales.

Tanto las librerías como los programas completos se estructuran en módulos y todos siguen las mismas restricciones de formato. Se debe tener especial cuidado a la hora de definir una librería por la naturaleza cambiante del lenguaje base.

Estructura de un módulo

Un módulo se compone de dos partes: las cabeceras y el código. Estas dos partes deben ir en orden y el orden puede influir en la ejecución. Los parámetros de configuración mencionados anteriormente son los que van en las cabeceras de cada módulo y son esenciales para el correcto funcionamiento.

No es totalmente incorrecto ver a las cabeceras como las directivas de preprocesador en lenguajes como C y C++, pero con la diferencia de que en Ulan las cabeceras tienen un orden establecido. A continuación se describe cada una de ellas en el orden que deben introducirse.

Cabeceras

Identificador de módulo

Se llama identificador de módulo al nombre con el que se referencia desde módulos externos. Solo tiene sentido introducirlo en el caso de que se esté desarrollando una librería para el lenguaje que debe ser importada en programas posteriores, por lo que es completamente opcional. A continuación se especifica la sintaxis de esta cabecera en BNF:

```
<s> ::= {“ ”} //Cualquier número de espacios

<c> ::= (Cualquier carácter alfanumérico o barra baja)

<Id> ::= <s> “Module ” <s> <c> {<c>} <s>
```

La secuencia de caracteres alfanuméricos que siguen a “Module” serán interpretados como el identificador del módulo. Este identificador debe ser único entre los archivos de todas las carpetas de búsqueda de módulos.

Dependencias

Se denominan dependencias a todos los módulos que deben ser ejecutados previamente para garantizar el correcto funcionamiento de un módulo en particular. Estas son especificadas mediante cabeceras *Import*, las cuales tienen la siguiente sintaxis:

```
<s> ::= {“ ”} //Cualquier número de espacios

<c> ::= (Cualquier carácter alfanumérico o barra baja)

<Id> ::= <s> “Import ” <s> “<” <c> {<c>} “>” <s>
```

La secuencia de caracteres alfanuméricos encapsulados en los caracteres “<” y “>” serán el identificador del módulo que se debe importar antes de ejecutar el módulo actual. Se calculará el orden topológico de las dependencias de un programa antes de ejecutarlo para evitar incompatibilidades.

Si un Módulo no tiene cabeceras *Import* se considerará que no tiene dependencias y se ejecutará directamente.

Sintaxis admitidas

Dada la naturaleza cambiante de la sintaxis de Ulan, cabe la posibilidad de que una sintaxis definida en alguna dependencia cause algún conflicto con el programa ya escrito. Para evitar esto se utilizan las cabeceras *Using*, las cuales tienen el siguiente formato:

```
<s> ::= {“ ”} //Cualquier número de espacios

<c> ::= (Cualquier carácter alfanumérico o barra baja)

<Id> ::= <s> “Using ” <s> <c> {<c>} <s> “ syntax”
```

La palabra que se encuentra entre Using y syntax tiene varias posibilidades y es la que le da el significado a la cabecera:

- **“every”:** se utilizarán todas las sintaxis de clases definidas en las dependencias, así como las de las clases básicas.
- **“no”:** no se utilizará **ninguna** sintaxis. Esto incluye las de las clases básicas.
- **“standard”:** se utilizarán las sintaxis de las clases básicas.
- **“defined”:** las sintaxis definidas en el módulo actual tomarán efecto al definirse. Este no es el comportamiento por defecto.
- **Nombre de clase:** se activará la sintaxis de una clase en particular.

Cabe destacar que, con la excepción de *Using no syntax*, se pueden encadenar estas cabeceras para hacer comportamientos más complejos. Si no se incluye ninguna cabecera *Using* se considerará que se quiere la sintaxis estándar. Asimismo, también es necesario mencionar que estas cabeceras no afectan a las funciones con sintaxis, sino a las sintaxis directas de las clases Ulan del programa.

Clases

Introducción

Se puede considerar que Ulan es un lenguaje de **tipado gradual**, ya que implementa mecanismos para escribir código con tipos estáticos y dinámicos.

Asimismo, se implementan medios para definir **casts implícitos** que pueden ser expresados también de manera explícita. Estas conversiones se harán en resolución de operadores y en resolución de argumentos de funciones si el caso lo requiere, por lo que se puede afirmar que este lenguaje es de **tipado débil**.

También cabe destacar que se permite la definición de clases simples sin herencia y funciones miembro dentro de las mismas. A estas clases definidas por el usuario se las denomina clases propias.

Tipos básicos

Boolean

Definición	Clase que representa valores los lógicos Verdadero y Falso
Parsing	Una instancia de este tipo está definida como la cadena de caracteres “True” o “False” sin distinción de mayúsculas y minúsculas, asociando estas cadenas a los valores Verdadero y Falso, respectivamente
Notación BNF	<true> ::= (“T” | “t”) (“R” | “r”) (“U” | “u”) (“E” | “e”) <false> ::= (“F” | “f”) (“A” | “a”) (“L” | “l”) (“S” | “s”) (“E” | “e”) <boolean> ::= <true> | <false>
Ejemplos	True, False

String

Definición	Clase que representa una cadena de caracteres
Parsing	Una instancia de este tipo está definida como cualquier cadena que se encuentre entre dos caracteres “”. Los caracteres son válidos mientras tengan encoding UTF-8. Pese a tener una sintaxis fácilmente identificable, se considera una clase indirecta, ya que las comillas son encapsuladores
Notación BNF	-
Ejemplos	“¡Hola, mundo!”, “Texto de ejemplo”

Number

Definición	Clase que representa un número, sea este positivo, negativo, entero o decimal. Está implementada para soportar operaciones con precisión arbitraria
Parsing	Un entero está definido como una sucesión de dígitos, que son caracteres numéricos del 0 al 9 (ambos inclusive). Una instancia de tipo Number está definida como un número entero precedido por un "-" opcional y seguido de un "." y otro entero, siendo ambos opcionales en conjunto
Notación BNF	<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" <integer> ::= <digit> {<digit>} <number> ::= ["-"] <integer> ["." <integer>]
Ejemplos	9.3, -3, -9.921

Matrix

Definición	Clase que representa una matriz numérica
Parsing	Se define una fila de matriz como instancias de tipo Number separadas por comas (','), estando estos dentro de un bloque encapsulado por los caracteres '[' y ']'. Se define una instancia de tipo Matrix como filas de matriz separadas por comas y encapsuladas en los mismos caracteres que las filas. Una matriz con filas de diferente longitud se detecta como si fuera una matriz cualquiera, pero acarreará un error de formato
Notación BNF	<s> ::= {" "} //Cualquier número de espacios <matrixRow> ::= "[" <s> <number> <s> {", " <s> <number> <s>} "]" <matrix> ::= "["<s> <matrixRow> <s> {", " <s> <matrixRow> <s>} "]"
Ejemplos	[[1, 4.2, 5], [4, 5, -2.4]], [[1, 2], [3, 4]]

Symbolic

Definición	Librería de manejo de expresiones simbólicas con variables numéricas, simplificación y transformación de expresiones algebraicas y resolución y simplificación de lógica proposicional
Parsing	Este tipo no tiene parsing directo, se crean variables simbólicas con la función Sym(String) y se construyen operaciones complejas con operaciones usando las mismas. No existen restricciones de formato con respecto a los nombres de variables simbólicas
Notación BNF	-
Ejemplos	Sym("x"), Sym("y_1")

Container

Definición	Librería de manejo de estructuras de datos para almacenar objetos. Los tipos soportados son vectores (arrays dinámicos), deques (colas doblemente terminadas), listas (listas encadenadas) y sets (conjuntos); todos ellos con variantes ordenadas por orden natural y Comparator
Parsing	Una instancia de este tipo está definida como una serialización de elementos de cualquier clase válida encapsulada dentro de llaves. Por defecto se considerará que el Container es un vector no ordenado, pero se puede utilizar el operador de definición para especificar otro tipo de Container. Opcionalmente puede ir precedido de un objeto de tipo Comparator encapsulado en corchetes para generar una variante ordenada por ese criterio. No se considera que esta clase tenga parseo directo, sino que es una clase indirecta.
Notación BNF	-
Ejemplos	{1, true, [[1, 2], [3, 4]]}, {1, var, {1, 2}} : Deque, [comp]{1, 2, 3}

Mapper

Definición	Array asociativo implementado como una tabla de hashes
Parsing	Se define una instancia de tipo Mapper como un Container en el que todas las expresiones encapsuladas son operaciones binarias de definición (arg1: arg2). Todas estas asignaciones serán codificadas en el array asociativo siguiendo el formato "Key: Value". Este tipo de definición es equivalente a la utilizada en lenguajes como *Python*
Notación BNF	-
Ejemplos	{3 : true, 6 : False, {1, 2} : "Prueba"}, {(2 + var) : 7, True : (var2 + 5)}

Image

Definición	Estructura de datos que almacena un array bidimensional de píxeles. Cada uno de estos píxeles almacena tres valores que representan las coordenadas RGB del color (24 bits). Se permitirá guardar esta imagen en formato bmp sin compresión
Parsing	Una instancia de esta clase debe ser creada utilizando la función canvas, la cual devolverá una imagen de un tamaño especificado con un color de fondo
Notación BNF	-
Ejemplos	Canvas(100, 100, "White"), canvas(500, 500, {150, 50, 75})

Functional

Definición	Lambda-expresión (función sin nombre que puede ser almacenada como cualquier otro objeto con una estructura particular). Se pueden especificar argumentos en serie y el tipo de los mismos. Por defecto los argumentos se consideran de tipo Any
Parsing	Este tipo no tiene parsing directo, se usa el operador de dependencia funcional junto a unos parámetros encapsulados en paréntesis y una captura (datos estáticos) encapsulados en una tupla de alto nivel
Notación BNF	-
Ejemplos	(x, y) : x + y, [n](x : Number…) : sum({i for i in n, i for i in x})

Comparator

Definición	Criterio de comparación de objetos. Se permiten criterios compuestos
Parsing	Este tipo no tiene parsing directo, se usa la función comparing para inicializar un objeto de este tipo
Notación BNF	-
Ejemplos	Comparing((x : Container) : x.size(), True)

Etiquetas especiales

None	Esta etiqueta hace referencia a un objeto vacío. Tan solo se usa de manera interna para identificar objetos sin inicializar.
Any	Esta etiqueta hace referencia a cualquier clase. Sirve para implementar funciones con argumentos o retornos no especificados
Void	Esta etiqueta sirve para especificar una función sin retorno (equivalente a void en C, C++, Java…)

Características compartidas

Todas las clases básicas tienen funciones internas que les permiten saber el tamaño exacto que ocupan en memoria en bytes, así como una función de hash, pero esta no existe para todos los tipos (Comparator y Functional no son hasheables y al intentar calcular el hash de un objeto de estos tipos ocurrirá un error de ejecución).

Cualquier clase definida por el usuario será hasheable y la función de hash se definirá automáticamente siempre y cuando todos sus atributos sean de tipos hasheables. Asimismo, la función que calcula el tamaño en memoria también se definirá automáticamente.

Cabe destacar que se definirá automáticamente una **función constructora** cuando se defina una clase. Esta función tendrá como parámetros los atributos de la clase y devolverá una instancia de la misma con los valores asignados. Siguiendo unas normas similares, también se definirán *getters* para cada atributo al definir una clase.

Referencias

Cualquier objeto puede ser referenciado por un objeto de tipo **referencia** (representadas por el operador &), evitando así hacer copias innecesarias de datos de gran volumen. Todas las variables son referenciadas por defecto al ser recuperadas. Estas referencias son el equivalente semántico a las referencias en C++.

Es posible especificar parámetros en funciones que sean obligatoriamente referencias, pero un parámetro que pida un valor también podrá ser satisfecho con una referencia del mismo tipo. Precisamente gracias a esto se pueden hacer las llamadas *asignaciones a referencia*, las cuales modifican el valor almacenado en una dirección de memoria en particular, permitiendo modificar objetos sin reasignarlos por completo.

Las referencias de referencias no están permitidas, si se intenta hacer una asignación con ese fin, la variable del lado izquierdo tomará el rol de referencia al valor que referencia la variable del lado derecho. Dada la siguiente secuencia de código:

```
a = 2^1000000 //Datos de gran volumen
b = a
c = b
```

Al terminar de ejecutarse tanto la variable b como la variable c estarán referenciando al mismo valor, al de la variable a. Esto es conveniente porque el volumen del objeto almacenado en a es considerable (aproximadamente 120kB). Asimismo, se puede especificar que se busca una copia de los datos almacenados en una variable con el operador de desreferencia.

También existe la posibilidad de obtener **referencias constantes** (representadas con el operador &&), que son el equivalente semántico a las referencias constantes en C++. Tienen exactamente los mismos usos que las referencias normales, pero no se puede modificar el valor al que apuntan. Estas referencias son devueltas automáticamente en situaciones en las que no se quieren copiar datos, pero la modificación de una referencia conllevaría comportamiento indefinido por la estructura en la que se encuentra el dato (Ejemplo: modificar una posición en un vector ordenado). Una referencia común puede enlazársele a una referencia constante si es necesario.

Para evitar modificaciones de datos no deseadas, no se permite que se apliquen funciones miembro no constantes sobre objetos que tengan almacenados una referencia constante. Esto permite utilizar estructuras tales como Containers ordenados de una manera mucho más segura.

Direccionamiento y enlaces

Tal y como se especificó en el apartado anterior, es posible que se le asigne un valor a una variable o argumento aunque sus tipos no coincidan del todo. Esto se debe a que el tipo de un objeto se divide en *clase* y *direccionamiento*, siendo lo primero la clase Ulan a la que pertenece el objeto y lo segundo la manera de referenciarse que tiene.

Mientras que la clase de un tipo puede ser cualquier clase definida anteriormente, el direccionamiento tan solo puede ser un *valor directo*, una *referencia* o una *referencia constante*. Estos siguen la siguiente jerarquía:

$$Referencia \rightarrow Referencia\ constante \rightarrow Valor\ directo$$

Cualquier objeto que coincida en clase con el argumento o variable a la que se vaya a asignar y tenga un direccionamiento con una altura igual o superior en la jerarquía podrá ser asignado de manera normal. Si se hace un cast implícito se considerará que el objeto es un valor directo, ya que no cambia el sitio donde está almacenado.

Definición

Se permite la definición de estructuras nuevas en Ulan, definiéndose estas como clases. Asimismo, se permite definir un parsing directo para estas clases utilizando el lenguaje USDL.

Defínción de estructura

Como todas las estructuras internas en Ulan, las clases pueden ser definidas por el usuario con la palabra reservada “Define”. En el caso de las clases la definición tiene la siguiente sintaxis:

$$Define\ class\ "(Nombre)"\ as$$

$$(Bloque\ indentado)$$

Siendo $Nombre$ la cadena con la representación de la clase y el bloque indentado un cuerpo de clase. El nombre de una clase debe seguir las restricciones de formato de nombres de elementos contextuales.

Defínción de parsing directo

Se pueden definir sintaxis para clases propias. Esto se puede hacer utilizando de nuevo la palabra reservada “Define” con la siguiente sintaxis:

$$Define\ syntax\ for\ (Clase)\ as\ "(Pattern)"$$

$$(Bloque\ indentado)$$

Siendo $Clase$ el nombre de la clase para la que queremos definir una sintaxis y $Pattern$ un String que contenga una expresión en USDL (lenguaje de definición de sintaxis de Ulan), seguido del cuerpo de la función con los argumentos extraídos de $Formato$. Esta función debe devolver un objeto de la clase a la que define o acarreará un error de tipo. Los argumentos extraídos estarán en una variable de tipo Mapper llamada *args*.

Funciones constructoras y acceso a atributos

A la hora de definir una clase propia en Ulan es necesario tener claro cómo se puede acceder a los datos que encapsula o cómo se puede instanciar. A continuación se especifican las normas exactas de definición de funciones constructoras y *getters*.

Una función constructora es un concepto equivalente a un *constructor* en lenguajes como Java y C++. Se trata de una función que devuelve una instancia de una clase dados unos parámetros básicos y que constituye la manera estándar de instanciar un objeto. En Ulan las funciones constructoras se instancian automáticamente. Asimismo, los *getters* son funciones que permiten acceder a los atributos de una instancia de una clase. Las reglas que sigue el intérprete para declarar estas funciones son las siguientes:

- **Función constructora:** la función tendrá el nombre de la clase entre dos caracteres '_' (barra baja). Los parámetros de la función serán los atributos definidos para la clase ordenados de arriba abajo.
- **Getters:** estos serán definidos dependiendo del tipo exacto de cada atributo. Si el argumento es un valor directo o una referencia se definirá una función miembro para la clase con el mismo nombre que el atributo que devolverá una referencia al objeto, así como una versión que devuelve una referencia constante con un nombre precedido por el carácter 'c' (c minúscula). Si el atributo es una referencia constante tan solo se definirá este último.

Ejemplo:

```
Define class "State" as //Clase estado
   posx : Number //Posición
   posy : Number

   board : Matrix&& //Tablero (indica las paredes y el tamaño)
```

Las funciones definidas para esta clase serían las siguientes:

- **Función constructora:** _State_(posx : Number, posy : Number, board : Matrix&&)
- **Getters de posx:**
  - Matrix::posx() → Referencia a posx
  - Matrix::cposx() → Referencia constante a posx
- **Getters de posy:**
  - Matrix::posy() → Referencia a posy
  - Matrix::cposy() → Referencia constante a posy
- **Getter de board:**
  - Matrix::cboard() → Referencia constante a board

Casts

Introducción

Los casts (conversiones de tipos) en Ulan pueden ser expresados de manera explícita o implícita, teniendo en ambos el mismo efecto. Asimismo, cada estructura tiene sus propias reglas a la hora de aplicar casts de manera implícita (ver uso en operaciones, uso en funciones).

Casts básicos

En la versión estándar del intérprete de Ulan vienen definidos varios casts por defecto:

Tipo de origen	Tipo de destino	Descripción
Boolean	Number	Se le asigna el valor 0 a False y el valor 1 a True
	String	Conversión a String
Number, Matrix, Symbolic, Container y Mapper	String	Conversión a String

Uso explícito

Pese a que Ulan está diseñado para soportar conversiones implícitas, cualquier cast puede ser utilizado de manera explícita como una función de la siguiente forma:

$$Tipo1(Objeto\ de\ tipo\ 2)$$

$$\boldsymbol{Ejemplo}: String(-6.4)$$

Definición

Como todas las estructuras internas en Ulan, los casts pueden ser definidos por el usuario con la palabra reservada "Define". La sintaxis exacta a utilizar es la siguiente:

$$Define\ cast\ Tipo2(var : Tipo1)\ as$$

$$(Bloque\ indentado)$$

Siendo el bloque indentado el cuerpo de la función que transforma el argumento con clase $Tipo1$ en un objeto de clase $Tipo2$. El intérprete se encargará de comprobar los tipos de los argumentos y las redefiniciones en tiempo de interpretación.

Operadores

Introducción

Los operadores en Ulan se definen necesariamente usando tres propiedades concretadas en la siguiente tabla (excepto casos especiales):

Posición	**Infijo**	**Término asociado: "infix"** Se define un operador infijo como un operador binario: $\boldsymbol{Arg1}\ [\boldsymbol{op}]\ \boldsymbol{Arg2}$
	Prefijo	**Término asociado: "prefix"** Se define un operador prefijo como un operador unario que se coloca antes del argumento al que quiere aplicarse: $[op]Arg$
	Sufijo	**Término asociado: "suffix"** Se define un operador sufijo como un operador unario que se coloca después del argumento al que quiere aplicarse: $Arg[op]$
Agrupamiento (operadores binarios)	**Izquierda**	**Término asociado: "left"** Operadores que se operan de izquierda a derecha: $(a\ [op]\ b)\ [op]\ c\ =\ a\ [op]\ b\ [op]\ c$
	Derecha	**Término asociado: "right"** Operadores que se operan de derecha a izquierda: $a\ [op]\ (b\ [op]c)\ =\ a\ [op]\ b\ [op]\ c$
	Totalmente conmutativo	**Término asociado: "full"** Operadores tales que: $(a\ [op]b)\ [op]\ (c\ [op]\ d) =\ a\ [op]\ b\ [op]\ c\ [op]\ d$
Precedencia	Entero que indica la prioridad a la hora de agrupar operadores diferentes. Dados dos operadores $[op1]$ y $[op2]$, siendo la precedencia de $[op2]$ mayor que la de $[op1]$: $Arg1\ [op1]\ Arg2\ [op2]\ Arg3 = Arg1\ [op1]\ (Arg2\ [op2]\ Arg3)$	

Operadores básicos

En la versión básica del intérprete de Ulan vienen definidos varios operadores, separados en varias categorías según su ámbito de uso.

Operadores aritméticos

Estos operadores son los utilizados en el ámbito matemático del lenguaje. Tienen un funcionamiento estándar y vienen con las sobrecargas necesarias para utilizarlos en cualquier entorno, sea este básico o avanzado:

Representación	Agrupamiento	Posición	Precedencia	Descripción
+	full	infix	400	Suma
-	left	infix	450	Resta
*	full	infix	500	Multiplicación
/	left	infix	550	División
^	right	infix	600	Potencia
%	left	infix	350	Módulo
!	-	suffix	-	Factorial
-	-	prefix	-	Negación

Operadores lógicos

Estos operadores son los utilizados en cálculos lógicos para condiciones y en modelado de expresiones en lógica proposicional:

Representación	Agrupamiento	Posición	Precedencia	Descripción
&&	full	infix	100	AND lógica
||	full	infix	150	OR lógica
->	left	infix	200	Implicación
<->	left	infix	250	Bicondicional
!	-	prefix	-	Negación lógica

Operadores de comparación

Estos operadores son los utilizados para comparar objetos dependiendo de su valor. Su uso en serie no es tan común como el de los operadores aritméticos, ya que no tienen la semántica esperada. La lista es la siguiente:

Representación	Agrupamiento	Posición	Precedencia	Descripción
==	full	infix	300	Igualdad
!=	full	infix	300	No igualdad
<	left	infix	325	Menor a
<=	left	infix	325	Menor o igual a
>	left	infix	325	Mayor a
>=	left	infix	325	Mayor o igual a

Operadores de asignación común

Estos operadores se utilizan para asignar valores a variables o para cambiar valores de variables ya existentes en el código. Estos operadores no siguen las mismas normas de agrupamiento que los aritméticos y los lógicos, sino que tienen normas especiales, ya que funcionan a un nivel superior de abstracción. A continuación se listan estos operadores:

Representación	Agrupamiento	Posición	Precedencia	Descripción
=	-	infix	0	Asignación de valores a variables.
+=	-	infix	0	Equivalente a: $arg1 = arg1 + arg2$
-=	-	infix	0	Equivalente a: $arg1 = arg1 - arg2$
*=	-	infix	0	Equivalente a: $arg1 = arg1 * arg2$
/=	-	infix	0	Equivalente a: $arg1 = arg1/arg2$
%=	-	infix	0	Equivalente a: $arg1 = arg1\%arg2$
^=	-	infix	0	Equivalente a: $arg1 = arg1\hat{}arg2$

El uso básico de estos operadores es $arg1\ [op]\ arg2$, siendo $arg1$ una variable (independientemente del tipado), y $arg2$ el valor a asignar. Precisamente por el carácter especial del primer argumento se dice que las asignaciones funcionan a un nivel superior, pero no se prohíbe su sobrecarga para su uso con otros elementos semánticos. A continuación se detallan los usos especiales de estos operadores:

- **Asignación múltiple**: el uso serializado de este operador ($Arg1 = Arg2 = Arg3$ …) tiene como consecuencia que a todos los argumentos, excepto al último, se les asigne el valor del último argumento, por lo que deben ser variables.
- **Asignación Paralela**: utilizar este operador con dos argumentos que sean tuplas de alto nivel del mismo tamaño ($[Arg1, Arg2, \dots] = [Val1, Val2, \dots]$) tiene como consecuencia que se copien los estados de las variables de la segunda tupla y se asignen a sus respectivas variables de la primera tupla, por lo que todos sus argumentos deben ser variables, estén definidas o no.

Operador de asignación a referencia

Existe un operador de asignación especial que tiene un funcionamiento diferente al de los operadores anteriormente mencionados. Este operador es el utilizado para modificar elementos individuales de Containers, atributos de clases propias y cadenas de referencias en variables.

Este operador tiene como representación ":=" y tiene una semántica bien definida: cuando el operando de la izquierda es una referencia **no constante**, se sustituye el valor al que apunta por el del operando derecho sin invalidar la referencia. Esta operación es la manera estándar de modificar estructuras sin necesidad de establecer un contexto más amplio para las asignaciones comunes.

Operadores especiales

Estos operadores tienen usos concretos dentro de las estructuras del propio lenguaje. También pueden poseer estructuras que no encajen con los estándares definidos anteriormente:

Representación	Agrupamiento	Posición	Precedencia	Descripción
:	right	infix	50	Definición/Dependencia funcional
::	right	infix	50	Pertenencia
[]	-	-	-	Operador de índice. $Arg1[Arg2]$
()	-	-	-	Operador de cálculo. $Arg1(Arg2, Arg3 \dots)$
*	-	prefix	-	Desreferencia
...	-	suffix	-	Expansión

Muchos de estos operadores se utilizan a la hora de definir comportamientos de bajo nivel o al describir estructuras de datos en el lenguaje y podrían confundir al intérprete, por lo que tienen la sobrecarga desactivada. Asimismo, también se incluyen los operadores de índice y de cálculo que, a pesar de poseer una estructura similar entre ellos, se separan del resto por no considerarse lo suficientemente numerosos como para crear una categoría propia.

El uso concreto del resto de operadores se especifica en otras partes del documento más relacionadas con su contexto:

- **Definición / Dependencia funcional**: Functional, Mapper, Argumentos de funciones.
- **Pertenencia**: Definición de funciones miembro.
- **Desreferencia**: Referencias.
- **Expansión**: Argumentos de funciones.

Operaciones básicas

Igual que en el caso de los operadores, la versión estándar del intérprete de Ulan viene con varias operaciones (sobrecargas de los operadores existentes) definidas por defecto. A continuación se presentan varias tablas divididas por categorías de operadores que especifican las sobrecargas incluidas.

Operaciones aritméticas

Las operaciones aritméticas incluidas por defecto quedan especificadas y descritas en la siguiente tabla:

Operador	Operandos		Descripción
+	Number	Number	Suma de números
	Matrix	Matrix	Suma de matrices
	Symbolic	Symbolic	Suma de expresiones simbólicas
	Number	Symbolic	Suma de constante y expresión simbólica (izquierda)
	Symbolic	Number	Suma de constante y expresión simbólica (derecha)
	String	String	Concatenación de Strings
- (infijo)	Number	Number	Resta de números
	Matrix	Matrix	Resta de matrices
	Symbolic	Symbolic	Resta de expresiones simbólicas
	Number	Symbolic	Resta de expresión simbólica a constante
	Symbolic	Number	Resta de constante a expresión simbólica
*	Number	Number	Producto de números
	Matrix	Matrix	Producto de matrices
	Number	Matrix	Escalado de matrices (izquierda)
	Matrix	Number	Escalado de matrices (derecha)
	Symbolic	Symbolic	Producto de expresiones simbólicas
	Number	Symbolic	Producto de constante y expresión simbólica (izquierda)
	Symbolic	Number	Producto de constante y expresión simbólica (derecha)
/	Number	Number	División de números
	Matrix	Number	Escalado de una matriz
	Symbolic	Symbolic	División de expresiones simbólicas
	Number	Symbolic	División de expresión simbólica y constante
	Symbolic	Number	División de constante y expresión simbólica
^	Number	Number	Potencia de números
	Matrix	Number	Potencia de una matriz
%	Number	Number	Módulo de un número
	Symbolic	Symbolic	Módulo de expresiones simbólicas
	Number	Symbolic	Constante módulo una expresión simbólica
	Symbolic	Number	Expresión simbólica módulo una constante

Operador	Operandos	Descripción
! (sufijo)	Number	Factorial de un número (admite decimales)
	Symbolic	Factorial de una expresión simbólica (admite decimales)
- (prefijo)	Number	Negación aritmética de un número
	Symbolic	Negación aritmética de una expresión simbólica

Operaciones lógicas

Las operaciones lógicas incluidas por defecto quedan especificadas y descritas en la siguiente tabla:

Operador	Operandos		Descripción
&&	Boolean	Boolean	AND de dos valores lógicos
	Symbolic	Symbolic	AND de dos expresiones simbólicas
||	Boolean	Boolean	OR de dos valores lógicos
	Symbolic	Symbolic	OR de dos expresiones simbólicas
->	Boolean	Boolean	Implicación lógica de dos valores lógicos (!arg1 || arg2)
	Symbolic	Symbolic	Implicación lógica de dos expresiones simbólicas
<->	Boolean	Boolean	Bicondicional lógico de dos valores lógicos (arg1 -> arg2 && arg2 -> arg1)
	Symbolic	Symbolic	Bicondicional lógico de dos expresiones simbólicas
! (prefijo)	Boolean		Negación de un valor lógico
	Symbolic		Negación de una expresión simbólica

Operaciones de comparación

Las operaciones lógicas incluidas por defecto quedan especificadas y descritas en la siguiente tabla:

Operador	Operandos		Descripción
==	Dos clases iguales cualesquiera con función de hash		Igualdad de dos valores
!=	Dos clases iguales cualesquiera con función de hash		No igualdad de dos valores
<	Number	Number	Comparación numérica
	String	String	Comparación lexicográfica
<=	Number	Number	Comparación numérica
	String	String	Comparación lexicográfica
>	Number	Number	Comparación numérica
	String	String	Comparación lexicográfica
>=	Number	Number	Comparación numérica
	String	String	Comparación lexicográfica

Operaciones especiales

La mayoría de los operadores especiales tienen usos internos en el lenguaje, por lo que su uso concreto está especificado en la sección relativa a ese contexto (ver operadores especiales). A continuación se detallan las sobrecargas incluidas por defecto de los operadores especiales que no son de uso interno.

Operador	Operandos		Descripción
[]	Container	Number	Devuelve una referencia al valor dentro del primer argumento en la posición que indique el segundo argumento de la operación si el subtipo de Container lo permite.
	Mapper	tipo cualquiera con función de hash	Devuelve una referencia al valor dentro del primer argumento en la posición que indique el segundo argumento. Si no existe lanzará una excepción.
()	Symbolic	[Mapper]	Devuelve el primer argumento con las sustituciones especificadas en el segundo argumento. Este último debe tener claves de tipo Symbolic y valores de tipo Symbolic, Number o Boolean dependiendo del tipo de sustitución. Se realizan cálculos numéricos por defecto.
	Symbolic	[Mapper, Boolean]	Se realiza una sustitución tal y como la de la operación anterior, pero hay un flag que permite elegir si se realizan operaciones numéricas o si se dejan indicadas.

La sobrecarga de estos operadores es ligeramente diferente a la de los demás operadores, se dan más detalles al respecto en el apartado de definición de operadores.

Casts implícitos en operaciones

Como ya se mencionó en el sistema de tipos, Ulan hará conversiones implícitas de tipos en operaciones si puede solucionar un error de cálculo. Esto se hará en todos los operadores aritméticos, lógicos y de comparación. También se harán en las operaciones especiales no relacionadas con el funcionamiento interno del lenguaje pero, al tener una estructura diferente, también seguirán criterios diferentes

Dada una operación cualquiera $Arg1\ [op]\ Arg2$, se intentará hacer un cast implícito únicamente en el caso de que la operación no está definida para los tipos de $Arg1$ y $Arg2$, en cuyo caso se buscarán todas las posibles operaciones realizables haciendo un cast sobre **uno de los operandos**. A partir de aquí puede que se encuentre una única posibilidad o puede que se encuentren varias. Tan solo se hará la operación si se encuentra una única posibilidad, ya que el segundo caso acarrearía un error por ambigüedad.

Un ejemplo podría ser la siguiente línea de código, suponiendo el estado de fábrica del intérprete y los casts por defecto:

$$var = 3.5 + True$$

Viendo la lista de operaciones aritméticas, se puede observar que la operación $Number + Boolean$ no está definida, pero al observar la lista de casts, vemos que es posible convertir el segundo operando a Number, mientras que el único cast posible desde Number es a String y no soluciona nada. La solución, por lo tanto, será hacer una conversión implícita de Boolean a Number al segundo operando:

$$var = 3.5 + True$$

$$\Downarrow$$

$$var = 3.5 + 1$$

Y así queda resuelto el problema de la operación no definida, aunque no siempre es posible aplicar esta medida. Cabe destacar que para ejecutar esta corrección hace falta hacer una búsqueda de todos los casos posibles dadas las conversiones definidas, por lo que se puede deducir que los casts implícitos son ligeramente más lentos en el intérprete que los explícitos.

Es necesario destacar que en el operador de cálculo las operaciones son ligeramente diferentes, ya que no se harán conversiones implícitas al primer operando. Los demás operandos serán convertidos siguiendo las normas de conversiones en argumentos de funciones.

Definición

Como la mayoría de las estructuras internas en Ulan, los operadores pueden ser definidos por el usuario con la palabra reservada *Define*. En el caso de los operadores primero hay que definir el operador y después las operaciones. La sintaxis exacta es:

$$Define\ [full \mid left \mid right]\ (infix \mid prefix \mid suffix)\ operator\ "(Nombre)"\ [prec\ (Number)]$$

Siendo $Nombre$ la representación del operador como cadena y el Number la precedencia. El objeto de tipo String que define la representación debe seguir las normas especificadas en el apartado de formato de un operador.

Los operadores de tipo *infix* son los únicos que deben pertenecer a la categoría *left*, *right* o *full*, ya que no tiene sentido hablar de agrupamiento en operadores unarios. Asimismo, los operadores unarios no tienen precedencia.

Tras definir un operador se pueden definir las operaciones asociadas al mismo utilizando una sintaxis especial. La secuencia exacta a utilizar es:

$$Define\ operation\ (arg1 : Tipo1)\ op\ (arg2 : Tipo2)\ as$$

$$(Bloque\ indentado)$$

Siendo la operación definida para el operador con la representación dada en el String para los tipos $Tipo1$ y $Tipo2$, siendo estos a su vez valores directos (no se permite la sobrecarga para operandos con otro direccionamiento). El bloque indentado es el cuerpo de la función que define a la operación. Esta definición es aplicable a operadores unarios utilizando la siguiente cabecera:

$$Define\ operation\ [op]\ (arg1 : Tipo1)\ [op]\ as$$

En este caso, es necesario tener en cuenta si el operador es prefijo o sufijo para colocarlo antes o después del argumento. El bloque indentado que hace de cuerpo de la función que define a la operación mantiene la semántica.

Cabe destacar el caso de los operadores de agrupamiento *full* (completamente conmutativos), ya que tienen una característica especial a la hora de definir operaciones asociadas. Cada vez que se defina una operación para operandos de dos clases diferentes se definirá la análoga con los tipos inversos de manera automática, preservando los nombres de los argumentos y el cuerpo de la función que modela a la operación. Esto permite tiempos de desarrollo más rápidos en librerías y un enfoque más matemático.

Los tipos de retorno de todas las operaciones definidas por el usuario serán inferidas del código del cuerpo de función que sigue a la cabecera. Si el algoritmo no puede obtener el tipo de retorno, se establecerá como Any.

Sobrecarga de operadores especiales

La sintaxis para la definición de operaciones anterior solo es válida para operadores con representación estándar. Esto hace que la definición de sobrecargas para los operadores de cálculo y de índice sean ligeramente diferentes. La sintaxis de la cabecera para el operador de índice es la siguiente:

$$Define\ operation\ (arg1 : Tipo1)[arg2 : Tipo2]\ as$$

Se considera que el operador de índice es un operador binario de la siguiente forma:

$$Arg1[Arg2]$$

Asimismo, la sintaxis para el operador de cálculo es bastante parecida a la del operador de índice. La sintaxis es la siguiente:

$$Define\ operation\ (arg1 : Tipo1)(arg2 : Tipo2, arg3 : Tipo3, \dots)\ as$$

Siguiendo la cabecera anterior, se considerará que la operación tiene la siguiente estructura:

$$Arg1(Arg2, Arg3, \dots)$$

Por supuesto, también existe el caso en el que tan solo existe $arg1$:

$$Arg1()$$

Funciones

Introducción

Así como en la mayoría de lenguajes de programación, Ulan puede agrupar código parametrizado y llamarlo funciones. Estas se definen con un nombre, una lista de argumentos y un tipo de retorno.

También existe la posibilidad de definir funciones miembro que pertenezcan a una clase. Estas serán llamadas desde instancias de la clase a la que pertenecen y podrán afectar al objeto en cuestión sin necesidad de referencias.

Parámetros

Los parámetros de una función se definen utilizando paréntesis que contienen variables (sean estas tipadas o no tipadas). Estos parámetros, al contrario que los tipos normales, pueden tener tipos especiales especificados con el operador de expansión (parámetros en serie), el cual indica que puede haber un número indeterminado de parámetros de ese tipo (se devolverán como un vector con el nombre del parámetro), habiendo ciertas restricciones con la elección de tipos.

Una lista de parámetros es válida si no se define un parámetro en serie de un tipo igual al del siguiente parámetro (los parámetros no tipados se consideran de tipo Any), ya que haría imposible de detectar al fin de la secuencia anterior. Asimismo tampoco se permiten parámetros no tipados tras un parámetro en serie.

Dada una lista de parámetros, se comprobará que unos argumentos de llamada encajan siguiendo las siguientes normas:

- Los argumentos normales (no en serie) se consideran válidos si su tipo coincide con el del parámetro de su posición o si el tipo del argumento tiene una conversión definida al mismo.
- Los argumentos en serie se consideran válidos si al menos el tipo de un argumento de llamada coincide con el tipo del parámetro. La secuencia de argumentos se interrumpirá si el tipo del siguiente parámetro coincide con el del argumento actual, por lo que solo se aplicarán conversiones si el argumento actual no rompe la secuencia.

Ejemplos:

$$(n : Number, bArgs : Boolean...)$$

$$(conts : Container\&\& ..., flag : Boolean, n : Number)$$

$$(arg1 ... , nArg : Number, c : Container\&)$$

Funciones básicas

La versión estándar del intérprete de Ulan viene con varias funciones definidas por defecto pertenecientes a diferentes ámbitos. A continuación se especifican estos ámbitos y las funciones pertenecientes a cada uno.

Funciones matemáticas básicas

Estas funciones son de cálculos matemáticos básicos y no tienen ningún efecto externo, tan solo devuelven el valor de la función especificada con la precisión que corresponda al número. Aunque no se especifique, se incluyen versiones simbólicas.

Nombre	Argumentos	Retorno	Descripción
abs	[Number]	Number	Valor absoluto
floor	[Number]	Number	Suelo
ceil	[Number]	Number	Techo
modPow	[Number, Number]	Number	Potencia modular
fact	[Number]	Number	Factorial
gamma	[Number]	Number	Función gamma
bin	[Number, Number]	Number	Coeficiente binomial
sqrt	[Number]	Number	Raíz cuadrada
nroot	[Number, Number]	Number	Raíz enésima
exp	[Number]	Number	Función exponencial
ln	[Number]	Number	Logaritmo natural
log	[Number, Number]	Number	Logaritmo genérico
plog	[Number]	Number	Función W de Lambert
sin	[Number]	Number	Seno
cos	[Number]	Number	Coseno
tan	[Number]	Number	Tangente
arcsin	[Number]	Number	Arcoseno
arccos	[Number]	Number	Arcocoseno
arctan	[Number]	Number	Arcotangente
sinh	[Number]	Number	Seno hiperbólico
cosh	[Number]	Number	Coseno hiperbólico
tanh	[Number]	Number	Tangente hiperbólica
arcsinh	[Number]	Number	Arcoseno hiperbólico
arccosh	[Number]	Number	Arcocoseno hiperbólico
arctanh	[Number]	Number	Arcotangente hiperbólica
sec	[Number]	Number	Secante
cot	[Number]	Number	Cotangente
csc	[Number]	Number	Cosecante
arcsec	[Number]	Number	Arcosecante
arccot	[Number]	Number	Arcocotangente
arccsc	[Number]	Number	Arcocosecante
sech	[Number]	Number	Secante hiperbólica
cot	[Number]	Number	Cotangente hiperbólica
csch	[Number]	Number	Cosecante hiperbólica
arcsech	[Number]	Number	Arcosecante hiperbólica
arccoth	[Number]	Number	Arcocotangente hiperbólica
arccsch	[Number]	Number	Arcocosecante hiperbólica
droot	[Number]	Number	Raíz digital

Funciones de álgebra lineal

Estas funciones sirven para hacer operaciones con matrices, así como con espacios vectoriales y sistemas de ecuaciones formalizados como tal.

Nombre	Argumentos	Retorno	Descripción
LUDecomp	[Matrix]	Container	Calcular la descomposición LU de la matriz dada
QRDecomp	[Matrix]	Container	Calcular la descomposición QR de la matriz dada
DEFDecomp	[Matrix]	Container	Calcular la descomposición DEF de la matriz dada
JacobiMatrix	[Matrix]	Matrix	Calcular la matriz de Jacobi de la matriz dada
GSMatrix	[Matrix]	Matrix	Calcular la matriz de Gauss-Seidel de la matriz dada
HHMatrix	[Matrix]	Matrix	Calcular la matriz de Householder de la matriz fila dada
nullMatrix	[i : Number, j : Number]	Matrix	Devolver la matriz nula de dimensiones *i* x *j*
identityMatrix	[order : Number]	Matrix	Devolver la matriz identidad de orden *order*
randomMatrix	[i : Number, j : Number, min : Number, max : Number, integer : Boolean]	Matrix	Devolver una matriz aleatoria de dimensiones *i* x *j* con números entre *min* y *max* opcionalmente enteros
eigenvalues	[Matrix]	Container	Devolver un Container con los autovalores de la matriz dada

Funciones de cálculo simbólico

Estas funciones se utilizan para manipular expresiones algebraicas y/o para obtener información sobre las mismas. No tienen efectos externos.

Nombre	Argumentos	Retorno	Descripción
sym	[name : String]	Symbolic	Generar variable simbólica con nombre *name*
pi	[]	Symbolic	Devolver constante simbólica *pi*
e	[]	Symbolic	Devolver constante simbólica *e*
(Funciones matemáticas básicas)	-	Symbolic	Devuelven una operación simbólica con la función básica especificada. No todas tienen una sobrecarga definida
solve	[eqs : Container]	Mapper	Resolver sistema de ecuaciones planteado

Funciones de lógica proposicional

Estas funciones sirven para implementar técnicas de resolución proposicional y de transformación de expresiones booleanas. No tienen efectos externos.

Nombre	Argumentos	Retorno	Descripción
sat	[Symbolic]	Boolean	Decidir si la expresión lógica dada es satisfactible
	[Container]	Boolean	Decidir si el conjunto de expresiones lógicas dado es consistente
tautology	[Symbolic]	Boolean	Decidir si la expresión lógica dada es una tautología
	[Container]	Boolean	Decidir si el conjunto de expresiones lógicas dado es una tautología
contradiction	[Symbolic]	Boolean	Decidir si la expresión lógica dada es una contradicción
	[Container]	Boolean	Decidir si el conjunto de expresiones lógicas dado es una contradicción
cnf	[Symbolic]	Symbolic	Calcular forma normal conjuntiva de la expresión lógica dada
	[Container]	Symbolic	Calcular forma normal conjuntiva del conjunto de expresiones lógicas dado
dnf	[Symbolic]	Symbolic	Calcular forma normal disyuntiva de la expresión lógica dada
	[Container]	Symbolic	Calcular forma normal disyuntiva del conjunto de expresiones lógicas dado
consequence	[expr1 : Symbolic, expr2 : Symbolic]	Boolean	Decidir si la expresión *expr2* es una consecuencia lógica de *expr1*
	[expr1 : Symbolic, exprs2 : Container]	Boolean	Decidir si el conjunto de expresiones lógicas *exprs2* es una consecuencia lógica de la expresión *expr1*
	[exprs1 : Container, expr2 : Symbolic]	Boolean	Decidir si la expresión lógica *expr2* es una consecuencia lógica del conjunto *exprs1*
	[exprs1 : Container, exprs2 : Container]	Boolean	Decidir si el conjunto de expresiones lógicas *exprs2* es una consecuencia lógica del conjunto *exprs1*

Operaciones numéricas

Estas funciones consisten en algoritmos generales aplicados a números:

Nombre	Argumentos	Retorno	Descripción
gcd	[Number, Number]	Number	Máximo común divisor
lcm	[Number, Number]	Number	Mínimo común múltiplo
factor	[Number]	Container	Descomposición en factores primos
modInv	[Number, Number]	Number	Inverso modular
pi	[n : Number]	Number	Constante pi con *n* decimales
e	[n : Number]	Number	Constante e con *n* decimales
sieve	[n : Number]	Container	Criba de números primos hasta *n*
rand	[min : Number, max : Number, integer : Boolean]	Number	Número aleatorio entre *min* y *max* opcionalmente entero

Entrada/salida

Estas funciones se utilizan para interactuar con el sistema sobre el que se ejecuta el intérprete de Ulan:

Nombre	Argumentos	Retorno	Descripción
print	[String]	Void	Mostrar por consola la cadena pasada como parámetro
println	[String]	Void	Mostrar por consola la cadena pasada como parámetro, seguida de un salto de línea
scan	[]	String	Pedir datos al usuario por consola
files	[String]	Container	Obtener la ruta completa de todos los archivos en la ruta pasada como parámetro
lines	[String]	Container	Devuelve un Container virtual que itera sobre las líneas del archivo especificado
characters	[String]	Container	Devuelve un Container virtual que itera sobre los caracteres del archivo especificado
write	[file : String, content : String append : Boolean]	Void	Escribir *content* en el archivo en *path*. Si *append* es True se escribe al final del archivo
writeln	[file : String, content : String]	Void	Escribir *content* al final del archivo en *path*, seguido de un salto de línea
system	[String]	Void	Usar comando del sistema

Excepciones

Estas funciones son utilizadas para comprobar restricciones e interrumpir el flujo normal del programa si es necesario:

Nombre	Argumentos	Retorno	Descripción
assert	[crit : Boolean, msg : String]	Void	Si *crit* es False se lanza una excepción con el mensaje especificado en *msg*
forbid	[crit : Boolean, msg : String]	Void	Si *crit* es True se lanza una excepción con el mensaje especificado en *msg*
throw	[msg : String]	Void	Lanzar una excepción con el mensaje especificado en *msg*

Secuencias

Nombre	Argumentos	Retorno	Descripción
range	[Number]	Container	Devolver Container virtual que itera desde cero hasta el número dado de uno en uno
	[Number, Number]	Container	Devolver Container virtual que itera sobre el rango dado de uno en uno
sequence	[start : Any, end : Any, next : Functional]	Container	Devuelve un Container virtual que itera desde *start* hasta *end* avanzando con la función *next*
	[start : Any, end : Any, next : Functional, map : Functional]	Container	Devuelve un Container virtual que itera desde *start* hasta *end* avanzando con la función *next*, pero el objeto devuelto en cada iteración es transformado con la función *map*
intension	[start : Any, endCond : Function, next : Functional]	Container	Devuelve un Container virtual que itera desde *start* hasta que se cumpla *endCond* avanzando con la función *next*
	[start : Any, endCond : Function, next : Functional, map : Functional]	Container	Devuelve un Container virtual que itera desde *start* hasta que se cumpla *endCond* avanzando con la función *next*, pero el objeto devuelto en cada iteración es transformado con la función *map*

Gráficos

Las siguientes funciones se utilizan para generar imágenes con diferentes propiedades:

Nombre	Argumentos	Retorno	Descripción
canvas	[h : Number, w : Number]	Image	Genera imagen de *h* x *w* píxeles con fondo blanco
	[Number, Number, String]	Image	Genera imagen de *h* x *w* píxeles con el color de fondo especificado (nombre del color en inglés)
	[Number, Number, Container]	Image	Genera imagen de *h* x *w* píxeles con el color de fondo especificado (coordenadas RGB)
rainbow	[Number]	Image	Generar gradiente del tamaño especificado de colores que pasa por todos los tonos
gradient	[Number, String, String]	Image	Generar gradiente del tamaño especificado de colores que pasa por los dos colores pasados como parámetro
	[Number, Container, String]	Image	
	[Number, String, Container]	Image	
	[Number, Container, Container]	Image	
plot	[f : Symbolic, var : Symbolic, xmin : Number, xmax : Number, ymin : Number, ymax : Number, args : Mapper]	Image	Generar gráfico de *f* con respecto a la variable *var* en el intervalo [*xmin*, *xmax*], siendo los valores representados los del intervalo [*ymin*, *ymax*]. El parámetro *args* es un mapper que puede contener las siguientes claves: • **height/width**: altura y anchura de la imagen generada • **lineColor/RGB**: color de la línea como string o coordenadas RGB, respectivamente • **backgroundColor/RGB**: color de fondo • **axisColor/RGB**: color del eje de coordenadas • **axis**: si es false no se dibujará eje de coordenadas • **extraFunctions**: mapper de Symbolic a String o Container con funciones extra a representar en el mismo gráfico con sus respectivos colores de línea

Nombre	Argumentos	Retorno	Descripción
colorPlot	[f : Symbolic, varX : Symbolic, varY : Symbolic, xmin : Number, xmax : Number, ymin : Number, ymax : Number, zmin : Number, zmax : Number, args : Mapper]	Image	Generar gráfico de *f* con respecto a la variable *varX* en el intervalo [*xmin*, *xmax*] y *varY* en el intervalo [*ymin*, *ymax*], siendo los valores representados los del intervalo [*zmin*, *zmax*] mapeados como colores en el gráfico. Para esto se genera un gradiente que pasa por tres colores en orden (color para negativos, color para cero y color para positivos). Dependiendo del valor que de la función en cada pixel se coloreará siguiendo ese gradiente o como nanColor en el caso de que se encuentre fuera del dominio. El parámetro *args* puede tener las siguientes claves: • **height/width:** altura y anchura de la imagen generada • **colors:** tamaño del gradiente a generar para cada rango • **negativeColor/RGB:** color de los números negativos • **zeroColor/RGB:** color del cero • **positiveColor/RGB:** color de los números positivos • **nanColor/RGB:** color de los números fuera del dominio • **axisColor/RGB:** color del eje de coordenadas • **axis:** si es false no se dibujará eje de coordenadas

Variables globales

Estas funciones controlan variables globales del lenguaje:

Nombre	Argumentos	Retorno	Descripción
setShownDigits	[Number]	Void	Establecer el número de dígitos enteros que se mostrarán de los números
setShownDecimals	[Number]	Void	Establecer el número de decimales que se mostrarán de los números
toggleSciRepresentation	[Boolean]	Void	Activar y desactivar la notación científica

Funciones misceláneas

Estas funciones tienen unos mecanismos demasiado concretos como para introducirlos en otra categoría:

Nombre	Argumentos	Retorno	Descripción
qt	[]	String	Devuelve un String con el carácter “” (comillas dobles)
parse	[String]	Any	Devuelve un objeto resultante de parsear la cadena dada en el caso de que se pueda
time	[f : Functional, its : Number]	Number	Devuelve el tiempo de ejecución medio en milisegundos de la expresión funcional dada. La expresión no debe tener parámetros
hash	[arg]	Number	Obtener hash del objeto pasado como parámetro en el caso de que se pueda calcular
sizeof	[arg]	Number	Obtener tamaño en memoria en bytes del objeto pasado como parámetro. Si no es posible saberlo exactamente se dará la mejor aproximación posible
probability	[p : Number]	Boolean	Devuelve True con una probabilidad de un $(100 \cdot p)\%$. Precisamente por la naturaleza de la función, el parámetro p debe encontrarse en el intervalo $[0, 1]$
comparing	[crit : Functional]	Comparator	Devuelve un objeto de tipo Comparator que utiliza como criterio la expresión funcional pasada como parámetro
	[crit : Functional, inv : Boolean]	Comparator	Devuelve un objeto de tipo Comparator que utiliza como criterio la expresión funcional pasada como parámetro. Si *inv* es True, el criterio será invertido
sym	[name : String]	Symbolic	Devuelve una variable simbólica que tiene *name* como representación interna. Esta es la manera estándar de construir variables simbólicas

Funciones miembro básicas

La versión estándar del intérprete de Ulan viene con varias funciones miembro definidas por defecto para cada clase básica, siendo estas fácilmente catalogadas como constantes y no constantes dependiendo de su semántica. A continuación se especifican todas ellas junto con su significado.

Boolean

Este tipo no tiene ninguna función miembro propia, dada su simplicidad.

String

Estas funciones sirven para modificar elementos de un String y aplicarle transformaciones (algunas aplicables con funciones externas).

Nombre	Argumentos	Retorno	Descripción
remove	[Number]	Void	Se elimina el carácter en la posición especificada en el argumento
	[from : Number, size : Number]	Void	Se eliminan los caracteres en el intervalo [*from*, *from + size*)
matches $O(n \cdot \max(O(p)))$	[pattern: String]	Boolean	Comprobar si el String de llamada sigue el formato especificado en *pattern*, siendo este una línea de USDL
size	[]	Number	Devuelve el tamaño del String
at	[n : Number]	String	Devuelve el carácter en la posición *n*-ésima en el String de llamada
replace	[n : Number, char : String]	Void	Cambiar el carácter en la posición *n*-ésima por *char* en el String de llamada.

Number

Estas funciones pertenecen a la clase Number y se dedican, principalmente, a modificar atributos internos de la clase y a hacer comprobaciones de propiedades, ya que las funciones de cálculo suelen ser externas.

Nombre	Argumentos	Retorno	Descripción
negate	[]	Void	Se cambia el signo del Number de llamada
setPrecison	[Number]	Void	Se cambia la precisión individual del número (ver detalles de implementación de Number
isEven	[]	Boolean	Devuelve True si el Number de llamada es par

Nombre	Argumentos	Retorno	Descripción
isPrime	[]	Boolean	Devuelve True si el Number de llamada es primo
isPerfectSquare	[]	Boolean	Devuelve True si el Number de llamada es un cuadrado perfecto
getIntDigitNumber	[]	Number	Devuelve el número de dígitos enteros en base 10 del Number de llamada
getDecDigitNumber	[]	Number	Devuelve el número de dígitos decimales del Number de llamada
getIntDigit	[n : Number]	Number	Devuelve el *n*-ésimo dígito entero en base 10 del Number de llamada
getDecDigit	[n : Number]	Number	Devuelve el *n*-ésimo dígito decimal en base 10 del Number de llamada
mantissa	[]	Number	Devuelve la mantisa del Number de llamada
round	[n : Number]	Void	Redondea el Number de llamada a *n* cifras decimales

Matrix

Estas funciones pertenecen a la clase Matrix y sirven para modificar los números en diferentes posiciones o aplicar transformaciones elementales, entre otras cosas.

Nombre	Argumentos	Retorno	Descripción
at	[i : Number, j : Number]	Number	Obtener el número en la posición (*i*, *j*) en la matriz de llamada
replace	[i : Number, j : Number, n : Number]	Void	Sustituir el número en la posición (*i*, *j*) por *n* en la matriz de llamada
getRowNumber	[]	Number	Obtener el número de filas de la matriz de llamada
getColNumber	[]	Number	Obtener el número de columnas de la matriz de llamada
round	[n : Number]	Void	Redondea todos los números de la matriz de llamada a *n* cifras decimales
rij	[i : Number, j : Number]	Void	Permutar las filas *i*-ésima y *j*-ésima
ria	[i : Number, a : Number]	Void	Escalar la fila *i*-ésima por un factor *a*
rija	[i : Number, j : Number, a : Number]	Void	Sumarle a la fila *i*-ésima la fila *j*-ésima escalada por un factor *a*

Nombre	Argumentos	Retorno	Descripción
cij	[i : Number, j : Number]	Void	Permutar las columnas *i*-ésima y *j*-ésima
cia	[i : Number, a : Number]	Void	Escalar la columna *i*-ésima por un factor *a*
cija	[i : Number, j : Number, a : Number]	Void	Sumarle a la columna *i*-ésima la columna *j*-ésima escalada por un factor *a*
echelon	[canon : Boolean, pivoting : Boolean]	Void	Escalonar la matriz de llamada parcialmente (*canon* = False) o de forma canónica (*canon* = True). Tiene un parámetro para activar el pivoteo
trace	[]	Number	Devuelve la traza de la matriz
rank	[]	Number	Devuelve el rango de la matriz
det	[]	Number	Devuelve el determinante de la matriz
transpose	[]	Matrix	Devuelve la matriz traspuesta
inverse	[]	Matrix	Devuelve la matriz inversa
solve	[system : Matrix, vector : Matrix]	Container	Devuelve la solución del sistema lineal planteado por la matriz y el vector

Symbolic

Estas funciones pertenecen a la clase Symbolic y se aplican transformaciones a los objetos de este tipo.

Nombre	Argumentos	Retorno	Descripción
simplify	[Boolean]	Void	Se simplifica el Symbolic de llamada. Opcionalmente se puede expandir durante la simplificación para obtener mejores resultados
expand	[]	Void	Expandir todos los términos del Symbolic de llamada
diff	[var : Symbolic]	Symbolic	Calcular derivada del Symbolic de llamada con respecto a *var*
	[var : Symbolic , n : Number]	Symbolic	Calcular derivada *n*-ésima del Symbolic de llamada con respecto a *var*
ndiff	[var : Symbolic , n : Number]	Number	Calcular derivada en *var* = *n* de manera numérica
nintegrate	[var : Symbolic , s : Number, e : Number]	Number	Calcular integral definida entre *s* y *e* de manera numérica

Nombre	Argumentos	Retorno	Descripción
roots	[]	Container	Devuelve las raíces del Symbolic de llamada. Los algoritmos a aplicar varían dependiendo del tipo de expresión. Se pueden introducir argumentos opcionales en forma de Mapper con las siguientes claves: • **maxRoots:** número máximo de raíces a calcular si es necesario aproximarlas • **xmin/xmax:** intervalo en el que se elegirán las estimaciones iniciales para las raíces • **rootSeparation:** separación mínima entre raíces para considerarlas raíces distintas • **bracketingIts:** número de iteraciones realizadas en la fase de bracketing (más información en la siguiente sección)
	[Mapper]	Container	

Container

Estas funciones pertenecen a la clase Container y sirven para ejecutar diferentes algoritmos en los elementos dentro de los mismos. Muchas de estas funciones tienen equivalentes externos.

Nombre	Argumentos	Retorno	Descripción
at	[n : Number]	Any&&	Devolver una referencia constante al elemento en la posición *n*-ésima en el Container de llamada si el subtipo lo permite (usar operador [] para obtener una referencia común)
size	[]	Number	Devolver el número de elementos dentro del Container de llamada
add	[obj : Any]	Void	Añadir *obj* al Container de llamada. En casos en los que se permite añadir al principio y al final de mismo este método será equivalente a *pushBack*

Nombre	Argumentos	Retorno	Descripción
pushBack	[obj : Any]	Void	Añadir *obj* al final del Container de llamada. En casos en los que la posición sea irrelevante o tengan funciones especiales (Containers ordenados, por ejemplo) se añadirá de manera genérica
pushFront	[obj : Any]	Void	Añadir *obj* al principio del Container de llamada. En casos en los que la posición sea irrelevante o tengan funciones especiales (Containers ordenados, por ejemplo) se añadirá de manera genérica
remove	[obj : Any]	Void	Eliminar un elemento igual a *obj* en el Container de llamada si lo hubiera
removeAll	[obj : Any]	Void	Eliminar todos los elementos iguales a *obj* en el Container de llamada
removeIndex	[n : Number]	Void	Eliminar el elemento en la posición *n*-ésima del Container de llamada
contains	[obj : Any]	Boolean	Devuelve true si el Container de llamada contiene un objeto igual a *obj*

Mapper

Estas funciones pertenecen a la clase Mapper y sirven para ejecutar diferentes algoritmos en los elementos dentro de los mismos.

Nombre	Argumentos	Retorno	Descripción
at	[key : Any]	Any&&	Devolver una referencia constante al elemento asociado a la clave *obj*
values	[]	Container	Devolver un Container con los valores del array asociativo
keys	[]	Container	Devolver un Container con las claves del array asociativo
add	[key : Any, Value : Any]	Void	Añadir el par (*key*, *value*) al array asociativo. Si *key* ya tiene un valor asociado, la función no tiene ningún efecto
containsKey	[key : Any]	Boolean	Comprobar si existe la clave *key* en el Mapper de llamada
containsValue	[value : Any]	Boolean	Comprobar si existe el valor *value* en el Mapper de llamada

Nombre	Argumentos	Retorno	Descripción
remove	[obj : Any]	Void	Eliminar todos las *keys* iguales a *obj* en el Mapper de llamada
	[key : Any, value : Any]	Void	Eliminar el par (*key*, *value*) del Mapper de llamada en el caso de que lo hubiera

Image

Estas funciones pertenecen a la clase Image y permiten dibujar figuras básicas a nivel de píxeles sin tolerancia a decimales.

Nombre	Argumentos	Retorno	Descripción
getHeight	[]	Number	Obtener la altura en píxeles de la imagen de llamada
getWidth	[]	Number	Obtener la anchura en píxeles de la imagen de llamada
getPixel	[x : Number, y : Number]	Container	Obtener el color del pixel en las coordenadas (*x*, *y*)
setPixel	[x : Number, y : Number, color : String]	Void	Asignar el color pasado como parámetro al píxel de la imagen de llamada que se encuentra en la posición (*x*, *y*)
	[x : Number, y : Number, color : Container]	Void	
line	[x1 : Number, y1 : Number, x2 : Number, y2 : Number, color : String]	Void	Dibujar línea desde (*x1*, *y1*) hasta (*x2*, *y2*) en la imagen de llamada con el color especificado
	[x1 : Number, y1 : Number, x2 : Number, y2 : Number, color : Container]	Void	
rect	[x1 : Number, y1 : Number, x2 : Number, y2 : Number, color : String, fill : Boolean]	Void	Dibujar un rectángulo con diagonal desde (*x1*, *y1*) hasta (*x2*, *y2*) en la imagen de llamada con el color especificado y opcionalmente rellenado con el color elegido
	[x1 : Number, y1 : Number, x2 : Number, y2 : Number, color : Container, fill : Boolean]	Void	

Nombre	Argumentos	Retorno	Descripción
triangle	[x1 : Number, y1 : Number, x2 : Number, y2 : Number, x3 : Number, y3 : Number, color : String, fill : Boolean]	Void	Dibujar un triángulo con vértices (*x1*, *y1*), (*x2*, *y2*) y (*x3*, *y3*) en la imagen de llamada con el color especificado y opcionalmente rellenado con el color elegido
	[x1 : Number, y1 : Number, x2 : Number, y2 : Number, x3 : Number, y3 : Number, color : Container, fill : Boolean]	Void	
circle	[x : Number, y : Number, radius : Number, color : String, fill : Boolean]	Void	Dibujar un círculo con centro en (*x1*, *y1*) y radio *radius* en la imagen de llamada con el color especificado y opcionalmente rellenado con el color elegido
	[x : Number, y : Number, radius : Number, color : Container, fill : Boolean]	Void	
arc	[x : Number, y : Number, radius : Number, a1: Number, a2 : Number color : String, fill : Boolean]	Void	Dibujar un arco con centro en (*x1*, *y1*) y radio *radius* desde *a1* radianes hasta *a2* en la imagen de llamada con el color especificado y opcionalmente rellenado con el color elegido
	[x : Number, y : Number, radius : Number, a1: Number, a2 : Number color : Container, fill : Boolean]	Void	
polygon	[points : Container, color : String, fill : Boolean]	Void	Dibujar polígono con vértices en las coordenadas especificadas en *points* con el color especificado y opcionalmente rellenado con el color elegido. El polígono resultante debe ser convexo para que se rellene de forma correcta
	[points : Container, color : Container, fill : Boolean]	Void	
save	[path : String]	Void	Guardar la imagen en la ruta especificada en *path*

Functional

Estas funciones pertenecen a la clase Functional y sirven para obtener información sobre los argumentos y construir funciones complejas.

Nombre	Argumentos	Retorno	Descripción
getArgNumber	[]	Number	Devolver el número de argumentos que toma el Functional de llamada. Los parámetros con tipos expandidos contarán como uno común
invoke	[args : Container]	Any	Llamar a la función con los argumentos pasados como Container en *args*. Esta función es utilizada en librerías para hacer algoritmos de binding y composición de funciones

Comparator

Estas funciones pertenecen a la clase Comparator y sirven para obtener información sobre el objeto y hacer criterios compuestos.

Nombre	Argumentos	Retorno	Descripción
then	[Functional]	Comparator	Devolver un Comparator con un criterio nuevo creado a partir del Functional dado por parámetro
	[Comparator]	Comparator	Devolver un Comparator con que usa los criterios del Comparator dado por parámetro tras usar el del Comparator de llamada
compare	[obj1, obj2]	Number	Comparar *obj1* y *obj2* siguiendo el criterio del Comparator de llamada. El resultado sigue la siguiente norma: $res = \begin{cases} -1 & si\ obj1 < obj2 \\ 1 & si\ obj1 > obj2 \\ 0 & si\ obj1 = obj2 \end{cases}$

Cuerpo de una función

Se le llama cuerpo de una función al bloque indentado en el que define las acciones que realiza una función, recibiendo este una serie de argumentos de alguna manera. Este puede tener todas las líneas de código que sean necesarias y saltos de línea que serán ignorados por el intérprete.

La ejecución del cuerpo de una función es exactamente la misma que la de cualquier otro fragmento de código, pero con la excepción de que devuelve un valor al contexto de llamada (también llamado contexto superior). Este valor es especificado por la sentencia *return*.

La ejecución secuencial de estos bloques es idéntica a la de cualquier otro fragmento de código, pero con la diferencia de que esta se interrumpirá al ejecutar una sentencia *return*, devolviendo el valor a la derecha de la misma. Este es el funcionamiento estándar para cualquier lenguaje imperativo. Cabe destacar el caso especial del retorno de variables, ya que cuando se devuelve una de estas se devuelve el dato directo que está almacenado en su interior. Esto último se debe a que las variables son devueltas como referencias a los valores que contienen independientemente de su tipo.

En el caso de las funciones miembro, al principio del bloque se definirá una variable llamada *self* que almacenará una referencia (constante si la función es contante) al objeto de llamada de la función. Esta será definida en el contexto del cuerpo, por lo que no remplazará a ningún valor definido anteriormente y se podrá referenciar en el código con total normalidad.

Ejemplo:

```
Define function f(n : Number, flag : Boolean) as
    if n < 10
        return n
    else
        return (n/2 if flag else n*2)
```

El cuerpo de esta función se ejecutará hasta la segunda línea si el primer argumento es menor que 10, ya que se ejecutará un *return*. En caso contrario, se ejecutará hasta el final.

Es posible que una función no ejecute un return. En ese caso se dirá que la función no tiene retorno. Esto es interpretado a nivel interno mediante el retorno de un objeto de tipo Void. Cabe destacar que se considerará que una función tiene retorno de tipo Any si no siempre ejecuta un return.

Definición

A la hora de definir funciones en Ulan, se puede optar por definir una función externa o una función miembro, teniendo ambas una sintaxis parecida. También se permite definir funciones con sintaxis especial.

Los nombres de las funciones deben seguir las restricciones especificadas en el apartado de formato de elementos contextuales.

Funciones externas

La sintaxis exacta es:

$$Define\ function\ Nombre(Args)\ as$$

$$(Bloque\ indentado)$$

Siendo $Nombre$ el nombre de la función escrito sin comillas, $Args$ los argumentos de la función entre paréntesis, siguiendo el estándar de la mayoría de lenguajes. El bloque indentado será el cuerpo de la función y tendrá un contexto un nivel superior al de la definición. El tipo de retorno de la función será inferido si es posible.

Funciones miembro

La sintaxis exacta es:

$$Define\ [const]\ function\ TipoM :: Nombre\ (Args)\ as$$

$$(Bloque\ indentado)$$

Siendo $TipoM$ el tipo al que pertenece el objeto de llamada y manteniéndose el resto de definiciones. Si se quiere declarar una función miembro constante basta con utilizar la palabra reservada *const*, pero es necesario tener en cuenta que *self* será una referencia constante en ese caso. El bloque indentado sigue representando al cuerpo de la función y el cambio de contexto es el mismo. Se seguirá infiriendo el tipo de retorno de la función.

Funciones con sintaxis propia

Se pueden definir funciones utilizando una sintaxis definida en USDL y extrayendo los argumentos para la llamada desde ella. La sintaxis exacta a utilizar sería:

$$Define\ syntax\ from\ "Pattern"\ as$$

$$(Bloque\ indentado)$$

Siendo $Pattern$ el patrón que se añadirá al intérprete para detectar a partir de la siguiente línea. El bloque indentado es el cuerpo de la función, pero se definirá una variable de tipo Mapper llamada *args* con los argumentos capturados de la sintaxis. El tipo de retorno se inferirá como en el resto de funciones.

Secuencias

Introducción

Se le llama secuencia a un conjunto de palabras reservadas con expresiones intercaladas para realizar alguna acción. Estas suelen ser controladores de flujo del programa o definiciones de estructuras, pero se ofrecen más posibilidades.

Controladores de flujo

Sentencia if

Esta secuencia es estándar para la mayoría de lenguajes. Consiste en la especificación de una condición lógica y un segmento de código que debe ejecutarse solo si se cumple esa condición. La sintaxis exacta es:

$$If\ (Expresion)$$

$$(Bloque\ indentado)$$

Teniendo que devolver la expresión un valor lógico necesariamente. En caso contrario, su ejecución acarreará un error de tipo. El bloque indentado indica el código que se ejecutará si la expresión se cumple.

Sentencia else

Al igual que la sentencia *if*, *else* es una palabra utilizada para lo mismo en la mayoría de lenguajes imperativos. Esta sirve para especificar un bloque de código que debe ejecutarse en el caso de que no se cumpla la condición de una sentencia *if*. La sintaxis exacta es:

$$If\ (Expresion)$$

$$(Bloque\ indentado)$$

$$else$$

$$(Bloque\ indentado)$$

Siendo el segundo bloque indentado el que se ejecutará en caso de que no se cumpla la condición del if.

Sentencia if-else

Tras ejecutar un *if* es posible encadenar otras condiciones utilizando directamente esta secuencia en vez de usar un *if* dentro de un *else*. La sintaxis exacta es:

$$Else\ if\ (Expresion)$$

$$(Bloque\ indentado)$$

Sentencia if-else inline

Esta sentencia no es más que azúcar sintáctico para el operador ternario, el cual está incluido en muchos lenguajes. Su funcionamiento es similar a un bloque *if-else*, pero devuelve un valor. Se podrá predecir el tipo de este si se devuelve un objeto del mismo tipo en ambos casos. La sintaxis exacta es:

$$(Expresion1)\ if\ (Condicion)\ else\ (Expresion2)$$

Se devolverá el valor de la primera expresión si se cumple la condición y el de la segunda expresión en caso contrario.

Bucle while

Este es el bucle más básico. Repite todo un bloque de código mientras una expresión lógica dada siga evaluándose como verdadera. La sintaxis exacta es:

$$While\ (Condicion)$$

$$(Bloque\ indentado)$$

Siendo el bloque indentado el conjunto de líneas que se ejecutarán de manera serializada. Se puede interrumpir esta secuencia con una secuencia *break*.

Bucle for

Ulan admite únicamente lo que podría ser un equivalente a un bucle *for extendido* en lenguajes como Java o C++. Esta filosofía viene de lenguajes como *Python*, que consideran innecesario implementarlos. La sintaxis exacta es:

$$for\ (Variable)\ in\ (Container)$$

$$(Bloque\ indentado)$$

Siendo la variable especificada en el cuerpo del *for* la que se definirá en un contexto superior dentro del bloque indentado. El Container especificado es el que tiene los elementos sobre los que vamos a iterar y que vamos a asignar a la variable especificada. Es posible adaptar este bucle a Mappers haciendo de $Variable$ una tupla de alto nivel con dos variables (key y value, respectivamente). Se puede interrumpir esta secuencia con una secuencia *break*.

Bucle for múltiple

Así como se permiten los bucles for normales, se permite especificar bucles anidados en una sola línea utilizando tuplas de alto nivel. La sintaxis exacta es la siguiente:

$$for\ (TuplaVariables)\ in\ (TuplaContainers)$$

$$(Bloque\ indentado)$$

Siendo $TuplaVariables$ una tupla de alto nivel con variables como argumentos y $TuplaContainers$ una tupla con Containers como argumentos. Una variable en la posición *n* iterará sobre los valores del *n*-ésimo Container en la tupla. Los elementos de la derecha actuarían como los interiores en el bucle anidado. Se puede interrumpir esta secuencia con una secuencia *break*.

Comprensión de listas

Los mecanismos de comprensión de listas son elementos que crean Containers a partir de bucles implícitos. Estas estructuras hacen que el código sea mucho más legible y están implementadas en muchos lenguajes.

Estas estructuras deben estar encapsuladas en llaves para funcionar correctamente, pudiendo especificar el subtipo de Container con el operador de definición.

For simple inline

El mecanismo de comprensión de listas más básico es el for simple inline. Esta sentencia crea un generador de valores dada cierta fórmula especificada y asigna estos valores a un container del tipo especificado. La sintaxis exacta es:

$$(Expresion)\ for\ (Variable)\ in\ (Container)$$

Esta estructura almacenará en un Container del tipo especificado todos los valores que tome $Expresion$ con cada una de las asignaciones a $Variable$ de los elementos de $Container$.

Se puede añadir una sentencia if al final para filtrar los resultados y almacenar solo los que cumplan cierta condición en el Container. La sintaxis exacta sería la siguiente:

$$(Expresion)\ for\ (Variable)\ in\ (Container)\ if\ (Condicion)$$

Siendo $Condicion$ la condición de filtrado.

Al igual que en el for común, se permite adaptar este bucle a Mappers siguiendo exactamente la misma estrategia, incluyendo la posibilidad de hacer filtros implícitos como en cualquier otra comprensión de listas.

For múltiple inline

El funcionamiento de este mecanismo es exactamente el mismo que el de un for simple inline, pero con la diferencia de que almacena los valores de un bucle anidado en vez de los de un bucle simple. La sintaxis es la siguiente:

$$(Expresion)\ for\ (TuplaVariable)\ in\ (TuplaContainers)$$

Teniendo las variables la misma semántica que en otros mecanismos de comprensión de listas. Es posible añadir una sentencia if para hacer un filtrado implícito, al igual que en bucle for implícito simple, siendo la sintaxis la siguiente:

$$(Expresion)\ for\ (TuplaVariable)\ in\ (TuplaContainers)\ if\ (Condicion)$$

Siendo $Condicion$ la condición de filtrado.

Definiciones

Las definiciones son un caso especial de secuencias en Ulan, ya que sirven para extender la semántica del lenguaje. Estas adiciones pueden ser algo tan simple como una función o algo tan complejo como sintaxis propia que modificará el comportamiento del intérprete.

Estas definiciones, al tener un uso diferente al resto de expresiones en este lenguaje, también tienen unas restricciones a la hora de utilizarlas. Estas restricciones se especifican en el apartado de contextos.

La información específica de cada posible definición en Ulan está en la sección correspondiente a la estructura que define. A continuación se ofrecen enlaces que llevan a cada uno de los apartados con secuencias de este tipo:

- Clases y parsing directo
- Casts
- Operadores y operaciones
- Funciones comunes y con sintaxis especial

Sintaxis propia

Introducción

Una de las características distintivas de Ulan es que el usuario puede definir sintaxis propia utilizando un lenguaje propio llamado USDL (*Ulan Syntax Definition Language*). Un claro ejemplo de esto es la definición de parsing directo de clases, aunque también se puede definir sintaxis que realice una acción dados ciertos argumentos (lo cual es considerado una función con sintaxis especial en este documento).

Esto abre paso a la creación de diferentes paradigmas de programación dentro del propio intérprete de Ulan, lo que lo convierte en una herramienta extremadamente potente cuando es usada por las manos correctas. Asimismo, puede tener un interesante uso experimental.

Patrones USDL

El lenguaje UDSL es una sintaxis que nos permite definir reglas de parsing para el lenguaje de programación Ulan. Este es parecido a la notación de Backus-Naur, pero con la diferencia de que permite especificar qué parte de las cadenas de caracteres son argumentos que hay que parsear y cómo hacerlo.

Estructura básica de USDL

Una línea de USDL contiene varios artefactos que son interpretados de izquierda a derecha. Cada uno de estos indica una regla que debe seguir una cadena para que cumpla el patrón especificado. Si todos los patrones se cumplen correctamente y en orden (de izquierda a derecha), se dice que la cadena de caracteres cumple el patrón compuesto.

Asimismo, también se pueden especificar las localizaciones exactas de argumentos parseables en una cadena junto al formato (Marcadores USDL). Esta técnica es utilizada para especificar constructores por String para clases definidas por el usuario.

Una línea de USDL es interpretada a partir de un String en Ulan. Este String debe contener los patrones con sus respectivas sintaxis en el orden que se quieran evaluar y separados por un número arbitrario de espacios (al menos uno). Cada patrón puede tener subpatrones anidados dependiendo de su tipo. A continuación se especifican todos los subpatrones definidos en USDL.

Patrón texto

Este es el patrón básico para la construcción de una regla de sintaxis. Con este patrón se puede especificar que haya un cierto texto en una cadena. La sintaxis BNF es la siguiente:

$$< ASCIIChar >::= (Cualquier\ caracter\ ASCII)$$

$$< textPattern > ::= "'" < ASCIIChar > \ \{< ASCIIChar >\}\ "'"$$

Tal y como se puede observar, no se permiten patrones texto vacíos, el patrón debe contener un carácter como mínimo.

Patrón texto difuso

Este es un patrón derivado del patrón texto. Permite especificar un texto parecido al especificado, pero con la misma longitud. La similitud entre dos cadenas es calculada utilizando la distancia de Levenshtein. La sintaxis exacta es la siguiente (presuponiendo la definición de la clase básica Number):

$$< ASCIIChar >::= (Cualquier\ caracter\ ASCII)$$

$$< textPattern > ::= "f" < Number > "'" < ASCIIChar > \ \{< ASCIIChar >\}\ "'"$$

Dada esta sintaxis podemos observar que es necesario introducir un número y una cadena no vacía, la cadena es la que se usará para comparar con la sintaxis a analizar y el número puede tener dos significados dependiendo de su valor:

- **Valor mayor o igual que 1**: El número indica la distancia de Levenshtein máxima entre la cadena especificada y la cadena a analizar. Debe ser un entero.
- **Valor menor que 1**: El número indica el coeficiente de similitud mínimo entre la cadena especificada y la cadena a analizar. Este coeficiente se calcula dividiendo la longitud de las cadenas entre la distancia de Levenshtein entre ambas. Debe ser un número positivo.

Dado que este patrón tiene un poder expresivo muy grande, se restringe su definición a cadenas que solo contengas letras, dígitos, espacios o el carácter "_" para no provocar inconsistencias en el estándar (ver estabilidad de encapsulamiento). Tampoco identificará como correctas a subcadenas que contengan cualquier otro carácter.

Patrón opcional

Este patrón indica uno o varios subpatrones que pueden seguirse o no en la cadena que se analiza. Dicho de otra forma, indica una serie de patrones opcionales. La sintaxis BNF exacta es la siguiente:

$$< pattern >::= (Cualquier\ patrón\ USDL)$$

$$< s >::= " "\ \{" "\}\ \ //Al\ menos\ un\ espacio$$

$$< optionalPattern > ::= "["\ \{" "\}\ < pattern > \{< s >< pattern >\}\ \{" "\}\ "]"$$

Patrón repetición

Este patrón indica uno o varios subpatrones que pueden seguirse un número arbitrario de veces. Es posible indicar el número mínimo y máximo de repeticiones. La sintaxis exacta es la siguiente (presuponiendo la definición de la clase básica Number):

$$< pattern > ::= (Cualquier\ patrón\ USDL)$$

$$< s > ::= \text{" "} \{\text{" "}\}\ //Al\ menos\ un\ espacio$$

$$< seriesPattern > ::= [< Number >] \text{"}\{\text{"} \{\text{" "}\}\ < pattern > \{< s > < pattern >\} \{\text{" "}\} \text{"}\}\text{"}\ [< Number >]$$

Se puede observar que hay dos números opcionales al principio y al final del patrón. Estos indican el número mínimo y máximo de repeticiones, respectivamente, y deben ser números enteros. Si falta el primero se considerará que el número mínimo es 0 y si falta el segundo se considerará que el número máximo es indeterminado (infinitas repeticiones, potencialmente).

Patrón alternativa

Este patrón funciona como un operador e implica cualquiera de las alternativas dadas como operandos son válidas. Es posible agrupar patrones con paréntesis para especificar grupos de patrones. La sintaxis exacta es la siguiente:

$$< pattern > ::= (Cualquier\ patrón\ USDL)$$

$$< orPattern > ::= < pattern > \{\text{" "}\} \text{"|"} \{\text{" "}\} < pattern > \{\{\text{" "}\} \text{"|"} \{\text{" "}\} < pattern >\}$$

Dada tal estructura tan solo uno de los operandos se cumplirá en la cadena analizada. Si se cumplen varios se escogerá el primero empezando desde la izquierda que lo haga.

Patrón por símbolos reservados

Este patrón funciona como un patrón texto, pero no va encapsulado en comillas y cada símbolo representa un conjunto de posibles caracteres en vez de uno solo. Los símbolos reservados válidos y la sintaxis exacta se especifican a continuación:

$$< letter > ::= \text{"l"}\ //Cualquier\ letra$$

$$< digit > ::= \text{"}d\text{"}\ //Cualquier\ dígito$$

$$< quote > ::= \text{"}q\text{"}\ //Comilla\ simple$$

$$< anyChar > ::= \text{" } * \text{ "}\ //Cualquier\ símbolo$$

$$< symbol > ::= < digit > |< letter >| < quote > | < anyChar >$$

$$< symbolPattern > ::= < symbol > \{< symbol >\}$$

Patrón rango

Este patrón admite un carácter que se encuentre entre dos caracteres especificados, ambos inclusive. La sintaxis exacta es:

$$< Char > ::= (Cualquier\ caracter\ de\ 1\ byte)$$

$$< rangePattern > ::= "["\ < Char >\ "-"\ < Char >\ "]"$$

Se puede observar que hay dos caracteres que se especifican entre corchetes en la sintaxis. El carácter de la izquierda es el carácter mínimo y el de la derecha es el carácter máximo del rango.

Ningún carácter encapsulador será considerado válido en la definición o ejecución de este patrón.

Patrón clase

Este patrón permite al usuario definir una regla que coincida con la definición de la sintaxis de una clase. Estos se utilizan para abreviar expresiones que incluyen números o patrones complicados. La sintaxis exacta es la siguiente:

$$< className > ::= (Nombre\ de\ una\ clase\ definida\ en\ Ulan)$$

$$< classPattern > ::= "<"\ < className >\ ">"$$

El nombre de la clase introducido entre los símbolos “<” y “>” determinará la clase cuyo patrón se quiere buscar.

Marcadores USDL

Los marcadores son la segunda parte del lenguaje USDL. Estos se utilizan para especificar qué parte de la cadena que se está analizando es un argumento a parsear y cómo hacerlo exactamente.

Identificadores

Los identificadores son palabras reservadas que indican a qué tipo de sintaxis nos estamos refiriendo cuando seleccionamos una subcadena. Dada esta subcadena podemos categorizarla utilizando uno de los siguientes identificadores, siendo estos independientes de mayúsculas y minúsculas:

- **Classname**: Este identificador tiene el nombre de una clase directa e indica que una cadena es un elemento de dicha clase. Es sensible a mayúsculas y minúsculas.
- **Lit**: Este identificador indica que una cadena es un String, pero hace que no sea necesario introducir las comillas. Esto es utilizado para procesar cadenas complejas directamente del texto introducido.
- **Val**: Este identificador indica que una cadena es un valor cualquiera. Esta cadena se intentará clasificar como clase directa o variable, en ese orden. Marcador argumento

Este marcador es la herramienta principal para indicar posiciones de argumentos en cadenas parseables. La sintaxis de este artefacto es parecida a la de una función, esta es la sintaxis exacta:

$$< id > ::= (Cualquier\ identificador)$$

$$< USDLPattern > ::= (Cualquier\ patrón\ USDL, simple\ o\ complejo)$$

$$< str > ::= (Cadena\ de\ caracteres\ ASCII)$$

$$< argMarker > ::= "Arg("\ \{"\ "\}\ < USDLPattern > \{"\ "\}\ ","\ \{"\ "\}\ < id > \{"\ "\}\ ","\ \{"\ "\}\ < str > \{"\ "\}\ ")"$$

Dada esta sintaxis, el patrón dentro del primer parámetro de la función será identificado como un argumento del tipo del identificador del segundo argumento con el nombre del tercer argumento. Estos son los datos que se le pasarán al cuerpo de la función, en el caso de que esta sintaxis se utilice en una definición.

Todas las ocurrencias de un marcador argumento con un cierto nombre (tercer argumento) deben tener el mismo identificador, sino su definición acarreará un error de formato.

Para especificar un argumento múltiple basta con usar varios marcadores argumento con el mismo nombre. Cada vez que se detecte un argumento con el mismo nombre que uno anterior, se considerará que este es un argumento múltiple y se unificarán.

Características de una sintaxis propia

Familia

Se dice que una cadena forma parte de la familia de una sintaxis propia si cumple las restricciones de formato especificadas en la línea de USDL que la define.

Estabilidad de encapsulamiento

Se dice que una sintaxis propia tiene estabilidad de encapsulamiento cuando existen cadenas en su familia que tienen el mismo número de encapsuladores de apertura que de cierre **de cada tipo**, teniendo en cuenta que un encapsulador dentro de un String no cuenta como símbolo válido (se omiten). Esta norma se aplica haciendo que un patrón solo pueda ser satisfecho por una cadena que cumpla esa condición.

Estabilidad de operadores

Se dice que una sintaxis propia tiene estabilidad de operadores si ninguna cadena perteneciente a su familia tiene ninguna subcadena que coincida con algún operador definido. Esta no es una característica necesaria para la aceptación de una sintaxis, pero si no se cumple podrían causarse resultados inesperados durante el parsing si la estructura no está encapsulada en paréntesis.

Superposición

Se dice que las sintaxis propias tienen la propiedad de **superposición**, lo cual no es más que una desambiguación que especifica que cualquier sintaxis propia que se superponga total o parcialmente con una *sintaxis nativa* (entendiendo por sintaxis nativa cualquier sintaxis que no haya sido definida por el programador) la sobrescribe, teniendo prioridad una sintaxis de clase sobre una sintaxis de función. Dicho de otra manera:

Dada la familia F_p de una sintaxis propia S_p y la familia F_n de una sintaxis nativa S_n, se puede afirmar que si $F_p \cap F_n \neq \emptyset$, el parser actuará dando por hecho que:

$$(F_p \cap F_n) \subseteq S_p$$

$$(F_p \cap F_n) \nsubseteq S_n$$

Teniendo en cuenta esto, cabe destacar que no habrá ningún tipo de aviso para cuando una sintaxis sobrescriba alguna característica clave del lenguaje, por lo que es necesario extremar la precaución al definir un patrón USDL similar a alguna sintaxis nativa.

Otro tipo de superposición importante es el que ocurre entre diferentes sintaxis definidas por el usuario. Estos conflictos son los más importantes porque **no existe manera de saber a cuál se le dará prioridad**. Esto se debe a que es imposible para el usuario conocer el orden exacto de instantación, así que no vale la pena implementarlo.

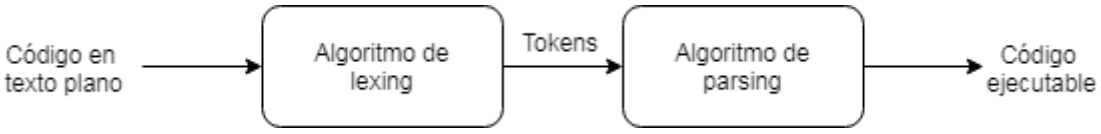
Sección IV

Implementación

Visión general

Procesado de código

El proceso para transformar cadenas de texto en código ejecutable es relativamente estándar, por lo que el intérprete no se desvía demasiado de las convenciones. Obviando optimizaciones de código, el flujo de los datos hasta su estado final sería el siguiente:



La parte personalizada comienza en el momento que introducimos rutinas de reescritura de código con fines de optimización. Estas se le aplicarían al resultado generado al final del flujo anterior y generarían otro código equivalente bajo ciertas asunciones. Esta parte podría ser considerada un módulo del intérprete de por sí, pero no se explicará en profundidad por su corta funcionalidad en su estado actual.

Comprobación de código

El código interpretado debe pasar una parte de validación para ser considerado correcto. Esta fase, se subdivide a su vez en otras dos: inferencia de tipos y comprobación semántica. Estas dos son ejecutadas una vez por cada expresión ejecutable (o varias si es necesario por alguna optimización) y tienen una finalidad clara:

- **Inferencia de tipos:** se realizan análisis de diferente tipo dependiendo de qué tipo de expresión se esté tratando y se intenta prever el tipo del resultado de la ejecución de la expresión. Esto permite hacer análisis estático de código y es una herramienta que habilita el tipado gradual.
- **Comprobación semántica:** se intentan detectar las combinaciones de expresiones y argumentos que causarían un error en tiempo de ejecución. Estos pueden ir desde una secuencia *if* con un argumento que no es evaluado a un objeto de tipo *Boolean* hasta una comprensión de listas que itera sobre elementos que no son de clase *Container*.

No todas las expresiones incorrectas son captadas en tiempo de interpretación, siendo un claro ejemplo las llamadas a función con parámetros no tipados, ya que estos plantean una incertidumbre que puede ser tratada de manera más sencilla (y efectiva en algunos casos) por omisión completa. Si no se detecta un error en tiempo de interpretación, será detectado durante la ejecución y se mostrará un error correcto para evitar fallos en el sistema.

Librerías básicas

Number

Estructura básica

Esta librería fue la primera en implementarse para este proyecto y su desarrollo implicó bastante investigación, ya que los algoritmos utilizados son muy específicos y abarcan bastantes áreas. Muchos de estos están optimizados paralelizándolos de manera transparente con una librería gestora de hilos.

Un objeto de tipo Number contiene dos arrays dinámicos que almacenan los dígitos enteros y los dígitos decimales. Por detalles de implementación no relevantes, en este caso los dígitos enteros están en orden inverso (el primer elemento del array es el último del número) y se decidió no cambiarlo para no rehacer los algoritmos básicos. Se decidió utilizar arrays dinámicos para disminuir la dificultad de la tarea, pero estos están usados de manera eficiente.

Los dígitos enteros están almacenados en base 10^9 para ahorrar espacio y hacer los cálculos más rápidos. Los dígitos decimales están almacenados en base 10 para manejar mejor el redondeo y la precisión. Se decidió no implementarlo en base 2 por los problemas que conlleva la conversión de bases posterior para la representación como cadena (puede ser inviable para números grandes) y por el desconocimiento de algoritmos efectivos.

También se almacena un atributo booleano que indica si el número es negativo. Este se utiliza de manera activa en las operaciones básicas para comprobar los casos de suma y resta, ya que las funciones básicas están implementadas haciendo ciertas suposiciones comentadas más adelante.

Una parte distintiva de esta librería es que los números poseen una precisión local, pese a existir una variable que almacena la precisión global. Esto se hace para evitar problemas de precisión en los algoritmos de aproximación, ya que si se usara únicamente la variable global esto causaría problemas en algoritmos multihilos.

Dicho esto, los atributos básicos están representados en el código con tipos nativos de la siguiente manera:

```
std::vector<uint32_t> intDigits;
std::vector<uint8_t> decDigits;
bool negative;
unsigned int precision;
```

Operaciones básicas

En este documento se definen las operaciones básicas como cualquiera que pueda ser asignada a un operador estándar con la prioridad correcta en C++ o, dicho de otra forma, la suma, resta, multiplicación, división y módulo.

Lo primero es implementar dos funciones: una que sume dos números positivos cualesquiera y otra que reste dos números positivos suponiendo que el primero es mayor que el segundo (calcula $a - b$ cuando $a \geq b \geq 0$). La lógica de estas consiste en agrupar los números de la misma manera que se haría una suma a mano, pero con dígitos de base mayor. El dígito de carry se guarda en una variable de tipo bool que se inicializa como a false (que es interpretado como un 0 al hacer el cast implícito a uint32_t).

Después de implementar estas dos funciones es necesario implementar otra que compruebe si un número es menor que otro. Esta función tiene una versión que tiene en cuenta el signo y otra que no. La implementación es bastante engorrosa y tiene en cuenta muchos casos, por lo que no quedará reflejada en este documento, pero la lógica es sencilla. Al acabar, se puede proceder a generalizar la suma y la resta comprobando los diferentes casos posibles y teniendo en cuenta propiedades tales como $a + (-b) = a - b$.

Tras haber generalizado estas dos se procede a implementar la multiplicación, tarea para la que hay que implementar varias funciones dependiendo de los algoritmos que se utilicen. En este caso, se han implementado 3:

- **Multiplicación trivial $O(1)$:** este algoritmo se basa en la conversión a tipos nativos para el cálculo de productos, por lo que solo se utiliza con número pequeños.
- **Multiplicación lenta $O(n^2)$:** este es el algoritmo de multiplicación básico, no tiene mucho que comentar. Consiste en sumas repetidas con multiplicaciones de orden constante.
- **Multiplicación de Karatsuba $O\left(n^{\frac{log(3)}{\log(2)}}\right)$**: este algoritmo es utilizado a partir de un n almacenado en una variable global que se calculó de manera experimental. Es necesario implementar funciones de suma optimizada para vectores de números enteros.
- **Multiplicación Toom-3 $O\left(n^{\frac{log(5)}{\log(3)}}\right)$:** este algoritmo requiere más trabajo que el anterior, ya que es necesario implementar un algoritmo de división de orden n, lo cual es relativamente complejo. También es necesario que utilice varios hilos para que tenga una diferencia notable. Se utiliza una secuencia de operaciones definida por Marco Bodrato para la interpolación [14]. Este algoritmo también se utiliza a partir de un límite calculado de manera experimental.

Lo siguiente que habría que implementar sería el algoritmo de división, lo cual es relativamente sencillo. Se implementan 3 algoritmos: Uno de división trivial (conversión a tipos nativos), uno de división N:1 (el utilizado en el algoritmo Toom-3) y la división larga. Es posible implementar un algoritmo de división rápida, pero se consideró que el rendimiento era aceptable para la mayoría de los casos. Dada la división, el módulo tiene una implementación trivial (se implementaron casos orientados a vectores para mejorar el rendimiento).

Funciones matemáticas

La mayoría de los cálculos matemáticos realizados en esta librería dependen enteramente de las operaciones básicas y son óptimos para la implementación dada, pero hay algunos casos especiales que tienen orientación a vectores y/o algoritmos multihilos transparentes (como es el caso del factorial).

Algunos de estos cálculos utilizan constantes precalculadas para poder funcionar (como es el caso de la función exponencial). Se implementan algoritmos spigot para poder calcular estas hasta el número de decimales que requiera la operación solicitada, tras lo cual se actualiza una representación como string global que las almacena. Esto se hace con e,π y $\ln(10)$, pero esta última no tiene algoritmo asignado a parte de la aproximación directa iterativa.

Se podría considerar algún caso especialmente difícil, como la función gamma o la función W de Lambert. Estas necesitaron utilizar algoritmos de aproximación especiales y, en el caso de la función gamma, una función compleja únicamente para calcular el número de términos que era necesario expandir en la aproximación de Spouge.

Primalidad y factorización

Esta librería incluye algoritmos de comprobación de primalidad y factorización que requirieron bastante investigación y pruebas. Se describen a continuación:

El algoritmo de comprobación de primalidad consiste en un test de Miller-Rabin de 10 iteraciones en el caso general (lo cual significa que tiene una probabilidad de fallo de $\frac{1}{4^{10}} \approx 0.0001\%$). Aun así, se utilizan variantes deterministas para números menores que 10^9 (se utiliza la base $\{2, 7, 61\}$) y para números menores que 10^{18} (se utiliza la base $\{2, 325, 9375, 28178, 450775, 9780504, 1795265022\}$), por lo que la probabilidad de acierto para números menores que este último es 100%. También hay algoritmos para comprobar directamente varias divisibilidades sin recurrir a algoritmos más pesados (2, 3 y 5).

Para la factorización se utiliza un algoritmo multihilos mixto Lenstra-Shanks que utilizará obligatoriamente un método de Lenstra y, dado un número de núcleos libres en el procesador $n > 0$, utilizará 1 método de Shanks y $n - 1$ métodos de Lenstra adicionales de manera paralela. Utilizando este algoritmo se han llegado a factorizar números de más de 30 cifras, pero se ha observado que suele funcionar mejor para números con factores pequeños, ya que el algoritmo de factorización general implementado no es muy potente.

También se realizan optimizaciones al algoritmo de factorización en el caso de que el número a factorizar sea un cuadrado perfecto (lo cual tiene su propia función optimizada) y en el caso de que haya un factor múltiple (se comprueba la multiplicidad de cada factor al calcularlo). Cabe destacar que se hace una criba rápida para primos pequeños antes de utilizar ningún otro algoritmo más pesado.

Symbolic

Estructura básica

En esta implementación, todas las expresiones simbólicas se representan como un árbol, estando este representado con tres atributos. Uno de estos atributos es de un tipo indeterminado y dependerá del tipo de nodo del que se trate.

El primer atributo es un enumerado que indica el tipo del nodo al que representa. Los tipos reflejados en ese enumerado son los siguientes:

- **Number:** El nodo es un número.
- **Constant:** El nodo es una contante matemática.
- **Variable:** El nodo es una variable simbólica.
- **Function:** El nodo es una función matemática.
- **Operation:** El nodo es una operación.

El segundo atributo es un entero sin signo que representa un identificador que es utilizado para acceder a un objeto almacenado en una lista global (constantes, funciones y operadores). Pese a poder ser especificado dentro del propio parámetro variable, se decidió que se tuviese siempre acceso a él para no complicar en exceso la implementación y para que el acceso a los nodos hijo tuviera menos saltos.

El tercer parámetro depende enteramente del primero, ya que se trata de un puntero genérico que se inicializa y gestiona como un tipo diferente dependiendo del tipo de nodo al que pertenezca. Este almacena los datos necesarios para que el nodo funcione correctamente. A continuación se especifica a qué se inicializa para cada tipo:

Nodo	Tipo asociado	Descripción
Number	`Number`	Se almacena el valor del número del nodo
Constant	`NULL`	No se almacena nada
Variable	`std::string`	Se almacena la representación como cadena de la variable
Function	`std::vector<Symbolic>`	Se almacenan los nodos hijo del nodo actual. Se interpretan como argumentos
Operation	`std::vector<Symbolic>`	Se almacenan los nodos hijo del nodo actual. Se interpretan como operandos

Habiendo visto todo esto, la representación como tipos nativos es la siguiente:

```
SymType type;
unsigned int id;
void* data;
```

Simplificación

Se implementa un algoritmo básico de simplificación que sirve en la mayoría de expresiones poco complejas. Este método tiene varias reglas de reducción programadas directamente y se ejecutan siguiendo un cierto patrón. La secuencia que ejecuta se explica a continuación:

1. Se cambian todos los números negativos por su valor absoluto negado por un nodo con el operador de negación. Esto facilita el reconocimiento de patrones en el árbol.
2. Se reduce el número de negaciones "sacándolas" de los productos y las divisiones cuando es posible. También se introduce un caso de simplificación para las potencias de base negativa y exponente impar.
3. Se eliminan los términos que no contribuyen a las operaciones (como los ceros en las sumas, por ejemplo).
4. Se convierten todas las restas a sumas para hacer más sencillo el reconocimiento de patrones.
5. Se expanden los paréntesis si se ha especificado con su correspondiente parámetro.
6. Se ejecuta un método recursivo complejo que recorre el árbol agrupando términos y haciendo diversas optimizaciones especificadas en el código dependiendo del tipo de nodo.
7. Se ordenan los términos siguiendo un criterio de prioridad.

Es cierto que este algoritmo no es óptimo, pero hacer un método más complejo no valía la pena, dado el estado de la aplicación. Se planea mejorarlo en un futuro.

Cálculo de raíces de polinomios

Para esta tarea se utiliza un algoritmo mixto PMR-Aberth-Newton que se divide en diferentes fases:

1. Se obtienen las raíces enteras con la regla de Horner-Ruffini.
2. Se Aplica el método PMR para transformar las raíces múltiples en raíces simples.
3. Se aproximan las raíces restantes con el método de Aberth. Estas raíces son refinadas con el método de Newton para evitar posibles errores de aproximación.
4. Se calculan las multiplicidades de las raíces aproximadas (estos cálculos forman parte del método PMR) y se añaden a la solución.
5. Se devuelve el conjunto de las soluciones enteras y aproximadas ordenado de menor a mayor.

Este algoritmo permite calcular sin problemas las raíces reales de casi cualquier polinomio. Los únicos fallos que se han apreciado son los causados por problemas de aproximación no solucionables.

Cálculo de raíces general

A la hora de encontrar las raíces de funciones más complejas se debe aplicar un algoritmo más genérico. Los pasos que se siguen son los siguientes:

1. Se realizan comprobaciones de formato para asegurar que la función es válida.
2. Se comprueba si se trata de un polinomio:
    a. Si es un polinomio se utiliza el método anteriormente descrito.
    b. Si no es un polinomio se comprueba si la función es derivable:
        i. Si no es derivable se intenta encontrar un intervalo con una raíz (bracketing) para no comenzar el método iterativo sin una estimación cercana (si no se encuentra un intervalo se escogerá un punto al azar). Acto seguido se calcula la raíz con el método de Steffensen.
        ii. Si es derivable se calcula su derivada y se sigue la misma estrategia que en el caso anterior, pero culminando con el método de Newton.

Representación gráfica

Los métodos de representación gráfica son los que se utilizan para obtener una imagen que represente una función en un intervalo dado. Las opciones que se presentan son la representación común (función *plot*) y la representación por colores (función *colorPlot*). Dado que la representación por colores comparte la mayoría de los detalles, solo de describe la primera. Los pasos que se realizan son los siguientes:

1. Se comprueba el formato de la función y se interpretan los parámetros opcionales si los hubiera.
2. Se instancian los datos base.
3. Para n en el intervalo [0, ancho de la imagen):
    a. Se obtiene la x discretizada en [*xmin*, *xmax*] para el valor de *n*.
    b. Se calcula la y correspondiente usando la función y se discretiza para calcular la altura en píxeles correspondiente.
    c. Si *n* > 0
        i. Si el punto anterior (*yPrev*) y el actual (*yCurr*) están en el intervalo [*ymin*, *ymax*]:
            1. Se dibuja una línea de (*n* – 1, *yPrev*) a (*n*, *yCurr*).
        ii. Si uno de los dos está fuera del intervalo:
            1. Se dibuja la línea hasta donde sea posible sin salirse de los bordes. Para esto se sustituyen valores en la ecuación de la recta.
4. Si el flag *axis* está activado:
    a. Se obtienen las escalas de los ejes de coordenadas:
        i. $scaleX = 10^{\left\lfloor \log_{10} \frac{\max(|xmin|,|xmax|)}{3} \right\rfloor}$
        ii. $scaleY = 10^{\left\lfloor \log_{10} \frac{\max(|ymin|,|ymax|)}{3} \right\rfloor}$
    b. Se dibujan los ejes con las divisiones marcadas por *scaleX* y *scaleY*

Resolución de lógica proposicional

Esta librería permite transformar cualquier expresión lógica a su forma normal conjuntiva y disyuntiva, así como comprobar si se trata o no de una tautología o una contradicción o comprobar si es satisfactible.

La base del algoritmo que permite hacer esto es un árbol semántico simulado con una lista encadenada bidimensional. Esta representa las diferentes ramas finales del árbol semántico, suponiendo que los nodos superiores no contienen información relevante. Siguiendo ciertas reglas de expansión se puede obtener un árbol completo de cualquier expresión y, por lo tanto, su forma normal disyuntiva. Para obtener la conjuntiva tan solo es necesario negar la expresión y aplicar las reglas de De Morgan.

Para comprobar si una expresión es satisfactible, si es una tautología o si es una contradicción tan solo se crea el árbol semántico y se realiza una búsqueda de patrones dependiendo del caso a analizar.

También se permite simplificar expresiones booleanas, pero eso depende principalmente de otro algoritmo similar al de simplificación general. Dicho esto, es necesario mencionar que se comprueba la validez de las expresiones a simplificar para hacer simplificaciones más profundas y complejas.

Resolución de sistemas de ecuaciones

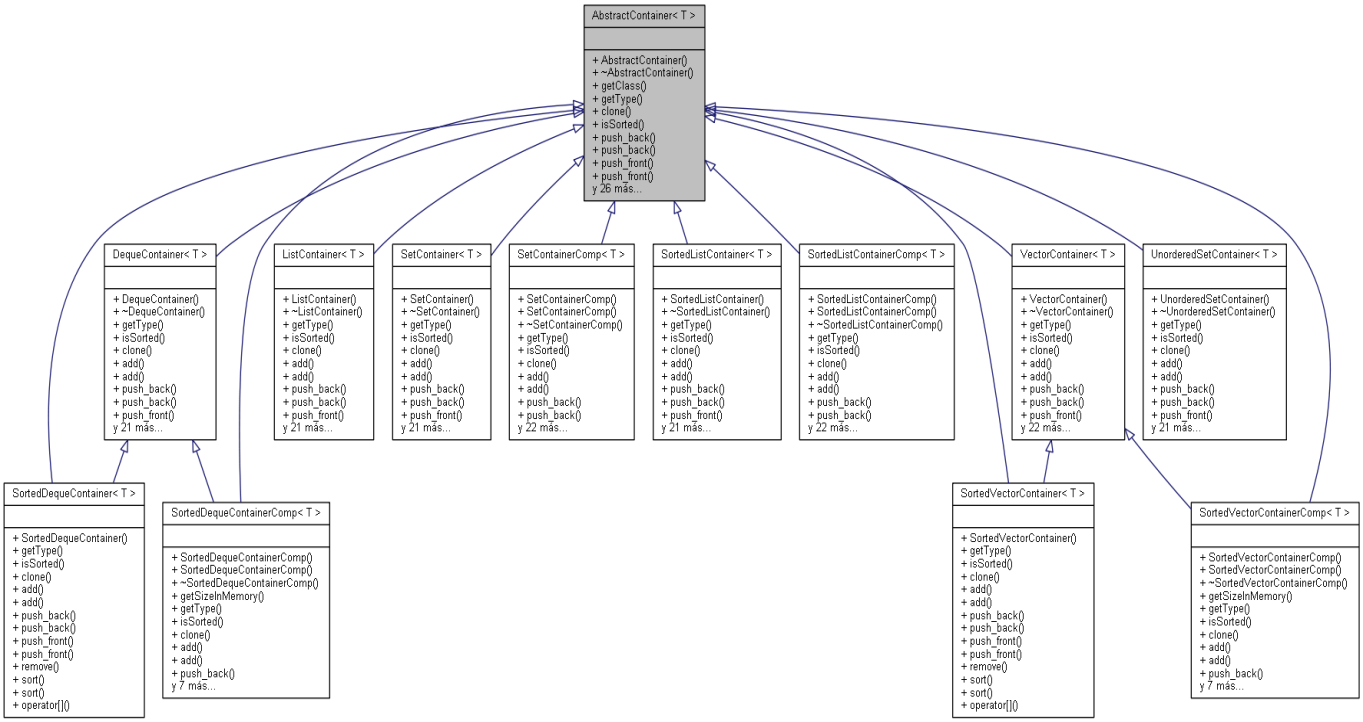
Otra de las funcionalidades que esta librería ofrece es la posibilidad de resolver sistemas de ecuaciones. El algoritmo para resolver un sistema es sencillo, tan solo realiza los siguientes pasos:

1. Se comprueba el formato de todas las expresiones proporcionadas para verificar que se trata de ecuaciones.
2. Se simplifican todas las expresiones simbólicas pasadas como parámetro.
3. Se comprueba si existe alguna expresión en el conjunto que no sea un polinomio, independientemente del grado o número de variables:
    a. Si hay alguna que no lo sea, se va al caso de resolución de sistemas no polinómicos (método de Newton generalizado para sistemas de ecuaciones).
    b. Si no la hay, se comprueba el grado máximo del sistema:
        i. Si se trata de un sistema lineal, se hace un algoritmo de conversión a matriz y se resuelve.
        ii. Si se trata de un sistema polinómico no lineal se utiliza el método de Newton generalizado para sistemas de ecuaciones.

Container

Estructura básica

Esta clase basa su estructura en el uso interno de un puntero a una clase polimófica llamada *AbstractContainer*, la cual tiene una jerarquía de clases que permite la abstracción total del tipo de almacenamiento. Su diagrama de herencias es el siguiente:



Es posible ver en el diagrama que hay varias clases con nombres intuitivos, pero también hay algunas que terminan en “Comp”. Estos subtipos son los que almacenan elementos ordenados según el criterio de un Comparator. Tienen una estructura diferente para mejorar el rendimiento.

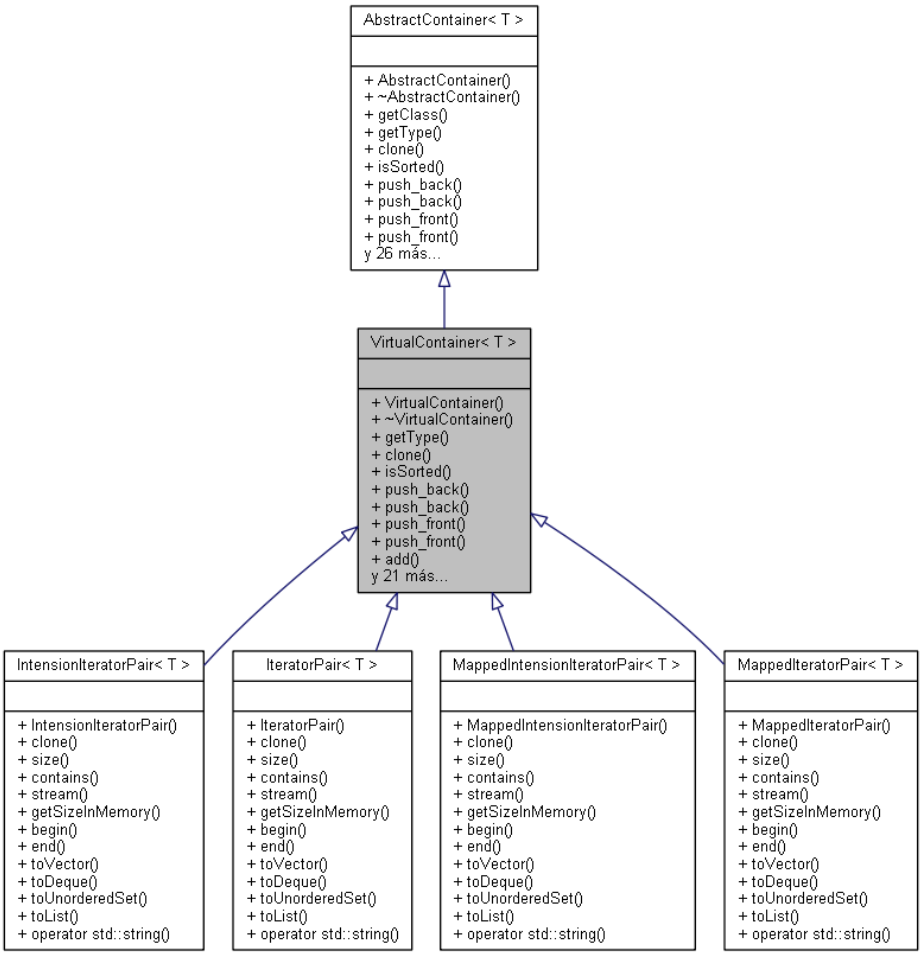
Es importante mencionar que la mayoría de subclases de *AbstractContainer* tienen como base a Containers STL del estándar de C++. Esto hace que, hasta cierto punto, la implementación de estos en Ulan dependa de este lenguaje y los cambios que se realicen sobre la estructura del mismo.

Una de los tipos que no dependen de STL es Set, ya que depende únicamente de una implementación propia de un árbol AVL llamada *AVLTree*. Explicar la estructura y funcionamiento de esta cae fuera del contexto de este documento.

AbstractContainer tiene numerosos métodos virtuales que solo pueden ser ejecutados por ciertos tipos de Containers, por lo que su mal uso vendrá acompañado de una *ContainerAccessException*. Algunos ejemplos de estas operaciones prohibidas son la búsqueda por índice en conjuntos no ordenados o el uso de la función *sort* en Containers ordenados.

Containers virtuales

El gráfico anteriormente mostrado está incompleto, ya que no representa a uno de los tipos de Container más importantes: los Containers virtuales. Una manera de ver estos objetos es como pares de iteradores con diferentes funciones de avance y comprobación de igualdad. La función principal de estos objetos es permitir la iteración sobre grandes volúmenes de datos sin necesidad de almacenarlos previamente. Para completar el gráfico anterior se debe combinar con el siguiente:

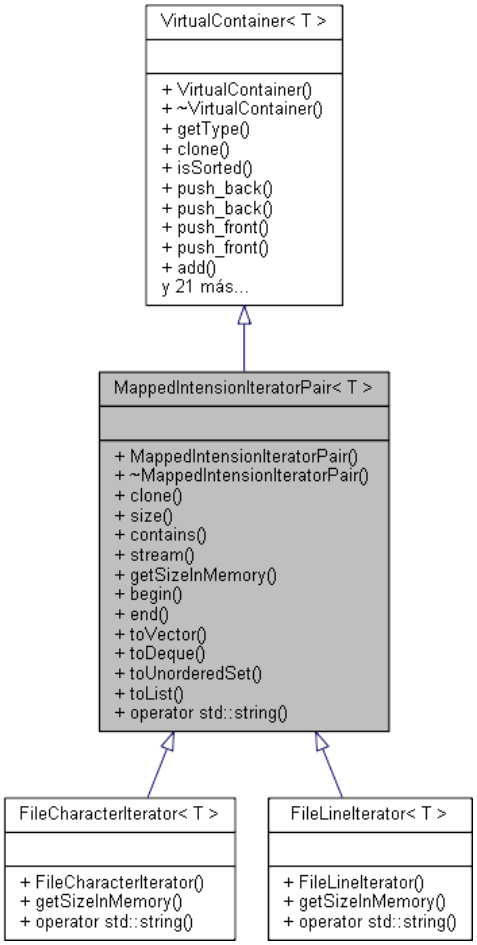


Se pueden identificar cuatro tipos de Containers virtuales:

- **IteratorPair:** par de iteradores con función de avance. Es lo que devuelve la función range.
- **MappedIteratorPair:** igual que el anterior, pero el objeto devuelto es transformado siguiendo una función.
- **IntensionIteratorPair:** itera desde un punto de inicio hasta que se cumpla un criterio especificado en una función.
- **MappedIntensionIteratorPair:** igual que el anterior, pero el objeto devuelto es transformado siguiendo una función.

Iteradores de archivos

Como se habrá podido observar, el sistema no provee las primitivas necesarias para iterar sobre el contenido de archivos, pero se ofrecen las funciones *lines* y *characters*. El funcionamiento de estas funciones se basa en dos estructuras auxiliares que heredan del Container virtual de tipo *MappedIntensionIteratorPair*:



El funcionamiento de estas clases es sencillo, basan su funcionamiento en iteradores mapeados con datos “falsos”. Internamente transforman un dato de tipo *None* (objeto vacío) en una línea o carácter del archivo a leer, pero no es más que una especie de *hack* para utilizar Containers virtuales con archivos. Esto funciona porque la función de mapeo actúa sobre unos atributos privados de las clases *FileCharacterIterator* y *FileLineIterator* que permiten mantener el control sobre el flujo de datos.

El control de *EOF* (*End Of File*) es realizado en la función anónima de comprobación de iteración final, que a su vez es el sitio donde el iterador es reiniciado. Esto se debe a que la otra opción es lanzar un *FileError* y obligar al programador a reinstanciar el iterador, lo cual no es aconsejable, dado que plantea serios inconvenientes para algoritmos que necesiten varias pasadas.

Matrix

Estructura básica

Esta clase es completamente dependiente de la clase básica Number, ya que la implementación de esta clase consiste en un array bidimensional de instancias de este tipo. Este es controlado de manera transparente por los constructores y destructores utilizando una gestión directa de memoria, no se utilizan wrappers ni containers STL.

Debido a esta decisión de hacer una gestión de memoria manual es necesario almacenar dentro de la clase el número de filas y de columnas, por lo que la clase final contiene tres atributos. La representación de estos sería tal que así:

```
Number** rows;
unsigned int nOfRows;
unsigned int nOfCols;
```

Operaciones básicas

Se implementan las tres transformaciones elementales aplicables a matrices sobre filas y columnas (permutación, escalado y suma con escalado), así como la suma de matrices, multiplicación de matrices y escalado mediante el operador de multiplicación y de división. La única operación relevante de estas es la multiplicación, ya que las demás están implementadas de manera óptima (sin utilizar varios hilos).

Se implementan tres algoritmos de multiplicación que son multiplexados dependiendo de ciertas condiciones:

- **Multiplicación larga** $\boldsymbol{O(n^3)}$**:** multiplicación estándar de matrices.
- **Multiplicación multihilos** $\boldsymbol{O\left(\frac{n^3}{k}\right)}$**:** multiplicación paralela de matrices. Este algoritmo utiliza *k* hilos que multiplican diferentes partes de la matriz a la vez para terminar en $\frac{1}{k}$ de tiempo.
- **Multiplicación de Strassen** $\boldsymbol{O\left(n^{log_2(7)}\right)}$**:** algoritmo de Strassen para multiplicación de matrices cuadradas de orden cercano a una potencia de dos.

La manera de decidir qué algoritmo se utiliza para un producto consiste en calcular un coeficiente de *cuadratura* para ambas matrices y comprobar lo cerca que están de ser una matriz de orden igual a una potencia de dos. Si tienen una cuadratura alta y un orden aceptable, se utilizará el algoritmo de Strassen, sino se utilizará un algoritmo largo o paralelo dependiendo de la disponibilidad de los núcleos del procesador que indique el gestor de concurrencia.

Algoritmos destacables

Aquí se describen los algoritmos de álgebra lineal con una estructura destacable o con una decisión de diseño a mencionar:

- **Determinante:** para el cálculo del determinante de una matriz se utiliza un algoritmo que utiliza la factorización LU. Teniendo en cuenta que esta se calcula mediante el algoritmo de Gauss, podemos afirmar que su cálculo es de orden $O(n^3)$.
- **Inversión de matrices:** para esta operación también se utiliza el algoritmo de Gauss, por lo que un razonamiento similar nos permite razonar que se utiliza un algoritmo de orden $O(n^3)$.
- **Potencia:** se utiliza un algoritmo similar al utilizado en la exponenciación de números enteros, por lo que el orden de esta operación pertenece al orden $O(M(n) \cdot \log(n))$, siendo $M(n)$ el orden de complejidad de una multiplicación.
- **Factorización QR:** se decidió que el algoritmo más adecuado para esta implementación era el que depende de las reflexiones de Householder, por lo que también se incluyen funciones que permiten calcularlas. Este algoritmo resulta ser de un orden de complejidad $O(n^3)$.
- **Autovalores:** para el cálculo de estos se utiliza el algoritmo QR. Este tan solo calcula los autovalores reales, por lo que es muy limitado. Además, está en fase inicial, por lo que no se recomienda su uso para aplicaciones serias. Se pretende hacer un algoritmo mixto más sofisticado y general.
- **Resolución de sistemas de ecuaciones:** se utiliza el algoritmo de eliminación de Gauss-Jordan para resolver sistemas representados como matrices, por lo que se puede afirmar que este algoritmo es de orden $O(n^3)$.
- **Matrices especiales:** se incluyen algoritmos para calcular las matrices de Jacobi y de Gauss-Seidel asociadas a las matrices de un sistema de ecuaciones. Estas son calculadas con la factorización DEF, la cual es de orden $O(n^2)$. Al no tener demasiadas operaciones destacables, el cálculo de estas matrices hereda el orden de complejidad de la factorización.

Comparator

Concepto

El objetivo de este objeto es crear una estructura que sea capaz de almacenar criterios dinámicos de comparación de objetos. Esto genera un problema evidente para el lenguaje utilizado para implementar Ulan, ya que los punteros a función nativos no pueden ser inicializados como una lambda-expresión con captura dinámica. Esto es solventado utilizando una estructura del estándar llamada *function*.

Se pretende que esta librería no solo permita la implementación de criterios simples, sino de criterios encadenados para los casos de empate o incluso criterios que dependan de otros objetos de este tipo. Asimismo, debería ser capaz de hacer operaciones con estos criterios de comparación (como la inversión parcial o total).

Un objeto muy similar al que se implementa aquí es la clase con el mismo nombre del lenguaje Java. Esta cumple la mayoría de los criterios pedidos para esta implementación.

Estructura básica

La implementación de la versión final depende de una lista de criterios representados como funciones binarias que devuelven un entero con signo y una lista de variables booleanas que indican si el criterio n-ésimo está invertido.

La lógica detrás de las funciones binarias depende de un wrapper genérico implementado utilizando metaprogramación con plantillas que dados dos objetos cualquiera del mismo tipo devuelve un número que indica si el primer objeto es mayor (devuelve 1), menor (devuelve -1) o igual (devuelve 0) que el segundo. Una llamada a la función binaria almacenada en la clase llamaría a esta función genérica tras convertir los operandos mediante una función adaptadora especificada en el criterio.

A la hora de calcular el resultado de una comparación se recorrerán los criterios en orden y se devolverá el resultado del primero cuya comparación no devuelva 0, siendo este resultado multiplicado por -1 en el caso de que la inversión de criterios esté activada para ese índice en particular.

Teniendo en cuenta que la implementación de la clase depende de un parámetro de plantilla T, la representación de los atributos como tipos básicos que se puede ver en la implementación de la clase es la siguiente:

```
std::vector<std::function<int(const T&, const T&)>> criteria;
std::vector<bool> inversion;
```

Image

Concepto

Esta clase, como su propio nombre indica, representa una imagen cuadrada con colores de 24bits (8 bits por canal). Los canales tienen el orden común (rojo, verde y azul respectivamente), pese a que en la matriz interna deben ser introducidos en otro orden.

Es necesario mencionar varias decisiones que se tomaron a la hora de elegir el formato exacto del archivo:

- A la hora de guardarlas como un archivo persistente en disco se utiliza la cabecera *Windows BITMAPINFOHEADER*, que es la más común por razones de compatibilidad, aunque no es la más completa. Sobre esta cabecera es necesario mencionar algunos aspectos:
  - Se ignora el campo *image size*.
  - Se ignoran los campos de resolución por metro.
  - Se especifica 0 como compresión.
  - No se especifica el campo *important colors*.
  - Se usa la paleta de colores por defecto.
- No se comprime la imagen al guardarla en un archivo, aunque es posible implementar una compresión PNG cambiando el campo *compression* en la cabecera DIB.

Estructura básica

La estructura de esta clase es sencilla, tan solo almacena un array bidimensional de píxeles, siendo cada pixel un array de tamaño 3 con las coordenadas RGB del color en esa posición. Se considera que la coordenada (0, 0) hace referencia a la esquina inferior izquierda de la imagen en cuestión, siendo el alcance válido hasta (ancho – 1, alto - 1).

La representación como tipos nativos es la siguiente:

```
unsigned char*** pixels; //{R, G, B}

unsigned int height;
unsigned int width;
```

Algoritmos de dibujo

Los algoritmos utilizados para dibujar las figuras básicas que se muestran en la lista de funciones se basan en el algoritmo de dibujado de líneas de Bresenham, con la excepción del círculo y el arco, que se basan en el algoritmo del punto medio.

Ninguno de los algoritmos utiliza *antialiasing* porque no parecía algo esencial. Además, los experimentos realizados con modificaciones sobre las funciones actuales no dieron resultados mucho mejores a los del algoritmo original.

Functional

Concepto

El lenguaje de desarrollo incluye todas las estructuras necesarias para almacenar lambda-expresiones con cualquier tipo de entrada y salida, pero esta solución no es suficiente para hacer una clase que almacene funciones de manera correcta. Esto se debe a que el sistema de tipos de Ulan está construido a un nivel superior, por lo que es necesario hacer comprobaciones de tipos de manera dinámica y establecer un modelo de función genérica para poder almacenarla de manera eficiente y sin limitaciones.

Estructura básica

El primer problema que surge al plantear el diseño de la clase es la estructura de una lambda-expresión genérica que permita hacer cualquier operación, independientemente de los tipos de llamada o salida. Para esto se utiliza el mecanismo de abstracción de tipos que se ha diseñado para C++. Dada la clase *Object*, se dice que una lambda expresión genérica tiene la siguiente forma dentro de la clase:

```
std::function<
Object(const std::vector<Object>&,   //Argumentos de llamada
       const std::vector<std::pair<std::string, Object>>&) //Captura
>
```

Después de resolver esto, se puede ver que es necesario almacenar los tipos de los argumentos aceptados por la función. Estos deben poder ser serializados o cualquier tipo de referencia, así como tipos genéricos. Para poder solucionar esto de una manera sencilla se decidió separar el problema y hacer una estructura que almacenase argumentos que se reutilizaría en las funciones comunes. Esta estructura se llama *Parameters* y tiene los métodos internos necesarios para hacer comprobaciones dinámicas de tipos.

Esta clase auxiliar basa su estructura en un array dinámico de tuplas que almacenan el nombre de los argumentos y si están serializados. Lo realmente interesante son los algoritmos de comprobación de argumentos, pero no son referentes a este apartado.

Tras hacer esto, tan solo queda añadir un atributo interno que almacene el tipo de retorno. Dado que una expresión funcional tiene casi la misma semántica que una clase, esta podría no tener un tipo de retorno asignado, por lo que se utiliza otra estructura auxiliar llamada *ReturnType*, la cual tiene los métodos internos necesarios para manejar de manera sencilla los argumentos tipados y no tipados. Igualmente, es necesario incluir una lista de objetos que actúen como datos estáticos (la captura de la lambda-expresión).

Dicho todo esto, la representación como tipos nativos de la clase es la siguiente:

```
std::vector<std::pair<std::string, Object>> capture;
Parameters argTypes; //Tipos de los argumentos
InternalFunction lambda;
ReturnType returnType;
```

Estructuras del lenguaje

Clases

Para implementar de manera efectiva la definición de clases, se dividió su manejo en dos partes: el esquema de clase y las instancias. A continuación se explica el funcionamiento básico de ambas entidades.

Esquema de clase

Se define un esquema de clase como el objeto que contiene toda la información básica referida al propio tipo que representa. Estas características son las siguientes:

- Un identificador interno único representado con un entero sin signo.
- Un nombre que identifique a la clase y que siga el formato de cadenas de elementos contextuales.
- Una lista de atributos con sus nombres y respectivos tipos.
- Un atributo lógico que indique si la clase posee función de hash.

Estos esquemas son representados en el código por la clase *InternalClass*, teniendo esta la siguiente representación en términos de tipos nativos:

```
unsigned int id;

std::string name;
std::vector<std::pair<std::string, ReturnType>> members;
bool hashable;
```

Instancias de clase

Definimos este artefacto como un Objeto Ulan cualquiera perteneciente a una clase definida por el usuario. Se utiliza este contexto para poder hacer una clase genérica que pueda contener cualquier número de atributos de cualquier tipo pese a pertenecer a la misma clase nativa. Este objeto debe contener una lista de atributos de tipo *Object* y un identificador que apunte a la clase a la que pertenece. La clase que modela este artefacto se llama *CustomClass* y sigue de manera común la estructura de abstracción de tipos.

A la hora de definir una instancia de este tipo se debe tener en cuenta que los atributos que se encuentran en su lista interna deben tener tipos que coincidan con los almacenados en el esquema de la clase, pero esto es resuelto de manera automática por la función constructora. Su representación como tipos nativos es la siguiente:

```
unsigned int id;
std::vector<Object> attributes;
```

Operadores

Tal y como se mencionó en su correspondiente sección, los operadores tienen tres atributos internos que los caracterizan y definen, pero no son los únicos que son necesarios para almacenarlos en una clase. Los atributos necesarios son los siguientes:

- Un identificador interno único representado con un entero sin signo.
- Una representación como cadena que siga el formato de operadores.
- Un número entero sin signo que represente la precedencia.
- Un enumerado que represente el agrupamiento. En este caso se llama *OpGroup*.
- Un enumerado que represente la posición y aridad del operador. En este caso se llama *OpPos*. Se incluye un valor especial para los operadores con funcionamiento no estándar (el de índice y el de cálculo).

Dicho esto, la clase que representa a este artefacto se llama *Operator* y tiene los siguientes atributos:

```
unsigned int id;

OpPos pos;
OpGroup group;
unsigned int precedence;
std::string representation;
```

Funciones

A la hora de implementar las funciones se siguió una estrategia similar que con las clases y se separó lo que se denomina el esquema de la función de sus sobrecargas. Dado que estas últimas no tienen una estructura explícita, se explica su funcionamiento más adelante.

En lo referente a los esquemas de función, podemos decir que son definidos por los siguientes atributos:

- Un identificador interno único representado con un entero sin signo.
- Un nombre que identifique a la función y que siga el formato de elementos contextuales.

La clase que modela este artefacto se llama *Function* y tiene los siguientes atributos:

```
unsigned int id;

std::string name;
```

La distinción entre funciones externas y funciones miembro no existe en los esquemas, sino que se decide dependiendo de qué haya almacenado en los repositorios de sobrecargas (explicado más adelante).

Variables

Estas son un caso especial de estructura interna, ya que podríamos decir que es una estructura parcial que depende de su almacenamiento para funcionar como su concepto semántico especifica. Dado este hecho, se describirán únicamente las funcionalidades que esta clase parcial proporcione.

El concepto que modela la clase *Variable* en el código es el de una especie de contenedor de objetos opcionalmente tipado. Los dos únicos atributos que tiene son un objeto interno y un tipo. Su representación en el código es la siguiente:

```
Object data;
ReturnType type;
```

El papel de esta estructura es, principalmente, ejecutar las asignaciones correctamente de manera dinámica. Para eso se implementa un algoritmo similar al de las operaciones unarias que hace un binding cuando es necesario y hace las conversiones implícitas pertinentes. Esto es porque, como se mencionó en la sección anterior, los operadores de asignación trabajan a un nivel superior al resto de operadores.

Constantes

Pese a que no tienen una relevancia excesiva, es necesario mencionar que se incluyen estructuras para almacenar constantes. Estas constantes son utilizadas en expresiones algebraicas, pero no tienen mayor relevancia en el resto del lenguaje.

Se almacenan en una lista global parecida a las de otras estructuras como operadores o clases, pero se hace únicamente para que sea extensible de manera manual en futuras mejoras de las librerías básicas.

Actualmente, la lista global tan solo contiene dos constantes: el número de Euler y pi. Ambas contienen la función interna que los calcula con precisión arbitraria y son utilizadas activamente durante operaciones complejas en la clase Number.

Excepciones

Las excepciones utilizadas de manera interna en Ulan no son más que errores de ejecución básicos que heredan de la clase *std:: runtime_error*, siendo estas a su vez capturadas por una clase llamada *UlanException*, la cual tiene atributos para saber la traza, información extra y recomendaciones. Todas siguen el siguiente patrón al ser definidas:

```
class ExceptionName : public std::runtime_error {
public:
    ExceptionName(std::string msg) : std::runtime_error(msg) {};
};
```

USDL

Concepto y diseño

La idea básica tras el diseño de USDL es poder definir de manera sencilla y concreta la sintaxis y la semántica básica de un concepto. Esto también se puede hacer mediante el uso de expresiones regulares o la notación de Backus-Naur, pero estas no terminan de tener el poder expresivo para dar a entender cuáles son las partes relevantes de una sintaxis (llamadas argumentos en USDL) ni cómo deberían interpretarse.

Se intenta resolver este problema haciendo un lenguaje con una sintaxis altamente inspirada en variaciones de BNF. Este permite especificar qué partes de una sintaxis son "relleno" y cuáles son los fragmentos que es necesario analizar de alguna manera especial. La diferencia es claramente apreciable cuando intentamos definir la sintaxis de una tupla numérica con diferentes lenguajes:

$$\boldsymbol{Regex}: \text{^\(\d\{1,\}(.\d+)? (,\d\{1,\}(.\d+)?)*\)\$}$$

$$\boldsymbol{BNF}: \text{"("} \ < Number > \{, < Number >\} \text{")"}$$

$$\boldsymbol{USDL}: '(' \ Arg(< Number >, Number, nums) \ \{',' \ Arg(< Number >, Number, nums)\} \ ')'$$

Si bien es cierto que la línea en USDL es la más larga, en ella se especifica en qué parte de la cadena de caracteres están ubicados los números de la tupla, que deber ser interpretados como una clase llamada Number (La clase numérica de Ulan) y se le ha dado el nombre $nums$ al argumento para que se defina automáticamente como una variable dentro del cuerpo de la función que hace de constructor. Es importante destacar que la expresión regular es muy confusa independientemente de la similar semántica, por lo que esta sintaxis ha sido completamente descartada a la hora de diseñar USDL.

Otro detalle importante que es necesario mencionar es que muchas implementaciones de expresiones regulares o similares tienen una complejidad exponencial por culpa de mantener una expresividad con referencias. Para evitar esto, se ha preferido utilizar una implementación sin estas (a riesgo de tener un poder expresivo menor) para poder garantizar una ejecución más eficiente.

En definitiva, este lenguaje fue creado para compenetrarse con Ulan y poder alcanzar un nivel de expresividad superior a prácticamente cualquier lenguaje existente con una curva de aprendizaje lo más llevadera posible.

Estructura básica

Opciones de representación

Es posible imaginar diferentes estructuras que puedan almacenar un patrón USDL, pero este abanico es reducido considerablemente si tenemos en cuenta que es necesario poder aplicar algoritmos de reconocimiento para detectar patrones incorrectos. Cuando estudiamos los diferentes algoritmos de ejecución posibles y apuntamos a la complejidad mínima nos damos cuenta de que no existe una única estructura que permita hacer todo esto, por lo que es necesaria una solución mixta.

Representación básica

La representación básica está implementada como un árbol parecido al de la clase Symbolic, utilizando un enumerado para especificar el tipo del nodo y los datos que contiene. Los tipos de nodo representados en esta estructura se corresponden con los diferentes patrones y marcadores USDL que existen.

Asimismo, cada tipo de nodo tiene una correspondencia de datos internos que está especificada en la siguiente tabla:

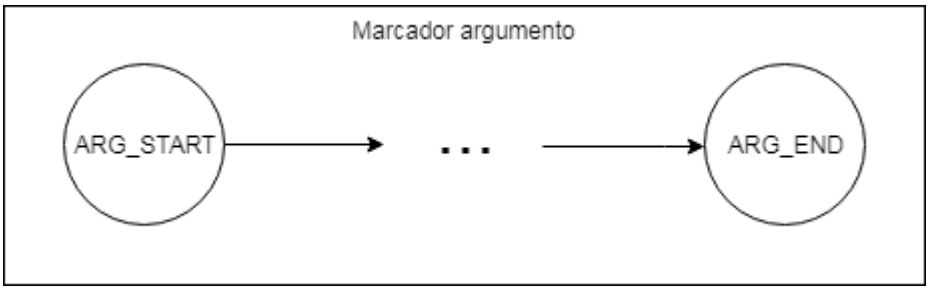
Nodo	Tipo asociado	Descripción
Composición	`std::vector<PatternNode>`	Se almacenan los nodos hijo del nodo actual. Se interpretan como los patrones escritos de izquierda a derecha
Texto	`std::string`	Se almacena el texto interno
Texto difuso	`std::pair <std::string, Number>`	Se almacena el texto interno y el número asociado al patrón
Opcional	`PatternNode`	Se almacena el patrón interno
Repetición	`std::tuple <Number, PatternNode, Number>`	Se almacenan el número de iteraciones mínimas, el patrón a repetir y el número de iteraciones máximas
Alternativa	`std::vector<PatternNode>`	Se almacenan los nodos hijo del nodo actual. Se interpretan como los patrones escritos de izquierda a derecha
Símbolo	`std::string`	Se almacena una cadena de caracteres con los símbolos del patrón
Rango	`std::pair<char, char>`	Se almacenan los dos caracteres del rango
Clase	`uint32_t`	Se almacena un identificador interno de la clase
Argumento	`std::tuple <PatternNode, ParsingIdentifier, std::string>`	Se almacena el patrón interno, una estructura auxiliar que permite especificar identificadores USDL y el nombre del argumento

Representación para ejecución

A la hora de ejecutar el algoritmo de reconocimiento es necesario utilizar una estructura que permita almacenar una máquina de estados, ya que es necesaria una compilación a NFA para poder ejecutar un algoritmo de orden de complejidad lineal. Esto es porque el algoritmo de ejecución elegido está claramente inspirado en las expresiones regulares de Thompson.

La estructura utilizada es similar a la anterior, pero los nodos forman un digrafo en los que cada uno de los nodos representa un estado diferente. Los datos almacenados en cada uno de los nodos finales son los mismos que en el árbol básico, pero con la diferencia de que el nodo de argumento es dividido en dos partes y que se añaden dos nodos de control nuevos. Se comenzará explicando la nueva semántica y funcionamiento del nodo argumento:

Este nodo ha sido dividido en Inicio y final de argumento, siendo estos los delimitadores de un patrón interno:



El nodo que contiene la información de parseo es el de inicio, el propio algoritmo de ejecución está diseñado para poder detectar automáticamente a qué argumento corresponde el nodo final, ayudándose de que no se permiten argumentos anidados.

En lo que respecta a los nuevos nodos se han introducido dos nodos auxiliares que delimitan el grafo que representa la máquina de estados. Estos nodos permiten que cualquier digrafo sea, como mínimo, débilmente conexo y permita un algoritmo de reconocimiento más sencillo. La lógica que sigue es exactamente la misma que la de la división del nodo anterior, ya que cualquier patrón debe empezar por el nodo de comienzo y debe terminar en el de fin.

Si no se tuvieran estos nodos habría problemas con los patrones que consisten en un patrón alternativa puro, ya que la compilación de este implica crear varios nodos y redirigir las referencias de la máquina de estados. Esto resultaría en un digrafo con varias componentes conexas y la estructura debería ser lo suficientemente inteligente como para almacenar nodos de inicio múltiples, y eso no es deseable existiendo esta solución.

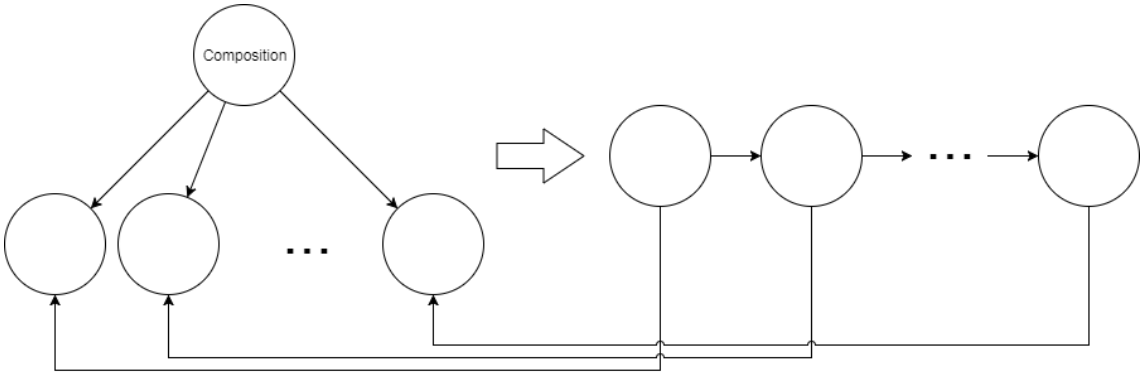
Es necesario mencionar que esta representación es obtenida a partir de la representación como árbol mencionada anteriormente. El algoritmo de compilación se especificará más adelante, ya que tiene una complejidad a tener en cuenta.

Algoritmo de compilación

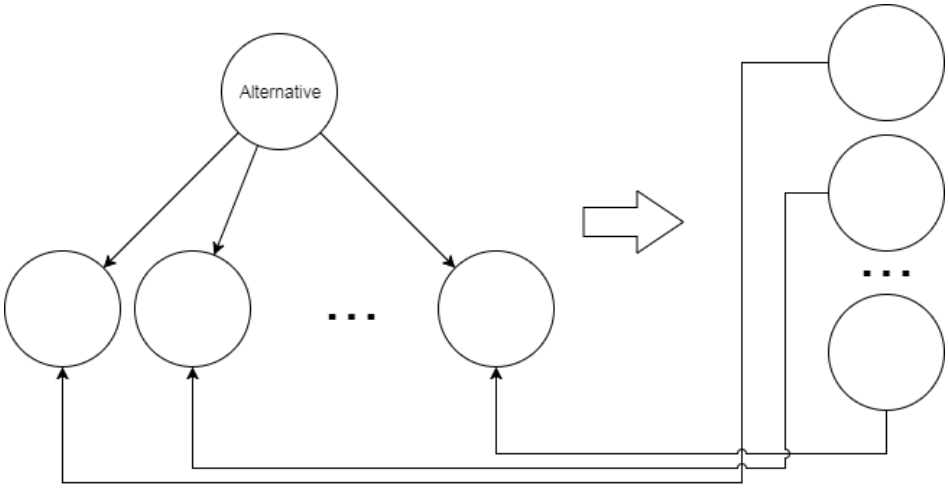
La idea principal del algoritmo de compilación consiste en utilizar nodos intermedios de tipo indeterminado para indicar grafos genéricos que deben ser extendidos por una función auxiliar. Se considera que cualquier nodo que no sea de tipo indeterminado tiene su representación final y solo debe tener sus salidas alteradas.

Todos los nodos indeterminados tienen como dato en su representación un puntero a un nodo del árbol de representación básica. Dependiendo del tipo de nodo al que apunte el algoritmo hará una de las siguientes operaciones:

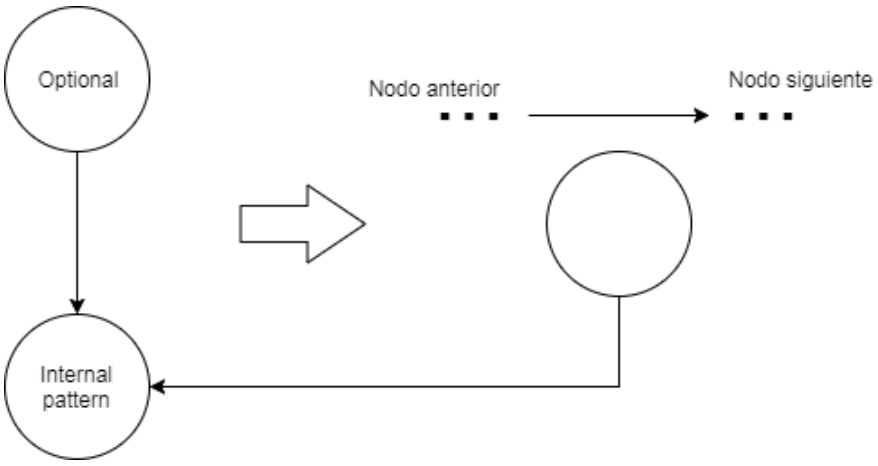
- **Composición:** se crean tantos nodos intermedios como hijos tenga el árbol de representación básica. Cada nodo nuevo n de la máquina de estados apuntará al nodo hijo n-ésimo del árbol básico. Se definen las salidas del último nodo como las salidas del nodo transformado y la de los demás como una sucesión.



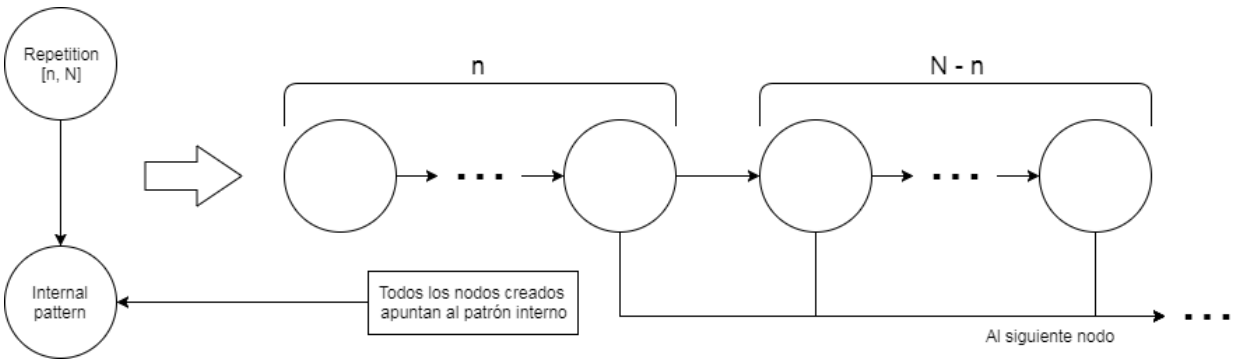
- **Alternativa:** se crean tantos nodos intermedios como hijos tenga el árbol de representación básica. Todos los nodos de la máquina de estados actual que apuntaban al nodo transformado ahora apuntarán a todos los nodos nuevos. Cada nodo nuevo n de la máquina de estados apuntará al nodo hijo n-ésimo del árbol básico. Se definen las salidas de cada nodo nuevo como las salidas del nodo transformado. No hay aristas entre los nodos recién creados.



- **Opcional:** se añaden todas las salidas del nodo que se está transformando a cada nodo de la máquina de estados actual lo contenga en sus aristas. Se actualiza el nodo del árbol básico al que referencia para que sea el patrón interno.



- **Repetición:** dados un número de iteraciones mínimas n y un número de iteraciones máximas N, se interpreta este patrón como una composición de n elementos que siguen el patrón interno seguidos de $N - n$ elementos opcionales que también lo siguen. Esto se basa en que patrones como "$2\{d\}5$" son equivalentes a "$dd\,\{d\}3$" y a "$dd\,[d]\,[d]\,[d]$". Este es el patrón con la implementación más complicada. Si no hay iteraciones máximas se creará un bucle en la máquina de estados en un único nodo opcional que también apuntará al patrón interno.



- **Argumento:** el funcionamiento de este tipo de nodos fue explicado en la sección en la que especificaba la estructura de ejecución de los patrones USDL. Consisten en una división entre inicio de argumento y fin de argumento para saber cuándo se sobrepasa el último nodo de un patrón interno y anotar la subcadena.
- **Nodos finales:** los nodos no mencionados en esta lista se consideran *nodos finales*. Estos son los que deben convertir un nodo indeterminado en un nodo de un tipo específico sobre el que no se debe iterar más el algoritmo de compilación. Contienen una copia casi exacta de los datos del árbol de representación básica.

Algoritmo de ejecución

El algoritmo de ejecución se basa en la suposición de que en la máquina resultante no hay nodos indeterminados, sino que son todos nodos finales. Esto es fácilmente demostrable, ya que en cada iteración del algoritmo de compilación se profundiza en el árbol de representación básica, por lo que acabará llegando a las hojas.

El algoritmo basa su funcionamiento en una pila que almacena elementos de una estructura auxiliar llamada *NFAState*. Esta estructura almacena un puntero a un nodo de la máquina de estados, un iterador numérico, una lista de argumentos parseados y una tupla que almacena el nodo argumento actual (si lo hay) junto a su iterador de inicio.

El algoritmo comienza con un único valor en la pila: un estado compuesto de un puntero al nodo de inicio de la máquina, un iterador inicializado a un número especificado por parámetros, una lista de argumentos parseados vacía y un argumento actual vacío. Acto seguido se comienza a iterar sobre cada nodo indeterminado en la pila hasta que esta esté vacía o se active un flag que indica que se ha terminado de ejecutar correctamente el patrón.

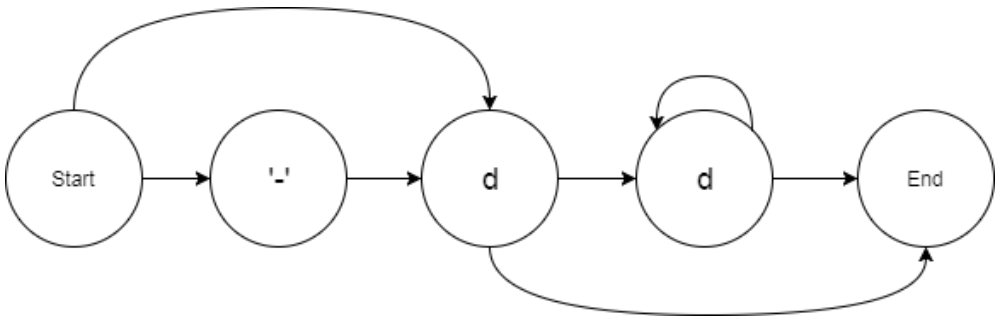
Cada una de estas iteraciones añadirá al final de la pila un estado nuevo por cada salida del nodo actual con un patrón que la cadena dada como parámetro cumpla en la posición que indique el iterador. Este se aumentará dependiendo del patrón seguido y el puntero interno será al nuevo nodo. Acto seguido, se elimina el estado anterior.

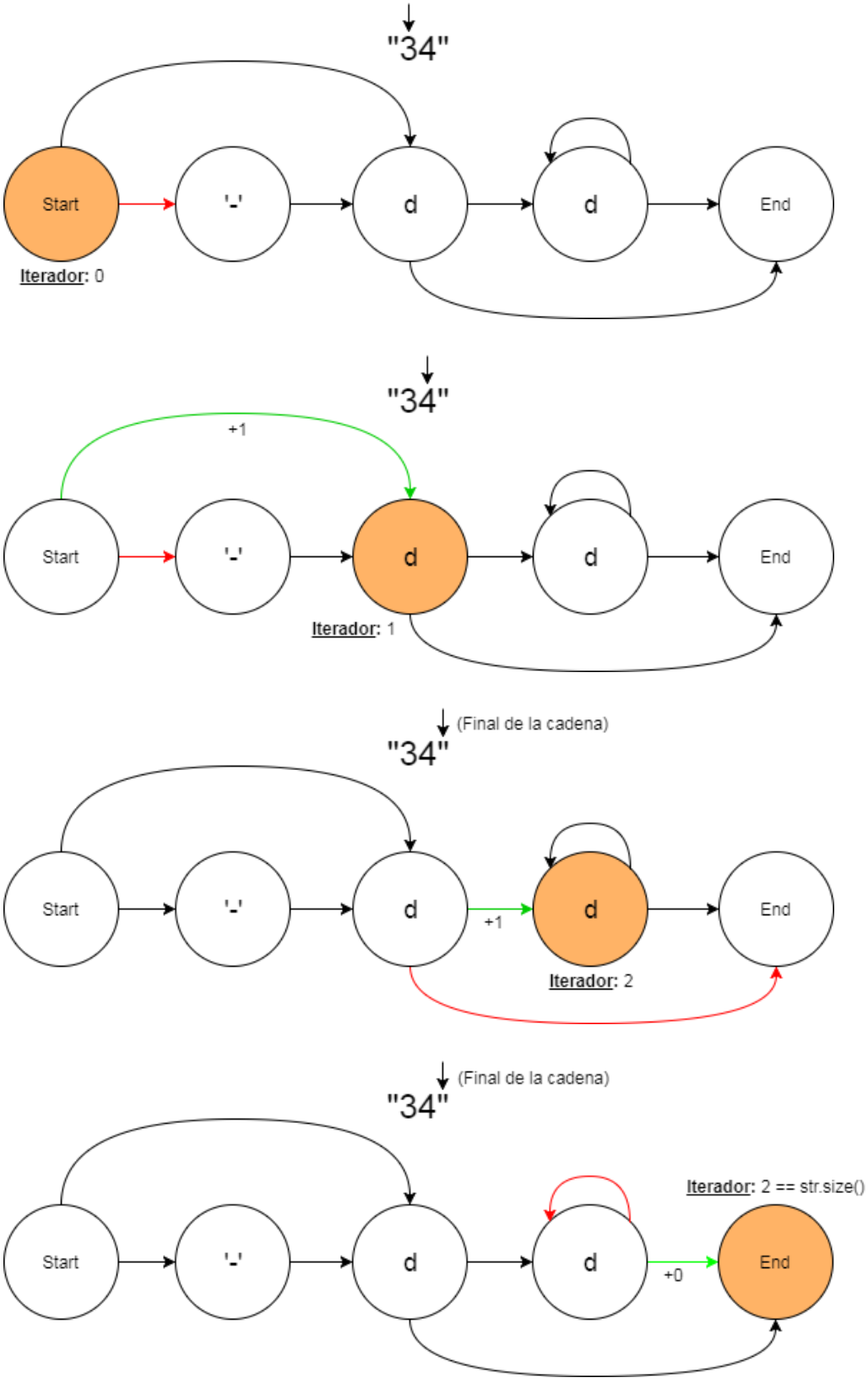
Cuando se llegue a un nodo de inicio de argumento se añadirá el estado que corresponde, pero con un argumento actual indicado por el iterador actual y los datos internos del nodo de argumento. Al llegar a un nodo cualquiera de fin de argumento se añadirá a la lista de argumentos parseados una estructura auxiliar que lo almacenará de forma eficiente.

Si se activa el flag de ejecución correcta se generará un estado final en una variable externa. Los datos internos del estado serán pasados a una estructura conveniente para su devolución al usuario.

A continuación se muestra de manera gráfica un ejemplo de ejecución exitosa para un patrón USDL que modela la sintaxis de un número entero de cualquier signo:

$$[' - '] \, 1\{d\}$$




"34"
Start
'-'
d
d
End
Iterador: 0
"34"
+1
Start
'-'
d
d
End
Iterador: 1
(Final de la cadena)
"34"
Start
'-'
d
+1
d
End
Iterador: 2
(Final de la cadena)
"34"
Iterador: 2 == str.size()
Start
'-'
d
d
+0
End

Mecanismos de abstracción

Tipos dinámicos

Estructura de objetos

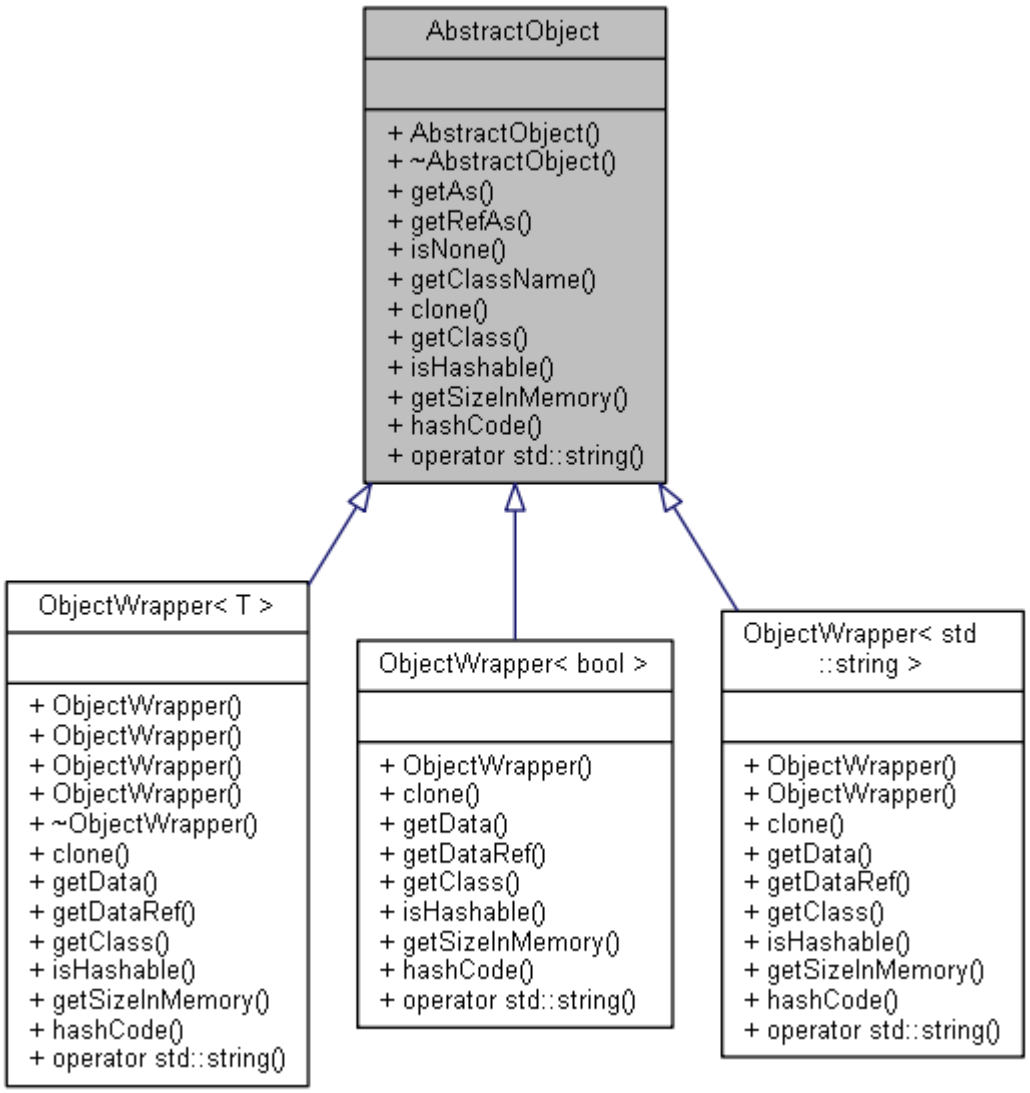
Dado el lenguaje de desarrollo, es necesario tener claro cómo se van a implementar las transformaciones básicas para que adaptar el código se convierta en una tarea más natural y se permita la traducción con más facilidad. El primer problema que se presenta es el tipado gradual de Ulan.

Para hacer que C++ permita utilizar tipos dinámicos se han utilizado técnicas de borrado de tipos mediante metaprogramación con plantillas que permiten tener un tipo base que maneja internamente el almacenamiento de cada tipo de dato usando punteros polimórficos. En este caso, se ha decidido que el nombre de esta clase sea *Object*.

Esta clase funciona como wrapper de un puntero polimórfico a una clase base llamada *AbstractObject*, la cual hereda de todas las especializaciones definidas para la clase paramétrica *ObjectWrapper*. Estas últimas tienen una relación de composición con el tipo de su parámetro y generan una interfaz automáticamente para que todos los algoritmos estándar sean compatibles con el tipo de objeto generado. Las funciones que deben definirse en cada especialización son las siguientes:

- **Constructores:** se deben definir los constructores copia y por rvalue (movimiento de datos), ya que se usan activamente.
- **Función *clone*:** esta función devuelve un puntero a una instancia de tipo *ObjectWrapper* que contiene una copia de los datos contenidos en la clase actual.
- **Getters por referencia:** se deben definir dos funciones llamadas *getData* (para devolver una referencia constante a los datos) y *getDataRef* (para devolver una referencia común a los datos).
- **Función *getClass*:** esta función devuelve un entero que actúa de identificador interno de la clase que contiene. Para los tipos básicos se devuelven valores de un enumerado predefinido que facilita la comprensión del código.
- **Función *isHashable*:** devuelve true en el caso de que la clase tenga función de hash definida.
- **Función *getSizeInMemory*:** devuelve una estimación del tamaño en bytes que ocupan los datos almacenados en la instancia de la clase.
- **Función *hashCode*:** devuelve el hash del objeto contenido en la instancia de la clase. Si el tipo no tiene función de hash se puede devolver cualquier valor.
- **Conversión a cadena de caracteres:** se debe definir la conversión implícita a la clase *std::string* para facilitar la lectura de datos. Si se trata de un tipo con datos difíciles de mostrar (por ejemplo, Comparator), se puede poner información parcial de la instancia (como el número de criterios, en el caso anterior).

Teniendo en cuenta todo esto, es necesario mencionar que existe un wrapper genérico para las clases con funcionamiento estándar, tan solo fue necesario definir especializaciones de plantilla para dos tipos nativos. El diagrama de herencias exacto sería el siguiente:



Una vez definidos los tipos básicos con sus respectivos wrappers, queda la tarea de almacenar los datos de manera segura. Esta tarea de gestión de memoria queda delegada a una clase intermedia llamada *DataBlock*, que es el tipo exacto del puntero que contendría una instancia de tipo *Object*.

Esta clase funciona de manera similar a los punteros inteligentes de tipo *shared_ptr*, los cuales hacen de la gestión de memorial algo trivial, pero con el coste de tener un rendimiento reducido. Para evitar este impacto se implementó una clase similar.

La clase *DataBlock* contiene un contador de referencias que va variando conforme se van creando objetos y referencias (el funcionamiento exacto se explica más adelante), un puntero a *AbstractObject* y un cierre de exclusión mutua que permite que el bloque sea accedido de manera paralela sin corromper el contenido. Este último se activaría en los casos que provocarían un error de segmentación.

Resolución de operaciones

Dado que las operaciones deberían poder realizarse independientemente del tipo concreto de los operandos, es necesario especificar un mecanismo que permita ejecutar cualquier operación eligiendo la función a aplicar de manera dinámica. También debe darse la posibilidad de intentar realizar una operación no definida que resulte en una excepción.

Este problema se resuelve implementando todos los operadores básicos en la clase *Object* y haciendo que este se ocupe de acceder al repositorio correcto para extraer la función a aplicar. Para esto es necesario implementar una función auxiliar para cada aridad de operadores posible que permita buscar operaciones en la lista global y permita comprobar las conversiones de manera implícita.

La versión para operadores unarios se puede obtener fácilmente a partir del algoritmo genérico para operaciones binarias, por lo que este es el único que se describe:

1. Se comprueba si la operación está definida para los tipos de los operandos dados como parámetros utilizando una función que accede a la lista de operaciones.
2. Si la operación existe se devuelve el resultado de su aplicación
3. En caso contrario:
    a. Se comprueban los casts posibles desde la clase del primer operando. Se comprueba la validez de la operación para cada uno de estas conversiones y se guarda la operación si hay solo una posible. Tener varias posibles resulta en un error por ambigüedad.
    b. Se hace lo mismo que en el paso anterior, pero con el segundo operando. Si se detectan varias operaciones posibles durante estos dos pasos, la operación resultará en un error por ambigüedad.
4. Si se devolvió un resultado sin errores se comprueba que la clase del objeto resultante es la que indica la operación. Si no lo es, acarreará un error de tipo.

Para concluir la resolución de este problema queda arreglar un problema que podría provocar un fallo de acceso a memoria: el acceso a los datos almacenados en un wrapper *Object* no es seguro, ya que tan solo se puede acceder a ellos si sabes su tipo de antemano. Una operación que espera un retorno en particular podría extraer un objeto de una clase errónea y provocar un error que cuelgue el sistema.

Para arreglar este problema se incluyen versiones de este algoritmo anterior que comprueban que el tipo del objeto devuelto (mediante la función *getClass*) concuerda con un tipo esperado pasado como parámetro. Esta función es utilizada dentro de los operadores de comparación en vez de su versión genérica, ya que estos son utilizados dentro de algoritmos de bajo nivel como ordenación o gestión interna de Containers y el lenguaje base no posee la capacidad de evitar estos errores.

Pese a que una opción para evitar estas comprobaciones podría haber sido hacer un método de obtención de datos más complejo, se decidió no hacerlo porque esto forma parte del núcleo de gestión de objetos de Ulan y el rendimiento óptimo es necesario.

Algoritmo de enlace

Los enlaces se realizan de manera interna de la siguiente manera:

- Si se pide un valor.
  - Si el objeto es un valor se devuelve una referencia a rvalue.
  - Si el objeto es una referencia de cualquier tipo se devuelve un objeto que almacene el puntero interno de la referencia para evitar copiar los datos. Esto incrementaría el contador de usos.
- Si se pide una referencia constante:
  - Si el objeto es una referencia constante se devuelve una referencia a rvalue.
  - Si el objeto es una referencia no contante se devuelve una copia del objeto transformada a referencia constante.
- Si se pide una referencia no constante se devuelve directamente una referencia a rvalue, ya que este argumento presupone que el enlace es posible y no hay más opciones.

Empaquetado de argumentos

A la hora de hacer una llamada a una función es necesario aplicar varios métodos de reconocimiento de patrones para determinar si los argumentos dados coinciden con los tipos de alguna de las sobrecargas definidas, así como es necesario aplicar casts implícitos donde sea la situación lo requiera. Estos procedimientos son realizados tanto en cálculo de funciones como en operaciones de cálculo, con ligeras variaciones en cada uno.

Lo primero que es necesario definir es una función que determine si una serie de argumentos (representados como un array dinámico de tipos *Object*) coinciden con una lista de parámetros dada, la cual resulta en un algoritmo de complejidad lineal. En el caso de que coincidan, este algoritmo también devuelve un array con el número del parámetro al que corresponde cada argumento de llamada. Este array es introducido en otra función que crea la representación final de argumentos de llamada.

La representación de argumentos de llamada final tiene ciertas propiedades:

- Se representa como un array dinámico de tipos *Object*.
- Se garantiza que cada argumento de llamada es exactamente del tipo pedido. Esto es gracias al algoritmo de enlace.
- Cualquier conjunto de argumentos que se correspondan con un parámetro de tipo serializado será a su vez representado como un Object de tipo interno Container. Este Container será un vector que conservará el orden original dado antes del empaquetado.

Al igual que en las operaciones comunes, es necesario tener en cuenta que cabe la posibilidad de que se defina una función errónea que devuelva un tipo que no concuerde con el especificado. Para evitar cuelgues, se realizan comprobaciones del tipo devuelto.

Referencias seguras

Otro problema a tener en cuenta con el lenguaje de desarrollo es que, al tener una clarísima orientación a software de bajo nivel, intentar acceder a un objeto que ya ha sido eliminado de la memoria es provocar un fallo de segmentación que hace que el programa deje de funcionar sin avisos claros. Para esto es necesario implementar un sistema de referencias que permita dar errores de ejecución sin ser muy pesado.

El sistema de referencias basa su funcionamiento en complicar un poco la abstracción de tipos haciendo que la clase *Object* apunte a un objeto de tipo *DataBlock* en vez de *AbstractObject*, siendo el primero el que gestionaría las referencias. Una instancia de tipo *DataBlock* contiene un puntero de tipo *AbstractObject* y un contador de usos.

El contador de usos se incrementa en 1 cada vez que un objeto de la clase *Object* es instanciado con su puntero interno apuntando hacia el bloque de datos al que pertenece o cuando una referencia se instancia referenciando a ese mismo bloque. Asimismo, cada vez que una instancia de esa clase se elimina, se reduce el valor almacenado en el contador en 1. Cuando en un destructor de este tipo se detecta que el contador ha bajado a 0, se borra el objeto almacenado en el *DataBlock*.

En este momento es necesario mencionar que las referencias en Ulan ocupan exactamente 16 bytes y se almacenan en una estructura auxiliar llamada *Reference*. Estas contienen un puntero a un objeto de tipo *DataBlock*, un flag que indica si es una referencia constante, un tipo para restricciones y un puntero a un cierre de exclusión mutua. Este último es opcional, pero normalmente apuntaría al cierre asociado a la variable en la que se almacena el valor referenciado (programado para futuras versiones con variables protegidas).

También es necesario comentar la posibilidad de reasignar los valores a los que apuntan las referencias cuando estas no son constantes y el nuevo valor tiene el tipo especificado en la referencia (en el caso de que haya restricción). Esta acción es la base de la reasignación de valores a variables que ya están definidas, siendo el procedimiento diferente cuando no lo están.

Ejemplo:

```
if true
   a = 3 //Se crea DataBlock para almacenar el 3 (contador = 1)

   b = a //Se incrementa el contador de usos por instanciar una referencia (contador = 2)
   c = b //Se vuelve a incrementar por copiar la referencia (contador = 3)

   c := 5 //El contador se mantiene igual, pero el bloque ahora contiene un 5 (contador = 3)

//Se cierra el contexto, por lo que se eliminan a, b y c.
//Tras esto, el contador pasa a ser 0 y se elimina el DataBlock
```

Almacenamiento de estructuras

Operadores y operaciones

En el caso de los operadores, se distingue el almacenamiento de la estructura del de las diferentes operaciones que se le atribuyen. Esto ayuda a que se puedan definir de manera rápida y a desacoplarlos de sus sobrecargas. Actualmente, la lista global de operadores está definida como un array dinámico de objetos de tipo *Operator* ordenado por representación como cadena de caracteres. Asimismo, se acompaña a esta lista con otro array dinámico que almacena punteros a *Operator*. Se asegura que este último tiene un puntero al operador con identificador interno igual a su posición.

Estas estructuras permiten los siguientes órdenes de complejidad para el acceso:

- **Mediante identificador interno**: $O(1)$ (lista auxiliar)
- **Mediante representación como cadena**: $O(\log(n))$
- **Cualquier otro criterio**: $O(n)$

En lo que respecta al almacenamiento de operaciones (sobrecargas de operadores), se definen las listas globales de la siguiente manera:

```
typedef std::list<std::pair<Parameters,
std::pair<std::function<Object(const Object&, const
std::vector<Object>&)>, ReturnType>>> CalculationOperatorRepository;
typedef Mapper<std::pair<unsigned int, unsigned int>,
std::pair<std::function<Object(const Object&, const Object&)>,
ReturnType>> BinaryOperatorRepository;
typedef Mapper<unsigned int, std::pair<std::function<Object(const
Object&)>, ReturnType>> UnaryOperatorRepository;

extern Mapper<unsigned int, CalculationOperatorRepository>
definedCalculationOperations;
extern Mapper<unsigned int, BinaryOperatorRepository>
definedBinaryOperations;
extern Mapper<unsigned int, UnaryOperatorRepository>
definedUnaryOperations;
```

Estas estructuras se basan en almacenar arrays asociativos con funciones como valores y pares de enteros que representan identificadores de clases como claves. Estos arrays son llamados repositorios y son almacenados a su vez en otros arrays asociativos con identificadores de operadores como claves. Esto permite que, conocido el operador y los tipos de los operandos, se pueda obtener la función a aplicar en tiempo constante.

Durante la comprobación de casts implícitos, estas comprobaciones son realizadas de manera activa, por lo que era importante priorizar los tiempos de acceso. Aun dado este hecho, no se utilizaron arrays dinámicos para almacenar los repositorios para evitar complicar la definición inicial y la expansión de las listas.

El caso del operador de cálculo se puede extrapolar de los ya expuestos. Tan solo consiste en una generalización de los repositorios anteriores.

Funciones y sobrecargas

Al igual que los operadores, las funciones están completamente desacopladas de sus sobrecargas, por lo que su almacenamiento es completamente diferente. Al igual que en el caso anterior, la lista de funciones está definida como un array dinámico de objetos de tipo *Function* ordenado por representación como cadena de caracteres. Siguiendo el mismo patrón, se acompaña esta lista de otra auxiliar que almacena punteros a las funciones con identificador interno igual a su posición.

Estas estructuras permiten los siguientes órdenes de complejidad para el acceso:

- **Mediante identificador interno:** $O(1)$ (lista auxiliar)
- **Mediante representación como cadena:** $O(\log(n))$
- **Cualquier otro criterio:** $O(n)$

En lo que respecta al almacenamiento de las sobrecargas de funciones, se definen las listas globales de la siguiente manera:

```
typedef std::list<std::pair<Parameters,
std::pair<std::function<Object(const std::vector<Object>&)>,
ReturnType>>> ExternFunctionRepository;
typedef Mapper<unsigned int, std::list<std::pair<Parameters,
std::tuple<std::function<Object(const Object&, const
std::vector<Object>&)>, ReturnType, bool>>>> MemberFunctionRepository;

extern Mapper<unsigned int, ExternFunctionRepository>
definedExternFunctionOverloads;
extern Mapper<unsigned int, MemberFunctionRepository>
definedMemberFunctionOverloads;
```

Se puede observar que las listas de funciones externas y de funciones miembro están separadas. Esto se debe a que las funciones miembro son almacenadas con un doble array asociativo y tienen una lambda-expresión que toma un argumento adicional que representa al objeto de llamada, así como un atributo que especifica si la función es constante. Es posible detectar el tipo de una función haciendo una comprobación rápida en ambas listas.

Dadas estas dos definiciones, se puede deducir que, conocidos el identificador interno de la función y los argumentos de llamada, se puede obtener la función de bajo nivel que modela la sobrecarga con complejidad $O\big(nArgs \cdot nFuncs + O(1)\big)$, siendo $nArgs$ el número de argumentos de llamada y $nFuncs$ el número de funciones que es necesario analizar para buscar ambigüedades (este último es variable dependiendo de si se está analizando una función miembro o una externa). En la mayoría de los casos, es prácticamente instantáneo, ya que no es común sobrecargar las funciones de sobremanera.

En lo que respecta a la adición dinámica de funciones nuevas, no debería ser un problema en la mayoría de los casos. Si no tenemos en cuenta la creación de listas internas para cada clase, las sobrecargas son de orden constante y los esquemas son de orden lineal.

Clases

Las clases también siguen el esquema de almacenamiento visto en los operadores y en las funciones, por lo que se emplean dos arrays dinámicos sincronizados para garantizar unos tiempos de acceso aceptables para su referencia constante. El primer array contiene todas las clases ordenadas por representación como cadena y el segundo contiene punteros a las clases cuyo identificador coincide con su posición.

Al igual que en estructuras similares, este almacenamiento permite los siguientes órdenes de complejidad para el acceso:

- **Mediante identificador interno:** $O(1)$ (lista auxiliar)
- **Mediante representación como cadena:** $O(\log(n))$
- **Cualquier otro criterio:** $O(n)$

En lo que respecta a las sintaxis directas de clases, están desacopladas de la propia estructura de la clase, siguiendo el patrón de las anteriores estructuras. También se almacenan como un tipo diferente a las demás sintaxis propias, ya que necesitan información adicional para ser identificadas.

La definición de la lista global es la siguiente:

```
extern Mapper<unsigned int,
std::pair<Pattern, std::function<Object(const std::string&,
unsigned int, unsigned int)>>> classSyntaxList
```

Como se puede observar, no es necesario especificar estructuras complejas, tan solo se trata de un array asociativo que asocia los identificadores internos de cada clase con los árboles de representación básicos de los patrones USDL que definen su sintaxis y con una función de parseo. El acceso a estos nodos es de tiempo constante dado el identificador interno.

Casts

Como se mencionó anteriormente, los casts no tienen una estructura interna explícita, sino que son almacenados de la siguiente manera:

```
typedef Mapper<unsigned int, std::function<Object(const Object&)>>
CastRepository;

extern Mapper<unsigned int, CastRepository> castList;
```

Se almacenan los casts en un array asociativo que mapea un tipo de origen a otro array asociativo que mapea el tipo de destino a la función a aplicar. Esta función es almacenada como una lambda-expresión dentro de un objeto *std::function*.

Variables

El método de almacenamiento de las variables es el que completa el funcionamiento correcto de esta estructura, ya que es el que permite asignarle nombres y es el que gestiona todo lo relacionado con los contextos. Este está basado en una estructura llamada *VariableList*, la cual contiene una lista encadenada de arrays asociativos de cadena a variable, siendo cada uno de estos un contexto.

Esta estructura está preparada para ser inicializada como una lista con un único valor reservado. Este valor debe ser un array asociativo vacío que representa al contexto primario cuando está vacío. A partir de ese momento se debe añadir un elemento al final de la lista cada vez que se abra un contexto y se debe eliminar el elemento del final cada vez que termine.

Siguiendo la lógica de la mayoría de lenguajes, el algoritmo de obtención de variables le da prioridad a los últimos contextos, ya que las redefiniciones en contextos inferiores (últimos) sobrescriben a los superiores (anteriores). Dada la estructura que se utiliza para almacenar las variables, el orden de complejidad de obtención dado el nombre de una variable es logarítmico con respecto al número de variables definidas en el contexto, por lo que debería ser prácticamente inmediato en la mayoría de casos.

Cabe destacar que esta estructura está completamente protegida contra errores de concurrencia con un cerrojo de exclusión mutua recursivo.

Sintaxis propias

El almacenamiento de estas estructuras es muy similar al de las sintaxis de clases, pero no es necesario almacenar información de tipos. Se define esta lista en el código de la siguiente manera:

```
extern Mapper<unsigned int,
std::tuple<Pattern, std::function<Object(const std::string&,
unsigned int, unsigned int)>, ReturnType>> customSyntaxList;
```

La lógica detrás de esto es la misma que la de la sintaxis de clases: cada vez que haya que parsear una línea de código, se llamará al algoritmo de ejecución para las sintaxis que estén definidas en esta lista de manera secuencial, teniendo preferencia las primeras (propiedad de superposición). Si alguna de estas coincide con el formato de la cadena, se llamará a la función de parsing asociada con la subcadena analizada.

Se decidió almacenar las sintaxis en un array asociativo para hacer el proceso de almacenamiento más sencillo y los identificadores más explícitos, pero podría ser sustituido por un array dinámico.

Código

Representación de expresiones

Árbol de tokens

Pese a que no es nada raro que un algoritmo de parsing realice la etapa de lexing de manera implícita y obtenga una representación final directamente del texto, es conveniente que en Ulan sea explícita, ya que es un lenguaje con una naturaleza muy cambiante y podría complicar la implementación.

Siguiendo el diseño de otras clases descritas anteriormente, para representar esta etapa intermedia se utilizó un árbol con los siguientes tipos de nodos:

- **Object:** el nodo contiene un objeto. Aquí se almacena el resultado de parsear una clase con sintaxis definida.
- **Operator:** el nodo contiene un operador.
- **Reserved Word:** el nodo contiene una palabra reservada.
- **Class name:** el nodo contiene un nombre de clase.
- **Variable:** el nodo contiene un nombre de variable.
- **Function name:** el nodo contiene un nombre de función.
- **Container subtype:** el nodo contiene el nombre de un subtipo de Container.
- **Custom syntax:** el nodo contiene una función con sintaxis.
- **Composition:** el nodo contiene una lista de nodos que van seguidos en el código.
- **Encapsulation:** el nodo contiene otro nodo que está encapsulado de alguna manera.
- **Serialization:** El nodo es una serialización.

Cada uno de estos tipos de nodo está codificado de manera interna por un enum llamado *ParsingNodeType*. Dicho esto, queda especificar qué datos tiene cada tipo de nodo:

Nodo	Tipo asociado	Descripción
Object	`Object`	El nodo almacena el objeto que resulta de la ejecución de la función de parseo sobre la subcadena que representa al objeto
Operator	`unsigned int`	El nodo almacena el identificador interno del operador al que apunta
Reserved word	`unsigned int`	El nodo almacena el identificador interno de la palabra reservada a la que apunta. Las palabras reservadas están indexadas en una lista global como cualquier otra estructura y tienen un enum para representarlas
Class name	`unsigned int`	El nodo almacena el identificador interno de la clase a la que apunta

Nodo	Tipo asociado	Descripción
Variable	`std::string`	El nodo almacena la representación como cadena del nombre de la variable. No se refleja si está definida o no
Function name	`unsigned int`	El nodo almacena el identificador interno de la función a la que apunta
Container subtype	`unsigned int`	El nodo almacena el identificador interno del subtipo de Container al que apunta. Estas palabras reservadas también están indexadas en una lista global y tienen un enum propio
Custom syntax	`std::pair<unsigned int, std::string>`	El nodo almacena un par de datos: • Identificador interno de la función con sintaxis a la que hace referencia. Este identificador es el índice de la sintaxis, que es sinónimo del número de sintaxis propias que había cuando esta se definió • Subcadena a la que se le debe aplicar la función de parseo definida por el usuario
Composition	`std::vector<ParsingTree>`	El nodo almacena una lista de nodos que se encuentran uno detrás de otros en el código. Estos pueden ser expresiones complejas en ciertas situaciones que se mencionarán en el algoritmo de tokenización/lexing
Encapsulation	`std::pair<unsigned int, ParsingTree>`	El nodo almacena un par de datos: • Identificador interno del encapsulador que contiene a la expresión interna, ya que los encapsuladores también están identificados con números • Subárbol que se encuentra encapsulado
Serialization	`std::vector<ParsingTree>`	El nodo almacena la lista de nodos que están serializados. Dado que las serializaciones son un caso especial en la sintaxis no es necesario almacenar ningún identificador.

El manejo de la estructura es simplista y está dominado por el constructor por cadena, que es la única manera de construir un árbol de manera fiable. Toda la gestión de los datos de los diferentes nodos es similar a la de estructuras como las de la clase Symbolic.

Árbol de representación final

Una vez obtenido el árbol de representación intermedia se puede obtener la representación final como expresión Ulan mediante la utilización de numerosos algoritmos de reconocimiento de patrones. Esta clase resultante es de una complejidad mucho mayor que la anterior, por lo que su división en nodos también lo es, así que se presentarán divididos en diferentes secciones:

- **Funcionalidades básicas:** modelan comportamientos básicos del lenguaje, así como estructuras comunes.
  - **Object:** almacena un objeto con sintaxis definida. Un claro ejemplo sería la clase Number.
  - **Variable:** almacena una variable opcionalmente tipada. En el caso de que no lo esté, se le asignará el tipo Any.
  - **Container:** almacena una definición estándar de un Container con llaves junto a su subtipo.
  - **Comparator Container:** almacena una definición de Container con Comparator.
  - **Mapper:** almacena una definición de Mapper estándar.
  - **Lambda:** almacena una definición de lambda-expresión.
- **Asignaciones:** son una división necesaria para que el intérprete contemple más fácilmente las operaciones de asignación.
  - **Assignment:** asignación con operadores estándar.
  - **Indirect assignment:** asignación con operador de asignación a referencia.
- **Cálculos:** hacen referencia a cualquier llamada a un procedimiento definido que deba ejecutarse.
  - **Cast:** contiene un cast de un tipo a otro.
  - **Function:** contiene una llamada a función.
  - **Member function:** contiene una llamada a función miembro.
  - **Custom syntax:** contiene una llamada a una función con sintaxis.
  - **Sequence:** almacena una secuencia.
  - **Operation:** almacena una operación.
- **Nodos auxiliares:** estos nodos son utilizados únicamente de manera auxiliar durante el algoritmo de parsing. Una expresión final solo tendrá uno de estos nodos en localizaciones muy concretas.
  - **Class:** contiene un nombre de clase.
  - **Function definition:** contiene una definición de parámetros de función. Es utilizado para definir lambda-expresiones.
  - **Function definition with capture:** igual que el anterior, pero con una captura definida.
  - **Calculation operation definition:** internamente, las definiciones de operaciones pueden ser modeladas como operaciones comunes, pero dada la sintaxis del operador del cálculo es necesario que tenga su propio nodo.
  - **High level tuple:** contiene una tupla de alto nivel con cualquier elemento.

Es necesario mencionar que cada nodo de esta representación final como árbol tiene un elemento extra que el árbol anterior no tiene: un identificador numérico. Esto se debe a que más de la mitad de los nodos necesitan apuntar a una estructura global, así que tiene lógica evitar tuplas innecesarias en los datos mediante un atributo nuevo.

Queda entonces definir la semántica de este identificador en cada tipo de nodo y los datos que contiene en cada caso:

Nodo	Tipo asociado	Id	Descripción
Object	`Object`	Irrelevante	El nodo contiene una instancia de objeto
Variable	`std::pair<std::string, ReturnType>`	Irrelevante	El nodo contiene el nombre de una variable junto a su tipo si lo tiene
Container	`std::vector<Expression>`	Subtipo de Container	El nodo contiene los elementos que tiene la estructura. Pueden ser secuencias de comprensión de listas
Comparator Container	`std::pair<Expression, std::vector<Expression>>`	Subtipo de Container	El nodo contiene los elementos que tiene la estructura. Pueden ser secuencias de comprensión de listas
Mapper	`std::vector< std::pair<Expression, Expression>>`	Irrelevante	El nodo contiene los pares de elementos que tiene la estructura.
Lambda	`std::pair< std::pair< std::vector<std::string>, Parameters>, Expression>`	Irrelevante	Contiene una lista de nombres de variables que conforman la captura, los parámetros de la función y la expresión contenida

Assignment	`std::pair<unsigned int, std::pair<Expression, Expression>>`
	`std::pair<unsigned int, std::vector<std::pair<Expression, Expression>>>`
	`std::pair<unsigned int, std::vector<Expression>>`
		Id. del tipo de asignación que se está realizando	Dependiendo del tipo de asignación puede contener el id. del operador junto a:
			• Dos operandos
			• Una lista de pares de operandos
			• Una lista de operandos en orden
			Se hace una multiplexión de funcionalidad de manera interna

Nodo	Tipo asociado	Id	Descripción
Indirect assignment	`std::pair<Expression, Expression>`	Irrelevante	Contiene los dos operandos de la asignación
Cast	`Expression`	Id. de la clase destino del cast que se realiza	Contiene la expresión a la que se le quiere aplicar el cast
Function	`std::vector<Expression>`	Id. de la función que se llama	Contiene los argumentos de llamada de la función
Member function	`std::pair<Expression, Expression>`	Irrelevante	Contiene la expresión que devuelve el objeto de llamada y los argumentos de llamada
Custom syntax	`std::string`	Id. de la sintaxis en la lista global	Contiene la cadena a la que se le debe aplicar la función
Sequence	`std::vector<Expression>`	Id. de la secuencia a la que se apunta	Contiene las expresiones entre las palabras reservadas de la secuencia
Operation	`std::vector<Expression>`	Id. del operador	Contiene los operandos
Class	-	Id. de la clase	No tiene datos
Function definition	`Parameters`	Id. de la función que se está definiendo si no es una lambda-expresión	Contiene los parámetros de la función a definir
Function definition with capture	`std::pair< std::vector<std::string>, Parameters>`	irrelevante	Contiene los parámetros y la captura de la función a definir
Calculation operation definition	`std::pair<Expression, Parameters>`	irrelevante	Contiene la variable de llamada y los parámetros de la nueva operación a definir
High level tuple	`std::vector<Expression>`	Irrelevante	Contiene los argumentos encapsulados en la tupla

Estructuras auxiliares

Module

Esta clase almacena de manera ordenada un conjunto de líneas de código junto a su indentación original para poder identificar los contextos. Asimismo, también se incluye un nombre y un vector para poder obtener la correspondencia entre líneas del módulo y líneas reales en el archivo. Esto último es utilizado para dar mensajes de error más legibles.

Es importante mencionar que las cabeceras no son almacenadas en el módulo, sino que se modifican variables globales durante el proceso de parsing. Esto se debe a que no se considera necesario paralelizar el algoritmo.

Dicho esto, la representación como tipos nativos es la siguiente:

```
std::string name;

std::vector<std::pair<unsigned int, Expression>> body;
std::vector<unsigned int> lineMapping;
```

Program

Esta es la estructura final que se ejecutará al interpretar un programa Ulan completo. Es una clase sencilla que contiene un puntero al módulo principal del programa y una lista de punteros a cada uno de los módulos necesarios como dependencia. Se almacenan de esta manera en vez de manera directa porque las funciones definidas durante la etapa de parsing dependen de una referencia al módulo, la cual cambia si no se hiciera de esta manera. Los módulos son eliminados de la memoria de forma normal al terminar la ejecución.

Para saber en qué orden se deben interpretar los módulos se calcula el orden topológico con una búsqueda en profundidad. El último módulo en interpretarse es siempre el módulo principal del programa. Cabe destacar que no solo se interpretan las dependencias y se almacenan en estructuras, sino que se ejecutan por si tuvieran alguna definición de variables necesaria.

La representación como tipos nativos es la siguiente:

```
Module* mainModule;

std::vector<Module*> dependencies;
```

Parsing de cabeceras de módulo

Debido a la simplicidad del formato de las cabeceras el algoritmo de cada una de ellas depende enteramente de un patrón USDL asociado. Estos patrones están definidos en variables globales y se utilizan en funciones auxiliares que se ejecutan durante el parsing de cada módulo. La sintaxis de cada patrón es la siguiente:

- **Identificador de módulo:** $\{' '\}'Module'\{' '\}Arg(1\{l|d|'\_'\}, Lit, name)\ \{' '\}$
- **Dependencias:** $\{' '\}'Import'\{' '\}' < 'Arg(1\{l|d|'\_'\}, Lit, name)' > '\{' '\}$
- **Sintaxis aceptadas:** $\{' '\}'Import'\{' '\}' < 'Arg(1\{l|d|'\_'\}, Lit, name)' > '\{' '\}$

Parsing de expresiones

Lexing

El algoritmo de lexing de Ulan se basa en un recorrido recursivo sobre la cadena de llamada haciendo comprobaciones de formato siguiendo la sintaxis a nivel de texto de este lenguaje. Pese a no ser la estrategia más eficiente, sí se trata de la más flexible dada la naturaleza del lenguaje, ya que el intérprete es cambiante y es complicado definir conceptos tales como una sintaxis propia si no es mediante esta sintaxis recursiva.

Dicho esto, queda definir las comprobaciones de formato que se realizan, el orden en el que se hacen y las consecuencias que tienen sobre los iteradores y sobre la expresión a interpretar. A continuación se muestra la lista ordenada de comprobaciones junto a sus efectos:

1. **Si la subcadena está vacía:** se considera que la expresión es una serialización vacía. Esto solo ocurre cuando se trata de código encapsulado y sirve para simplificar el reconocimiento de patrones.
2. **Patrones USDL de clases aceptadas por el módulo:** si el resultado es positivo, se considera que el nodo es de tipo *Object* y se ejecuta la función de parsing correspondiente para obtener el objeto resultante y almacenarlo en el árbol.
3. **Patrones USDL de funciones con sintaxis:** si el resultado es positivo, se almacena el identificador de la sintaxis junto a la subcadena sobre la que se debe ejecutar la función de parsing.
4. **Comprobar si la subcadena es un elemento contextual:** si coincide se le asignará el tipo correspondiente (ver implementación de árbol de tokens para más información). Cabe destacar el caso de las palabras reservadas detectadas de esta manera, ya que se les asignará un nodo de tipo *Composition* con un solo elemento, ya que simplifica el reconocimiento de patrones.
5. **Búsqueda de encapsuladores:** si la subcadena actual está encapsulada de alguna manera se detectará y se le asignará un nodo *Encapsulation* con una expresión interna que consiste en avanzar un carácter cada iterador, ya que todos los encapsuladores ocupan exactamente un carácter.

6. **Comprobar si la subcadena es una serialización:** se considera que cualquier subcadena que tenga comas no encapsuladas es una serialización, calculándose los elementos de la misma los que se encuentran en los intervalos que separan las estos caracteres. Dadas *n* comas habrá *n* + 1 elementos, de lo contrario habrá un error de sintaxis.
7. **Realizar una búsqueda de secuencias:** se aplica un algoritmo de búsqueda de secuencias a la subcadena actual. Para esto se utiliza una estructura auxiliar llamada *Dictionary*, la cual almacena cadenas de una manera que permite aplicar el algoritmo de *Aho-Corasick*. Dado un *Dictionary* con las palabras reservadas que pueden formar secuencias se puede aplicar un algoritmo eficiente para buscar todas las palabras reservadas de la subcadena. Una vez encontradas se interpretan las expresiones que están entre cada par de palabras contiguas si la hubiera. Estas expresiones son interpretadas como nodos *Composition*.
8. **Realizar una búsqueda de operaciones:** se aplica un algoritmo parecido al anterior con un *Dictionary* que almacena los operadores infijos. Curiosamente, este algoritmo es mucho más complejo que el de búsqueda de secuencias, ya que es necesario aplicar varios filtros para distinguir varios casos ambiguos, como podría ser el caso de operadores prefijo o sufijo con la misma representación que uno infijo. Asimismo, es necesario tener en cuenta que puede haber operadores infijo cuya representación puede encajar con la de otro al concatenarlo con otro. El único caso que no está reflejado es el de varios operadores infijo cuyas representaciones concatenadas coinciden con la de un operador unario.
9. **Comprobar operaciones prefijo:** se considera que una subcadena es una operación prefijo cuando comienza con una subcadena que coincide con la representación de un operador prefijo. Se escogerá el operador prefijo con la representación más larga que encaje.
10. **Comprobar operaciones sufijo:** siguiendo los mismos criterios que el paso anterior, se considera que una subcadena es una operación sufijo cuando termina con una subcadena que coincide con la representación de un operador sufijo. Se escogerá el operador sufijo con la representación más larga que encaje.
11. **Comprobar composiciones complejas:** llegados a este punto, el único caso que queda comprobar son las composiciones de tokens que no necesariamente forman un elemento sintáctico completo. Un claro ejemplo de esto son las listas de argumentos con captura de las lambda-expresiones. Para encontrar estas separaciones se itera sobre la subcadena estableciendo una separación en cualquiera de estos casos:
    a. **Carácter alfanumérico seguido de un espacio o una apertura de encapsulador:** como puede ser el caso de llamadas a función.
    b. **Cierre de encapsulador seguido de apertura de encapsulador:** como el caso antes mencionado de las capturas.
    c. **Cierre de encapsulador seguido de un espacio o un carácter alfanumérico:** no tiene uso en la sintaxis actual, pero es necesario identificar este caso.

Parsing

Tras la etapa de lexing, tenemos un árbol que contiene toda la información necesaria para obtener la representación final para ejecución. Para esto es necesario realizar numerosas comprobaciones de formato sobre el árbol de tokens en un orden específico y teniendo en cuenta el contexto en el que se hacen. Las comprobaciones pueden ser divididas en comprobaciones atómicas (primera en realizarse, actúan sobre el árbol completo) y comprobaciones múltiples (últimas o por llamada recursiva, actúan sobre una lista de nodos). A continuación se describen las comprobaciones atómicas:

1. **Si el subárbol está encapsulado en paréntesis:**
    a. **Comprobar argumentos de lambda-expresiones:** si nos encontramos en el contexto de una operación de definición, se interpretarán estos paréntesis como unos parámetros de función.
    b. **Comprobar código encapsulado:** si no estamos en ese contexto, se interpreta como código estándar encapsulado en paréntesis.
2. **Si el código está encapsulado en corchetes:** se interpreta el árbol como una tupla de alto nivel, pero si no se está en el contexto adecuado se lanzará una excepción por tupla de alto nivel fuera de contexto.
3. **Comprobar definiciones de Mapper:** si el árbol coincide con la definición de un Mapper, se interpretará como tal. Cabe destacar que unas llaves vacías serán detectadas como un Mapper vacío.
4. **Comprobar definiciones simples de Container:** si el árbol coincide con la definición de un Container sin subtipo ni Comparator, se interpretará como tal.
5. **Comprobación de objetos:** si el nodo es un objeto se moverán los datos al nodo del árbol final.
6. **Comprobación de sintaxis propias:** si el nodo es una sintaxis propia se mueven los datos al nodo correspondiente en el árbol de representación final.
7. **Comprobación de variables:**
    a. **Variables no tipadas:** si el árbol es un nombre de variable o de función se interpreta como una variable de tipo Any.
    b. **Variables tipadas:** si el árbol es una operación de definición con un tipo en el operando derecho y un nombre de variable o de función en el derecho se interpreta como una variable del tipo especificado.
8. **Comprobar asignaciones paralelas:** si el árbol coincide con una asignación de tuplas de alto nivel se interpreta como una asignación paralela. Se realizan comprobaciones más concretas en la función de extracción del nodo.
9. **Comprobar operaciones prefijo:** si el árbol consiste en un operador prefijo seguido de una subexpresión, se interpreta como una operación prefijo.
10. **Comprobar operaciones sufijo:** si el árbol consiste en una subexpresión seguida de un operador sufijo, se interpreta como una operación prefijo.
11. **Identificar patrones complejos:** si ninguno si ninguno de los patrones anteriores se aplica al árbol dado como parámetro y el nodo actual es de tipo *Composition*, se pasa a la función de comprobaciones múltiples. Esta llamada es recursiva sobre los hijos del nodo sobre el que se encuentra el algoritmo.

Las comprobaciones múltiples son ligeramente más complejas que las atómicas, ya que pueden detectar patrones sobre el árbol que deban ser transformados si cumplen ciertas restricciones tras recorrer el algoritmo completo o depender de llamadas recursivas a la misma función. La lista completa en orden es la siguiente:

1. **Si se está analizando un único elemento:** se llama recursivamente al algoritmo de comprobaciones atómicas con el nodo único como argumento.
2. **Comprobar parámetros con captura:** si la lista de nodos consiste en una tupla de alto nivel con nombre de variables junto a unos parámetros encapsulados en paréntesis se interpretarán como un nodo de tipo *function definition with capture*.
3. **Comprobar variables tipadas:** si la sublista de nodos coincide con una variable tipada, se interpreta como tal.
4. **Comprobar construcciones de Container:**
    a. **Container con comparator sin subtipo:** si la secuencia es una tupla de alto nivel seguida de una construcción estándar de Container se interpreta como un array dinámico ardenado por Comparator.
    b. **Container con comparator y subtipo:** igual que el anterior, pero con un subtipo especificado.
    c. **Container sin comparator y con subtipo:** igual que el anterior, pero sin Comparator.
5. **Comprobar secuencias:** dada la construcción del árbol de tokens es sencillo realizar una comprobación de este tipo. Existe una lista global con las secuencias definidas en el sistema. Una posible optimización sería almacenarlas en una estructura similar a un árbol de radicales para tener un tiempo de búsqueda mejor. Las expresiones intermedias son parseadas de manera recursiva, empezando por comprobaciones atómicas.
6. **Comprobación de casts:** si la estructura es análoga a una llamada a función, pero con una clase, se parsea como un cast.
7. **Comprobación de llamadas a función:** si se trata de un nombre de función seguido de unos argumentos de llamada encapsulados en paréntesis, se parsea como una llamada a función.
8. **Comprobar operaciones binarias:** al igual que en el caso de las secuencias, el árbol permite establecer una estructura recursiva sencilla. Se parte la lista de nodos dividiendo por los tokens que correspondan con el operador con la precedencia más baja de la lista. Se utiliza un flag para no hacer comprobaciones redundantes sobre la lista. Si el formato coincide con el de una lambda expresión o una asignación, el nodo se transforma tras la ejecución del algoritmo.
9. **Comprobar funciones miembro:** si la lista termina con un operador de acceso seguido de una llamada a función, se parsea como una llamada a función miembro de la subexpresión anterior a esos tokens. Cabe destacar que las normas para el operador de acceso son ligeramente especiales.
10. **Comprobar operaciones de cálculo:** si la secuencia termina con unos argumentos encapsulados en paréntesis se interpreta como una operación de cálculo.
11. **Comprobar operaciones de índice:** se sigue la misma lógica que para la operación de cálculo, pero con tokens encapsulados en corchetes.

Parsing de módulos e inferencia

Cuando se habla del proceso de parsing de módulos se hace referencia al proceso completo que sigue el intérprete para pasar de un archivo de texto a un módulo Ulan válido. Se ignora el proceso de búsqueda de dependencias, ya que tan solo es un cálculo de orden topológico. Este apartado se centra en el parsing de un módulo sin dependencias.

Hablando de manera general y sin demasiados matices, el parsing de módulos consiste en el parsing de todas las líneas que contiene de manera secuencial. Estas líneas pueden tener un tratamiento ligeramente especial dependiendo de su tipo. A continuación se especifican los casos especiales que hay que tener en cuenta.

Manejo de contextos

Cuando se obtiene una nueva línea del archivo se obtienen dos datos: la cadena de caracteres resultante y un número entero sin signo que representa su indentación. En el momento que se detecta una subida en la indentación se abre un nuevo contexto. Asimismo, cuando se detecte una bajada se cerrarán los contextos que sean necesarios. Es necesario mantener una pila de contextos donde se almacene la indentación asociada y el tipo, ya que el parsing de expresiones individuales depende de estos contextos.

Es necesario mencionar que esto es solo una simulación de lo que ocurriría si se ejecutara el código para poder interpretar el contexto de parsing, pero los contextos reales no almacenan su tipo, sino que están representados por la profundidad de la estructura que almacena las variables. **El módulo no se puede ejecutar hasta estar completamente interpretado**.

Ejemplo:

```
var = 5 //Pila: {Primario}

Define function f() as
    a = 5 //Pila: {Cuerpo de función, Primario}
    return a

for i in range(100) //Pila: {Primario}
    if i > 50 //Pila: {Contexto común, Primario}
        f() //Pila: {Contexto común, Contexto común, Primario}

var = f() + 1 //Pila: {Primario}
```

Definición de estructuras

Otro caso especial a tener en cuenta es cuando se define una nueva estructura, independientemente del tipo al que pertenezca, ya que es necesario actualizar las estructuras del intérprete. Para esto se sigue el siguiente esquema:

1. **Parsear todas las líneas pertenecientes a la subrutina**, incluida la cabecera.
2. **Definir la estructura** eligiendo la definición exacta dependiendo del tipo de cabecera.
3. **Inferir el tipo de retorno** de la estructura si se trata de un cuerpo de función.

Algoritmo de ejecución

Teniendo el árbol de representación final y los algoritmos de resolución de operaciones, la ejecución de expresiones individuales se convierte en un proceso completamente intuitivo. La idea básica es empezar desde la raíz del árbol resultante y utilizar los identificadores internos para acceder a las estructuras correspondientes. Si se tiene una interfaz de acceso a las listas globales este proceso se abstrae completamente de cómo estén almacenadas y se convierte en una nueva capa del intérprete.

El algoritmo se complica en el momento que dejamos de hablar de ejecución de expresiones y hablamos de ejecución de módulos, ya que empezamos a necesitar una definición más clara de cómo se ejecuta código parametrizado. Para esto introducimos el concepto de *subrutina*, la cual es definida como un segmento de código secuencial con una cabecera y una indentación mínima. El sistema (concretamente la clase *Module*) cuenta con funciones que permiten ejecutar subrutinas dados el módulo y la línea que hace de cabecera. Independientemente de la definición anterior, la ejecución de módulos podría modelarse como la ejecución secuencial se sus líneas de código.

Modelado de cuerpos de función

Las expresiones que pueden hacer de cabecera de un cuerpo de función son variadas y cada una tiene sus propias maneras exactas de ser almacenadas, pero el algoritmo de ejecución es prácticamente el mismo: se ejecuta una función anónima que tiene como captura las variables necesarias para identificar los parámetros, una referencia constante al módulo y el número de línea correspondiente a la cabecera. La ejecución final suele tener la siguiente estructura:

1. **Apertura de contexto.**
2. **Definición de variables necesarias en lista global.**
3. **Ejecución de la subrutina** con manejo de excepciones para obtener traza de ejecución en caso de excepción.
4. **Cierre de contexto.**
5. **Devolución de valor.**

Asimismo, es necesario mencionar que estas funciones no se suelen ejecutar por sí solas, sino que se suelen usar *wrappers* que comprueban la validez de los parámetros y la coherencia del tipo de objeto devuelto. Estas comprobaciones son necesarias, dado el tipado gradual del lenguaje.

Modelado de controladores de flujo

Estas expresiones se detectan en el bucle de ejecución secuencial, ya que están categorizadas como expresiones “no ejecutables”. Al encontrarse una de estas se hace una multiplexión por identificador de secuencia. Todas estas secuencias suelen seguir este patrón:

1. **Definición concreta de variables de trabajo** (Containers de *for*, por ejemplo).
2. **Apertura de contexto.**
3. **Ejecución de subrutina.**
4. **Cierre de contexto.**

Detalles del intérprete e instrucciones de uso

Dicho todo esto, queda concluir explicando el funcionamiento exacto del archivo ejecutable generado por el código. Tan solo es necesario especificar cuáles son los argumentos de entrada del archivo, ya que no tiene ningún tipo de interfaz gráfica.

El primer argumento debe ser la ruta al archivo con extensión *.ulan* que se desea ejecutar. Este será el módulo del que se buscarán dependencias en los directorios de búsqueda, los cuales estarán especificados del segundo argumento en adelante, sin haber un número máximo. Cabe destacar que no es necesario especificar ningún directorio de búsqueda si no se tienen librerías.

Ejemplo:

"C:/ulan.exe" "C:/ulanPrograms/p.ulan" "-DC:/ulanLibs1" "-DC:/ulanLibs2" -OLazyLogic

Quizás la manera más común de utilizar un intérprete como este es utilizar un editor de texto que permita *plugins*. En el caso de Sublime Text 3, editor en el que se realizaron todas las pruebas y en el que se definió la sintaxis más completa, podría definirse un archivo con extensión *.sublime-build* con el siguiente contenido:

```
{
  "cmd": "start cmd /c \"\"Ruta_Intérprete\" \"$file\" \"-DRuta_ Librerías\" -OLazyLogic && echo. && pause\"",
  "shell": true
}
```

Como se puede observar, el intérprete de Ulan puede leer diferentes tipos de argumentos de entrada. Estos argumentos no tienen ningún orden en particular y pueden mezclarse como se vea conveniente. A continuación se describen:

- **Flags de optimización:** marcados con "-O<flag>", se trata de las diferentes técnicas que puede utilizar el intérprete para optimizar el código. Actualmente están definidos *LazyLogic* (habilita la evaluación cortocircuitada de operadores lógicos) y *PartialEvaluation* (permite al intérprete evaluar expresiones constantes).
- **Directorios de búsqueda:** marcados con "-D<directorio>", añaden un directorio a la lista interna de directorios de módulos Ulan. Las dependencias del programa deben estar especificadas con argumentos de este tipo.
- **Variables globales:** marcados con "P<variable>=<entero>", establecen un valor para un parámetro interno del intérprete. Actualmente están definidos *STDPrecision* (precisión por defecto para objetos de clase Number, que es 10 por defecto) y *SpacesPerTab* (tamaño de la tabulación en cálculos de indentación, que es 3 por defecto).

Sección V

Validación y resultados

Introducción

Finalidad

A continuación se van a mostrar algunos fragmentos de código en Ulan que presentan alguna característica interesante a nivel de funcionalidad. La finalidad principal de estos programas es demostrar que el intérprete funciona correctamente y que es capaz de generar datos semánticamente relevantes. Asimismo, también se pretende que se ejemplifique la expresividad del lenguaje.

Todos los programas que se van a mostrar en las siguientes páginas de esta sección tienen una salida que coincide con la esperada, por lo que se asume que su presentación constituye una validación suficiente de la corrección y completitud del producto desarrollado.

Pruebas automáticas

Pese a lo mencionado anteriormente, estos programas no han sido la única herramienta utilizada para comprobar la corrección del intérprete. A lo largo del desarrollo se han ido añadiendo casos de estudio a una suite de pruebas automáticas que contiene los datos necesarios como para hacer un análisis preliminar suficiente.

La suite generada se divide en las siguientes partes:

- **Pruebas de parsing:** se componen de alrededor de 70 casos distintos que analizan si las diferentes clases de expresiones se pueden crear correctamente y si se detectan errores de parsing en los casos adecuados. Estas son divididas en pruebas de lexing y de parsing para tener una mayor granularidad.
- **Pruebas de ejecución:** analizan la ejecución de 5 módulos de prueba en los que se generan diferentes valores utilizando diversas técnicas y se almacenan en variables. Estas variables son analizadas tras la ejecución de los módulos y se comparan con los valores teóricos que deberían tener almacenados.
- **Pruebas de rendimiento de parsing:** realizan un parsing repetido de diferentes expresiones de complejidad variable con el intérprete “vacío” para comprobar el tiempo medio de ejecución. Estas pruebas permiten separar el tiempo de lexing del tiempo de parsing, por lo que se pueden extraer datos tales como que el parsing es mucho más rápido que el lexing, permitiendo así un profiling de baja granularidad.

Ejemplos básicos

Funcionalidades básicas

Una vez completada la implementación del intérprete solo queda probar las características del lenguaje resultante. A continuación se muestran algunos ejemplos de programas que comprueban las funcionalidades básicas de Ulan.

Definición de funciones

En este programa se define una función para hacer comprobaciones de la *conjetura de Collatz*, que es un problema no resuelto en las matemáticas que afirma que toda secuencia construida a partir de un número siguiendo ciertas normas acabará en 1. La función resultante es la siguiente:

```
Define function collatz(n : Number) as
    counter = 0

    while n != 1
        if n.isEven()
            n /= 2
            counter += 1

        else
            n *= 3
            n += 1
            n /= 2
            counter += 2

    return counter

Define function namedPrint(name : String, obj) as
    println(name + ": " + obj)

n = 279731455495736617
c = collatz(n)

namedPrint("Number", n)
namedPrint("Collatz", c)
```

Salida:

```
Number: 279731455495736617
Collatz: 2258
```

Comprensión de listas

En este programa se generan varios Containers utilizando mecanismos de comprensión de listas con filtros y sin buscar una semántica especialmente relevante. El programa resultante es el siguiente:

```
c1 = {i for i in range(10) if i.isPrime()}
c2 = {{i, j} for [i, j] in [range(1, 7), range(1, 7)] if (i + j).isPerfectSquare()}
c3 = {i*100 + j*10 + k for [i, j, k] in [range(1, 9), range(1, 9), range(1, 9)] \
                                    if i*100 + j*10 + k == i! + j! + k!}

println(c1)
println(c2)
println(c3)
```

Salida:

```
{2, 3, 5, 7}
{{1, 3}, {2, 2}, {3, 1}, {3, 6}, {4, 5}, {5, 4}, {6, 3}}
{145}
```

Definición de operadores

El único propósito de este programa es ejemplificar la definición de operadores:

```
Define full infix operator "to" prec 800

Define operation (n1 : Number) to (n2 : Number) as
    return range(n1, n2)

c = {} : SortedVector

Define function fullAdd(c : Container&, n1 : Number, n2 : Number) as
    for i in n1 to n2
        c.add(i)

namedPrint("Before", c)

fullAdd(c, 0, 5)

namedPrint("After", c)
```

Salida:

```
Before: {}
After: {0, 1, 2, 3, 4}
```

Uso de sintaxis propias

En el siguiente programa se muestra un ejemplo de uso de una sintaxis propia (en este caso la definida en la librería Dice). Cabe destacar que se utiliza la función *namedPrint*, la cual fue definida en un apartado anterior. El programa es el siguiente:

```
dice = 1D20

namedPrint("Dice", dice)
namedPrint("Rolls", {dice.roll() for i in range(5)})
namedPrint("Direct rolls", {<2D8> for i in range(5)})
```

Posible salida:

```
Dice: 1D20
Rolls: {17, 3, 9, 1, 18}
Direct rolls: {5, 10, 13, 13, 9}
```

Lambda-expresiones

En este programa se usan lambda-expresiones para definir secuencias e iterar sobre ellas. El programa resultante es el siguiente:

```
namedPrint("Sequence definition", {i for i in sequence(-5, 5, (x) : x + 1)})
namedPrint("Intension definition", {i for i in intension(0, (x) : x == 10, (x) : x + 1)})

println("")

Define function nextPrime(n : Number) as
    if n%2 == 0
        n += 1

    else
        n += 2

    while !(n.isPrime())
        n += 2

    return n

namedPrint("Primes", {i for i in intension(2, (x) : x >= 50, (x) : nextPrime(x))})
```

Salida:

```
Sequence definition: {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4}
Intension definition: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Primes: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}
```

Programas intensivos

Análisis de secuencias de números primos

Es de esperar que al analizar la secuencia generada por los últimos dígitos de los números primos hasta un cierto límite numérico la distribución sea uniforme, pero resulta que la intuición no hace ningún bien en este caso. Si analizamos las secuencias de estos últimos dígitos y a su vez analizamos las ocurrencias de sucesiones de dígitos contiguos observamos que la distribución no es uniforme [15]. Precisamente para corroborar estos experimentos se realizó un programa Ulan que los simulaba:

```
Import <ContainerAlgorithms>

upperBound = 170000000
sequenceSize = 2

lastDigits = {i for i in sieve(upperBound)}.map((n : Number) : n.getIntDigit(n.getIntDigitNumber() - 1))
stats = range(lastDigits.size() - sequenceSize + 1).count((i) : {lastDigits[i + j] for j in range(sequenceSize)})
sum = stats.values().reduce((x, y) : x + y, 0)

for i in stats.keys()
   stats[i] := stats[i] / sum

println("Ordered results:")

for i in [comparing((c) : c[1], true)] {{j, stats[j]} for j in stats.keys()}
   p = i[1] * 100
   p.round(2)

   if p > 0
      println(i[0] + " -> " + p + "%")
```

La ejecución de este programa está controlada por dos parámetros que han sido tanteados para ser lo más altos posible sin provocar un crash por falta de memoria, ya que la versión de Ulan que lo ejecuta está compilada para 32 bits.

Para estos valores la salida es la siguiente:

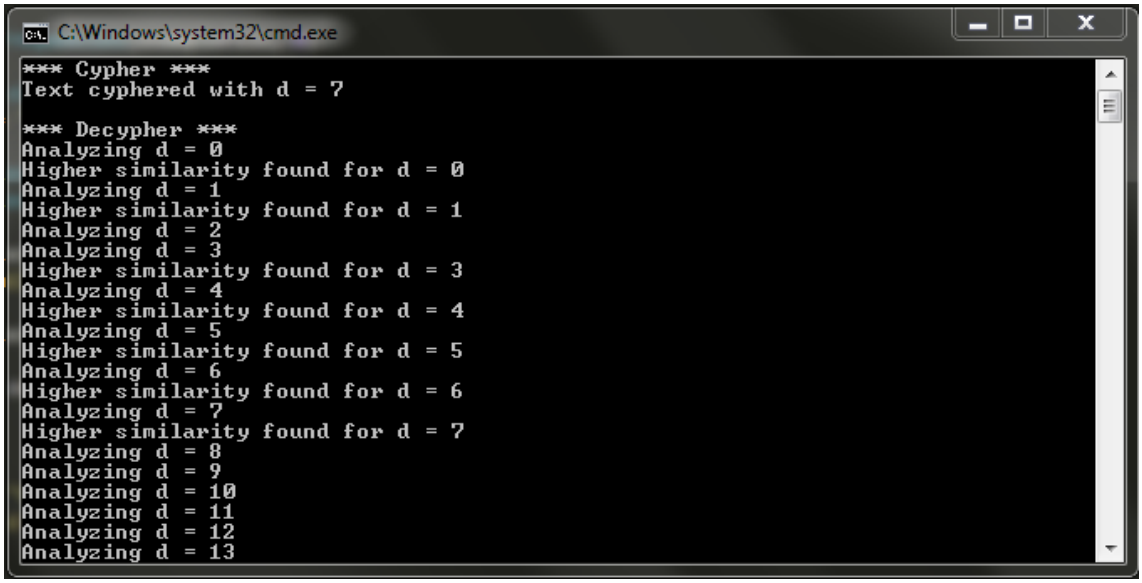
```
Ordered results:
{9, 1} -> 8.21%
{3, 9} -> 7.7%
{1, 7} -> 7.7%
{7, 9} -> 7.57%
{1, 3} -> 7.57%
{3, 7} -> 7.15%
{7, 3} -> 6.82%
{9, 3} -> 6.4%
{7, 1} -> 6.39%
{3, 1} -> 5.93%
{9, 7} -> 5.93%
{1, 9} -> 5.26%
{1, 1} -> 4.47%
{9, 9} -> 4.46%
{7, 7} -> 4.22%
{3, 3} -> 4.22%
```

Cifrado afín y análisis criptográfico

Existen numerosas maneras de cifrar un mensaje para dificultar su lectura por personas no deseadas, siendo el cifrado por desplazamiento una de las más antiguas. Esta técnica consiste en utilizar un alfabeto almacenado en una lista indexada y "desplazar" cada carácter del texto original un número d de caracteres para generar un mensaje nuevo.

El principal problema que tiene este método es que los lenguajes tienen unas frecuencias de uso de caracteres muy determinadas, por lo que basta con analizar un texto en el mismo lenguaje que el texto cifrado para saber estas frecuencias y probar todos los desplazamientos posibles (este número es igual al tamaño del alfabeto) para saber cuál de estos produce un texto que encaje mejor con las mismas. Esta puede resultar una tarea tediosa a mano, pero es automatizable.

Se implementó un programa en Ulan que permite cifrar archivos de texto utilizando este método, pero también se implementó un método que hacía un análisis de frecuencias de un texto de ejemplo y que las comparaba con los textos resultantes de hacer un desplazamiento para cada desplazamiento posible. Cuando se identificaba el desplazamiento con la similitud más alta, se realiza un cifrado con el desplazamiento inverso para deshacerlo.



Para asegurarse de que el algoritmo verdaderamente funciona, el texto de ejemplo y el texto cifrado solo comparten el lenguaje, pero provienen de diferentes sitios y son diferentes estilos literarios. Concretamente, para este ejemplo se utilizó un fragmento de *El árbol de la ciencia* (Pío Baroja) como texto base y un fragmento de *Planilandia* (Edwin A. Abbott) traducido al español como texto a cifrar.

Los datos experimentales confirman el hecho de que no es necesario hacer un análisis exhaustivo del texto cifrado para saber qué desplazamiento se utilizó, así que existe un límite de caracteres a analizar para hacer un criptoanálisis mucho más rápido.

Juegos

Buscaminas

Utilizando la clase básica Matrix se puede modelar un tablero de buscaminas fácilmente. En este caso, se modeló con la siguiente clase:

```
Define class "MinesweeperGame" as
   board : Matrix      //Tablero: Minas y números (Mina -> -1)
   markdowns : Matrix  //Marcas en el tablero (Nada -> 0, Abierto -> 1, Marcada -> 2)
```

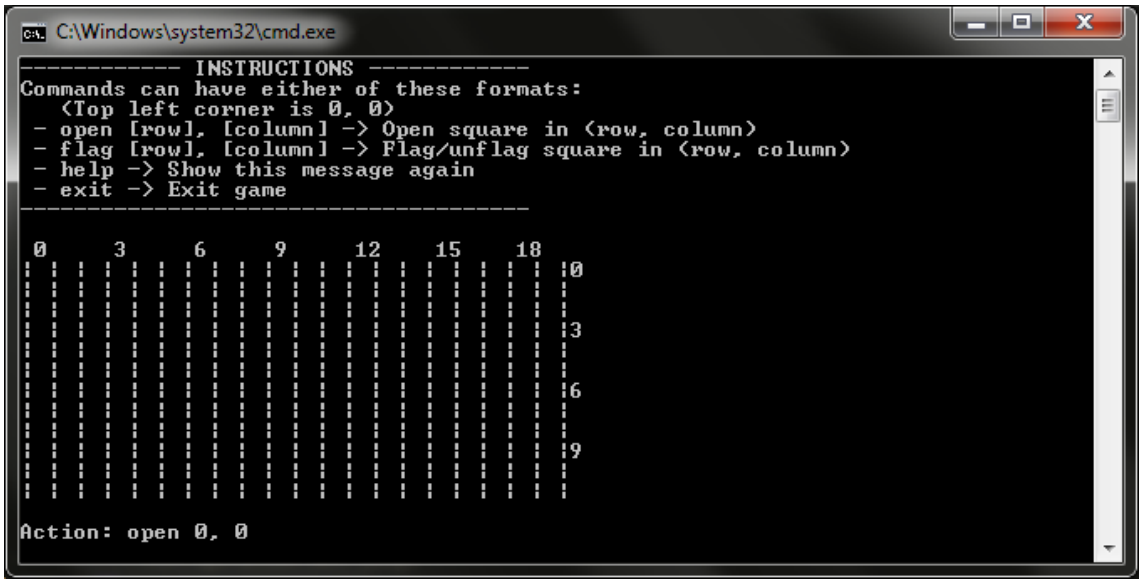
Ignorando aspectos de implementación no relevantes, se puede decir que todo el juego depende de una función miembro llamada *play* que se dedica a extraer los argumentos de los comandos introducidos por el jugador. Estos comandos son parseados utilizando la siguiente línea USDL:

$$Arg('flag' \mid 'open', Lit, action)\ \{' '\}\ Arg(1\{d\}, Number, i)\ \{' '\}\ ','\ \{' '\}\ Arg(1\{d\}, Number, j)\ \{' '\}$$

Teniendo en cuenta esto basta con inicializar una instancia de tablero inicial y llamar a la función *play* sobre esta:

```
//----------- Configuración del juego -----------
columns = 20
rows = 12
mines = 35
//-----------------------------------------------

game = _MinesweeperGame_(rows, columns, mines)

game.play()
```

El juego progresará exactamente como se espera de un buscaminas:



Tres en raya con inteligencia artificial sencilla

Si utilizamos un juego más sencillo podemos crear una inteligencia artificial básica que juegue siguiendo un algoritmo de búsqueda con heurística. Para esto elegimos el clásico *tres en raya*, apoyándonos en la librería *Search* y el algoritmo de búsqueda A*.

La clase que define un estado del tablero sería la siguiente:

```
Define class "TTTBoard" as
   board : Matrix
   turn : Boolean //True -> ponen las X, False -> ponen las O
```

Siguiendo la misma lógica que con el buscaminas, basamos toda la lógica en una extracción de comandos con USDL en una función *play*, pero con la diferencia de que tras cada turno del jugador la máquina pondrá una pieza. Para esto solo hay que analizar la función que calcula su movimiento:

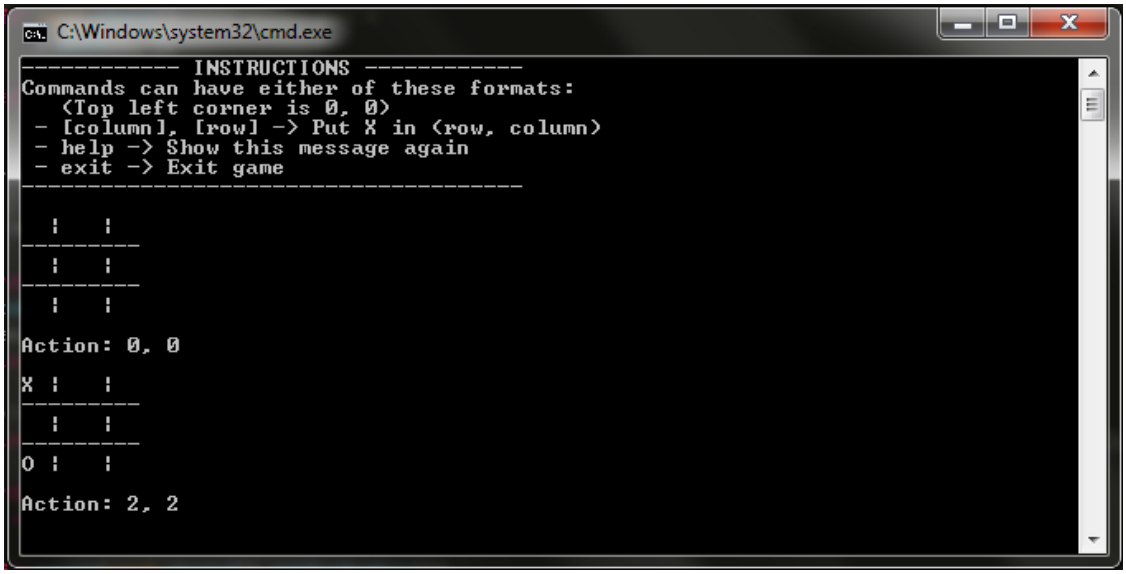
```
Define const function TTTBoard::nextMove(randomMoveProbability : Number) as
   if probability(randomMoveProbability) //Movimiento aleatorio (10% de las veces)
      moves = self.movements()
      return moves[rand(0, moves.size() - 1, true)]

   else //Movimiento con heurística
      return search_i(self, (x) : x.isFull() || x.isWon() == 2, (x) : x.movements(), (x) : x.heuristic(), astar_i)[1]
```

Se puede observar que existen dos posibilidades:

- La máquina ejecuta un movimiento al azar de entre los posibles
- La máquina ejecuta el movimiento óptimo según el algoritmo de búsqueda

Esto se hace para evitar jugar varias partidas similares, siendo *randomMoveProbability* un parámetro global que se puede configurar antes de iniciar el juego. Se considera que explicar la heurística que se sigue está fuera del contexto del documento.

Siguiendo estas pautas, el juego se desarrolla como debería:



Librerías adicionales

La mayoría de los lenguajes tiene soporte para definir módulos con código perteneciente a un ámbito. Estos son llamados *librerías*. La primera versión de Ulan es distribuida con algunas librerías con usos interesantes.

Complex

Una de las principales quejas que se podrían dar sobre el lenguaje es que no posee capacidades para hacer cálculos con números complejos. Esto es solucionado gracias a esta librería externa que le da al programador una gran cantidad de funciones relativas a esta clase de números.

La clase está definida como sería intuitivo hacerlo:

```
Define class "Complex" as
    real : Number
    imag : Number
```

Asimismo, se definen sintaxis para poder escribir números complejos con multiplicaciones implícitas ($3i$, por ejemplo) y no depender obligatoriamente de variables globales o constructores y el cast a String para representación correcta:

```
//******* Sintaxis *******

Define syntax for Complex from "[Arg(<Number>, Number, imag)] 'i'" as
    return _Complex_(0, args["imag"] if args.containsKey("imag") else 1)

//******* Casts *******

Define cast String(c : Complex) as
    res = (c.real() if c.real() != 0 || c.imag() == 0 else "") //Parte real
    res += (" + " if c.imag() > 0 && c.real() != 0 else "") // Signo "+"
    res += (" - " if c.imag() < 0 && c.real() != 0 else "") // Signo "-"
    res += (abs(c.imag()) if c.imag() != 0 && abs(c.imag()) != 1 else "") //Parte imaginaria
    res += ("-" if c.imag() == -1 && c.real() == 0 else "") //Negación de i
    res += ("i" if c.imag() != 0 else "") //Símbolo "i"

    return res
```

Esta librería da soporte para la gran mayoría de funciones que están definidas de manera nativa para números reales, tales como *sqrt*, *ln*, funciones trigonométricas comunes y funciones hiperbólicas. También se hacen las sobrecargas necesarias en los operadores aritméticos para poder operar con estos números de manera natural.

Se pueden mencionar de una manera más anecdótica la conversión a coordenadas polares con sus respectivas funciones de módulo y argumento y la función que comprueba si un número dado es un primo gaussiano.

Graph

Se define una clase para manejo de grafos, pero no se hace con la intención de ser óptima, ya que el lenguaje carece de las características necesarias para resolver problemas de este campo con el rendimiento necesario. Pese a esto, es necesario mencionar que la librería es completamente funcional y que incluye algún algoritmo común como el algoritmo de Dijkstra.

La clase está definida de la siguiente manera:

```
Define class "Graph" as
    vertices : Container
    edges : Container
```

Se permite la construcción de grafos mediante un constructor, pero lo más seguro es utilizar la función *emptyGraph* y añadir los vértices y aristas de manera manual con la interfaz proporcionada. Se dan algoritmos para generar grafos aleatorios.

Otra característica interesante de esta librería es la capacidad para exportar los grafos en formato dot (función miembro *toDot*) para su posterior renderización con programas como *graphviz*.

Dice

Esta librería no es demasiado importante, pero provee al usuario de una manera relativamente intuitiva de generar números aleatorios. Define una clase llamada Dice que modela una serie de tiradas de dado, teniendo cada uno un número determinado de caras. También se definen sintaxis que pueden ser utilizadas para crear estos objetos de manera sencilla y sin depender de constructores:

```
Define class "Dice" as
    rolls : Number
    sides : Number

Define syntax for Dice from "Arg(1{d}, Number, rolls) 'D' Arg(1{d}, Number, sides)" as //Sintaxis de la clase
    return _Dice_(args["rolls"], args["sides"])

Define cast String(dice : Dice) as
    return dice.rolls() + "D" + dice.sides()

Define const function Dice::roll() as //Simular tiradas
    res = 0

    for i in range(self.crolls())
        res += rand(1, self.csides(), true)

    return res

Define syntax from "'<' Arg(<Dice>, Dice, dice) '>'" as //Sintaxis para tirar dado directamente
    return args["dice"].roll()
```

Algoritmos

A parte de las librerías ya mencionadas, se definen varias otras que extienden las funcionalidades del lenguaje añadiendo algoritmos de uso común. A continuación se mencionan algunos ejemplos.

ContainerAlgorithms

Esta librería provee al usuario de numerosos algoritmos aplicables a cualquier objeto de tipo Container, por lo que es una de las librerías más importantes del lenguaje. Se dan sobrecargas para algoritmos tan comunes como el máximo y el mínimo con criterio opcional, acumulación con expresión funcional y elemento nulo, filtros o mapeos.

Se puede observar que los algoritmos proporcionados por esta librería han sido utilizados en algunos de los ejemplos anteriores.

Point

Esta librería también es de gran importancia, ya que define una clase llamada Point que permite manejar gráficos de manera mucho más cómoda que con coordenadas directas, ya que las sobrecargas se encargan de redondear los diferentes campos de cada punto y llamar a las funciones básicas correspondientes.

Cabe destacar que se incluyen transformaciones básicas con implementaciones optimizadas para dos dimensiones. Esto incluye algoritmos tales como traslaciones, rotaciones y simetrías.

Search

Esta librería es la utilizada para definir algoritmos de búsqueda con diferentes estrategias, así como con diferentes criterios de parada. Se puede ver su uso en el juego *tres en raya* definido en un apartado anterior.

Se definen algoritmos de búsqueda en anchura, búsqueda en profundidad, A* y sus respectivas versiones por intensión (no se busca un estado concreto, sino uno que cumpla una determinada condición).

Windows

Esta librería proporciona una interfaz de acceso a comandos de Windows mediante el uso de la función básica *system* con numerosos algoritmos de reconocimiento de patrones aplicados a los argumentos.

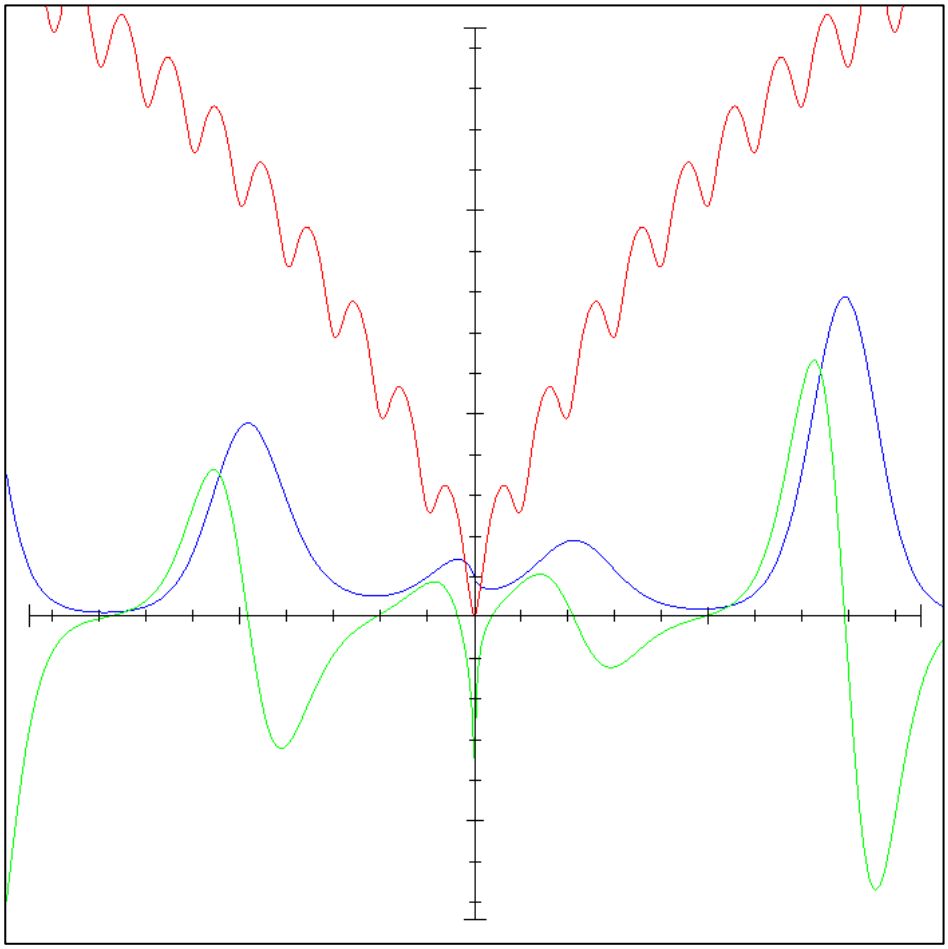
Se proporcionan funciones para renombrar archivos, copiarlos y demás comandos básicos aplicables en una consola *cmd*. También se proporciona una función genérica de comandos llamada *winCommand* que separa los argumentos de manera automática.

Representación de datos

Perspectiva analítica

Representación de funciones de una variable

Utilizando las funciones de representación de expresiones simbólicas podemos obtener gráficos interesantes:

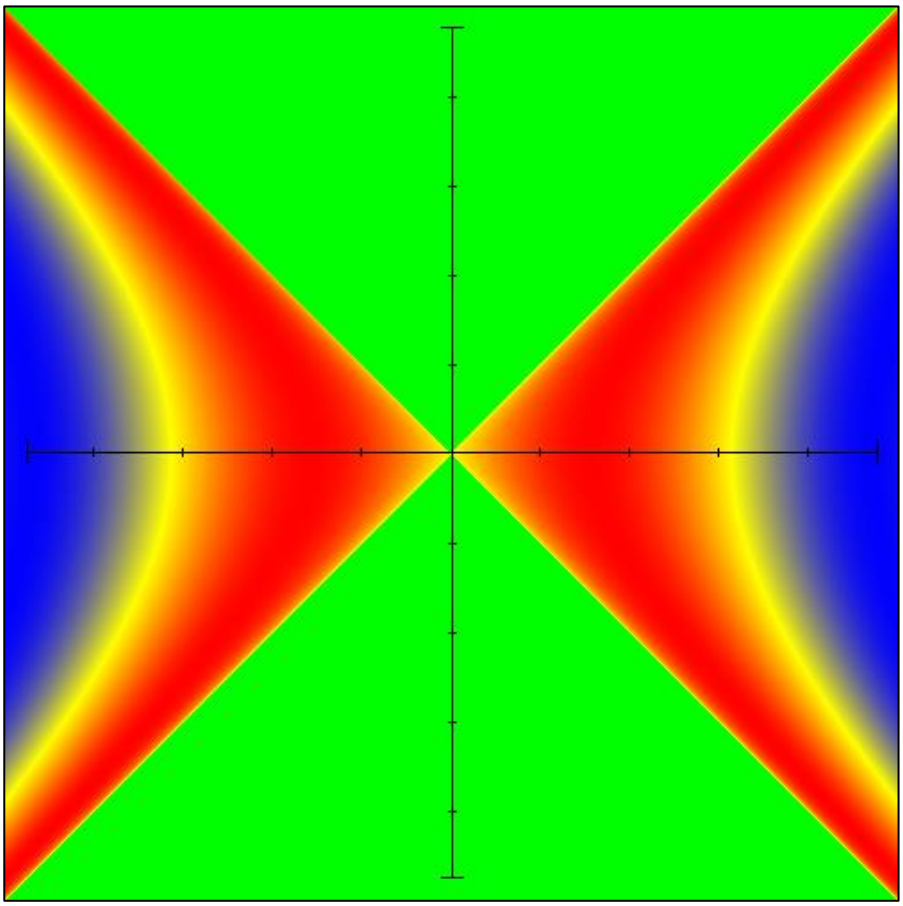


Siendo la correspondencia de colores la siguiente:

- **Rojo:** *función de Ackley* para $y = 0$
- **Azul:** $y = |x|^{\sin(x)}$
- **Verde:** derivada de la función anterior (calculada con función *diff*).

Representación de funciones de dos variables

Utilizando la función de representación de expresiones simbólicas de dos variables mediante el uso de un código de colores (*colorPlot*) podemos obtener gráficos bastante informativos. A continuación se muestra un ejemplo:



Este gráfico es el resultado de utilizar esta función para representar la siguiente función en el rango $[-5,5] \times [-5,5] \times [-1,1]$:

$$f(x,y) = \sin\left(\sqrt{x^2 - y^2}\right)$$

El código de colores elegido es el estándar:

- **Números positivos:** rojo.
- **Números negativos:** azul.
- **Cero:** amarillo.
- **Fuera del dominio:** verde.

Gráfica de bifurcación

En ámbitos relacionados con la biología es posible que se intente modelar el comportamiento de una población de individuos. Normalmente estas suelen seguir patrones complejos, pero existen maneras de simplificar estos patrones utilizando funciones iterativas basadas en polinomios cuadráticos. Un claro ejemplo es la siguiente función:

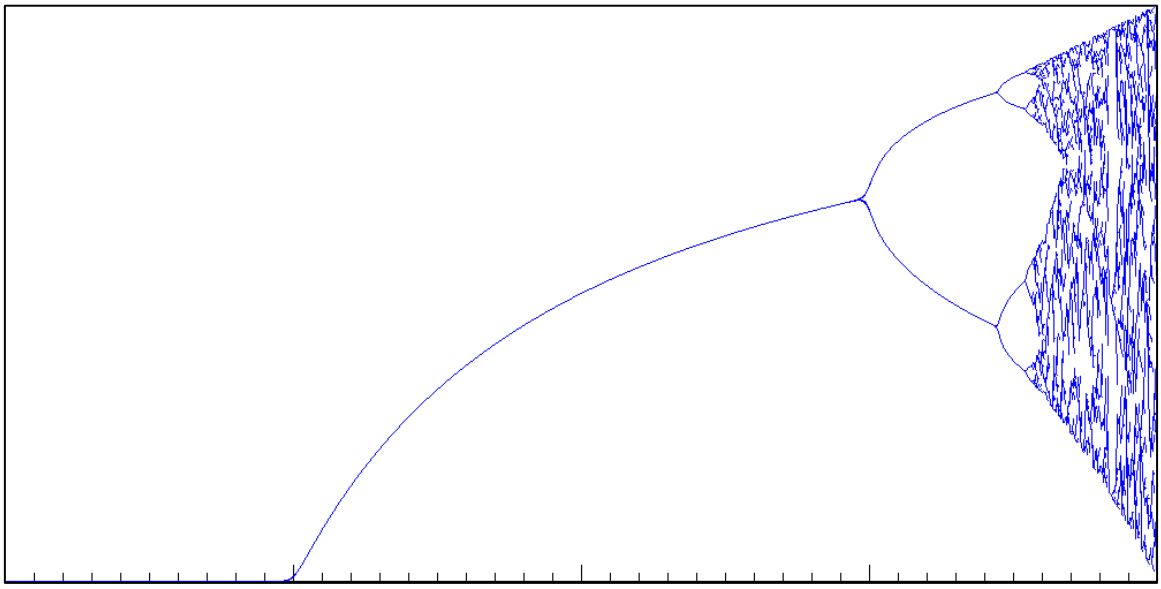
$$x_{i+1} = \lambda \cdot x_i \cdot (1 - x_i)$$

Siendo λ un *coeficiente de fertilidad*, podemos elegir una x_0 arbitraria y empezar a iterar para ver qué ocurre. Sorprendentemente, la función siempre acaba haciendo un bucle entre un número finito de valores distintos. Es posible extrapolar este comportamiento al de una población de individuos, ya que se pueden observar hechos tales como la extinción de los individuos por fertilidad baja:

$$\lim_{i \to \infty} x_i \approx 0 \; para \; \lambda < 1$$

Ahora bien, los ciclos anteriormente mencionados también pueden modelar comportamientos interesantes, por lo que viene bien utilizar un tipo de gráfica especial llamada gráfica de bifurcación para analizar estos patrones. Se ha realizado un programa en Ulan que dibuja exactamente este tipo de gráfica para la función expuesta anteriormente.

A continuación se muestra la gráfica para λ entre 0 y 4, siendo los valores del eje vertical entre 0 y 1:



Se puede observar que para $\lambda > 3.5$ puede ser difícil discernir los valores de la función. Esto se debe a que la distancia entre bifurcaciones sigue una serie geométrica, por lo que el gráfico cada vez es más denso y no renta subir la precisión numérica, ya que tan solo serviría para alguna iteración más. Los cálculos en este ejemplo se hicieron con 20 decimales.

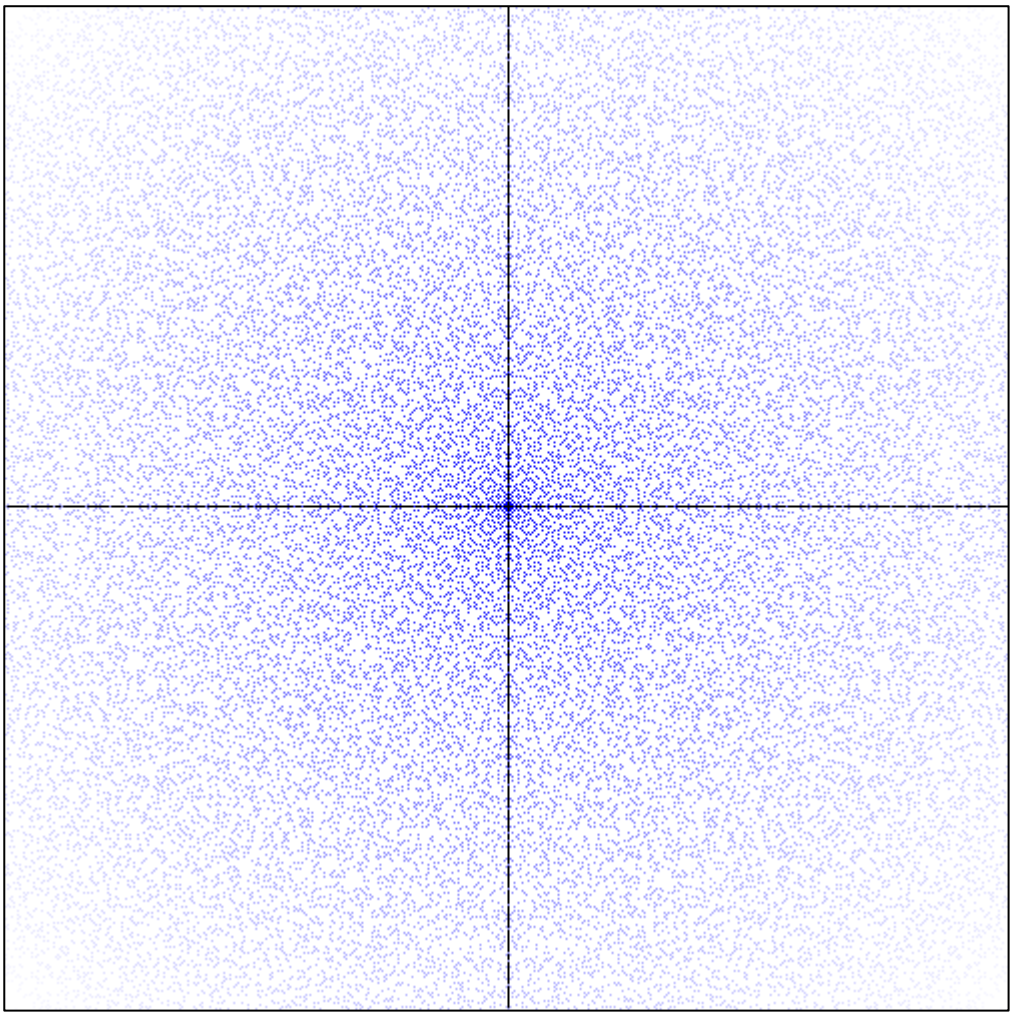
Primos gaussianos

Siguiendo la misma lógica que con los números reales, los números complejos pueden ser factorizados siguiendo unas normas parecidas. El único problema que tiene esto es que es necesario definir cuáles son los números primos que pueden conformar la factorización de un número complejo.

Para esto se define un primo gaussiano como un número $z = a + bi$ tal que:

$$\begin{cases} |a|^2 + |b|^2 \ es \ primo & si \ a \neq 0 \ y \ b \neq 0 \\ |a| \ es \ primo & si \ a \neq 0, b = 0 \ y \ a = 3 \ mod \ 4 \\ |b| \ es \ primo & si \ b \neq 0, a = 0 \ y \ b = 3 \ mod \ 4 \end{cases}$$

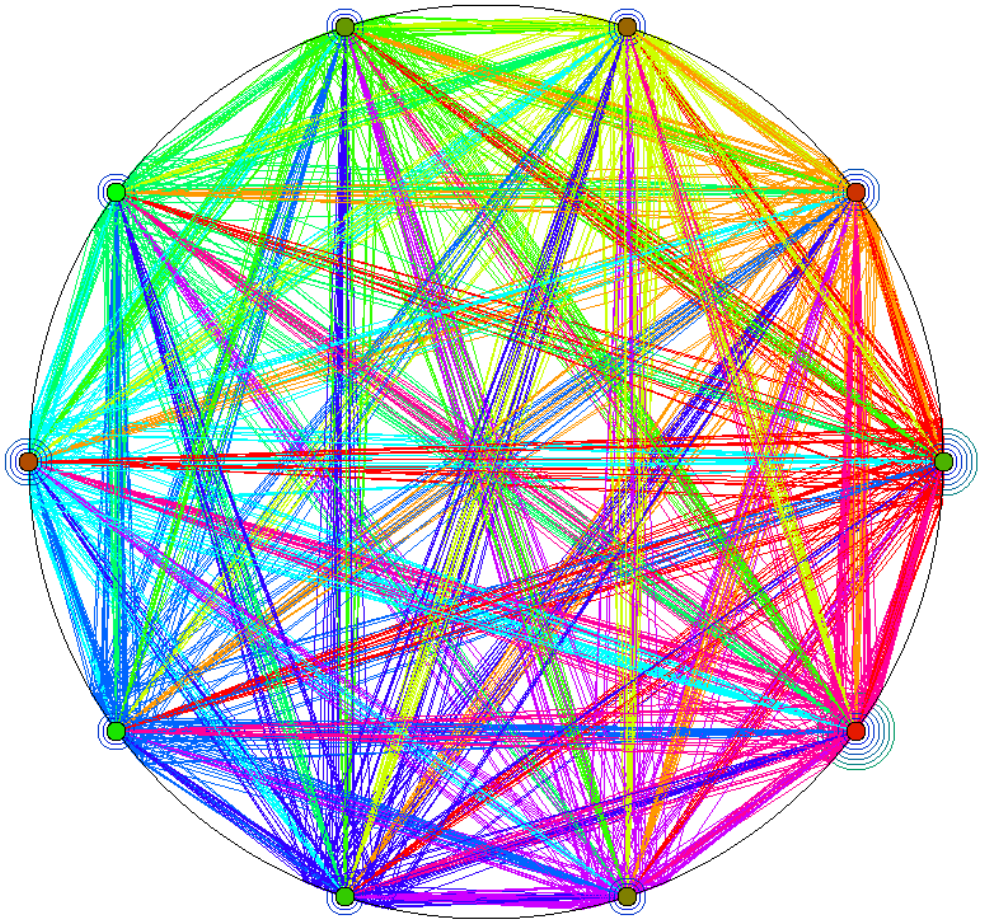
Es común representar estos números en un gráfico bidimensional, ya que los patrones que forman son agradables visualmente. A continuación se muestra una imagen de estos primos gaussianos representados en el intervalo $[-200, 200] \times [-200, 200]$ con un color degradado a blanco según su módulo:



Perspectiva artística

Representación gráfica de decimales de pi

Hacer representaciones artísticas utilizando los datos sobre la recurrencia de los dígitos decimales del número π no es algo poco común. En este caso se ha optado por una perspectiva común en una circunferencia:



Este gráfico se genera recorriendo los dígitos decimales de π en orden, dibujando una línea desde un punto de la circunferencia hasta otro, siendo estos escogidos rotando un vector con un ángulo igual a $\frac{2\pi}{10} \cdot n$, siendo n el dígito actual. El color de la línea se elige dependiendo del dígito actual, ya que cada uno tiene asignado un color con la función *rainbow*.

Los puntos que se pueden observar en la circunferencia tienen un color elegido en un gradiente de verde a rojo según su frecuencia absoluta en los dígitos calculados. Las circunferencias que los rodean representan la secuencia más larga de dígitos iguales consecutivos de entre los calculados.

Representación de conjuntos de Julia

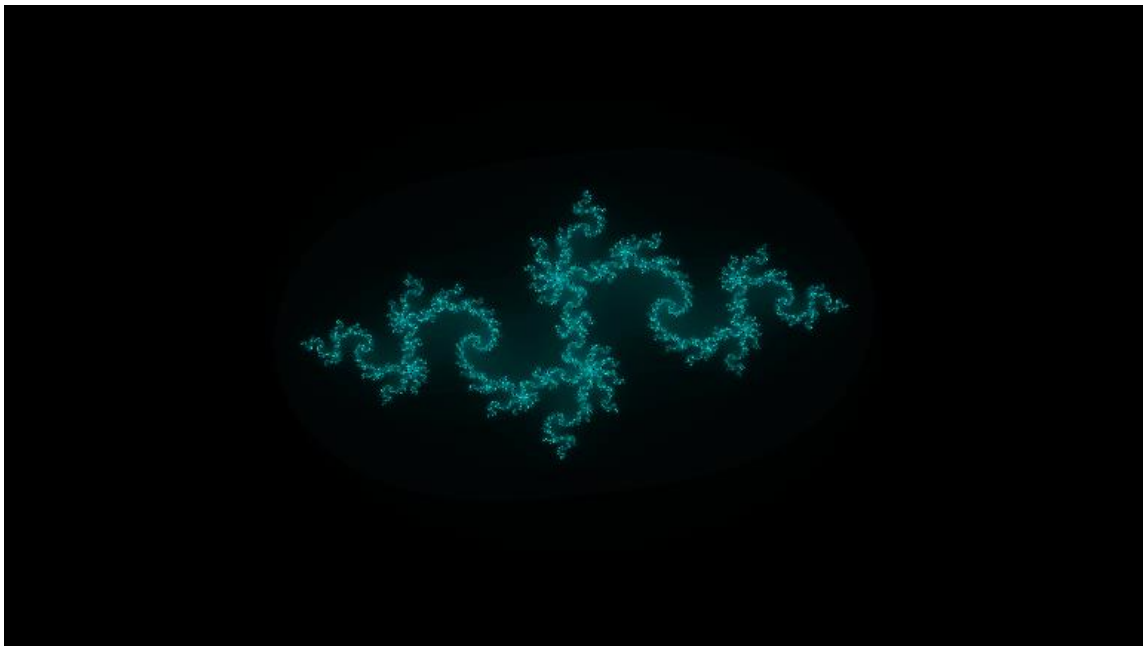
Siendo informales, los conjuntos de Julia son una serie de fractales que se generan al aplicar ciertas funciones a números complejos de manera iterativa. Normalmente estas funciones suelen tener la siguiente forma:

$$f(z) = z^n + c$$

$$z_{i+1} = f(z_i)$$

Cuando tras hacer un número fijo de iteraciones el número resultante tiene un módulo menor que una cierta constante elegida de antemano, se puede colorear ese punto del color de fondo. En el caso contrario se colorearía dependiendo del número de iteraciones que fueron necesarias para que su módulo sobrepasara el límite establecido.

Utilizando estas funciones especificadas anteriormente se generó el siguiente gráfico mediante la ejecución de un programa Ulan:



Los parámetros utilizados fueron los siguientes:

$$n = 2$$

$$c = -0.835 - 0.2321i$$

Encuadre:

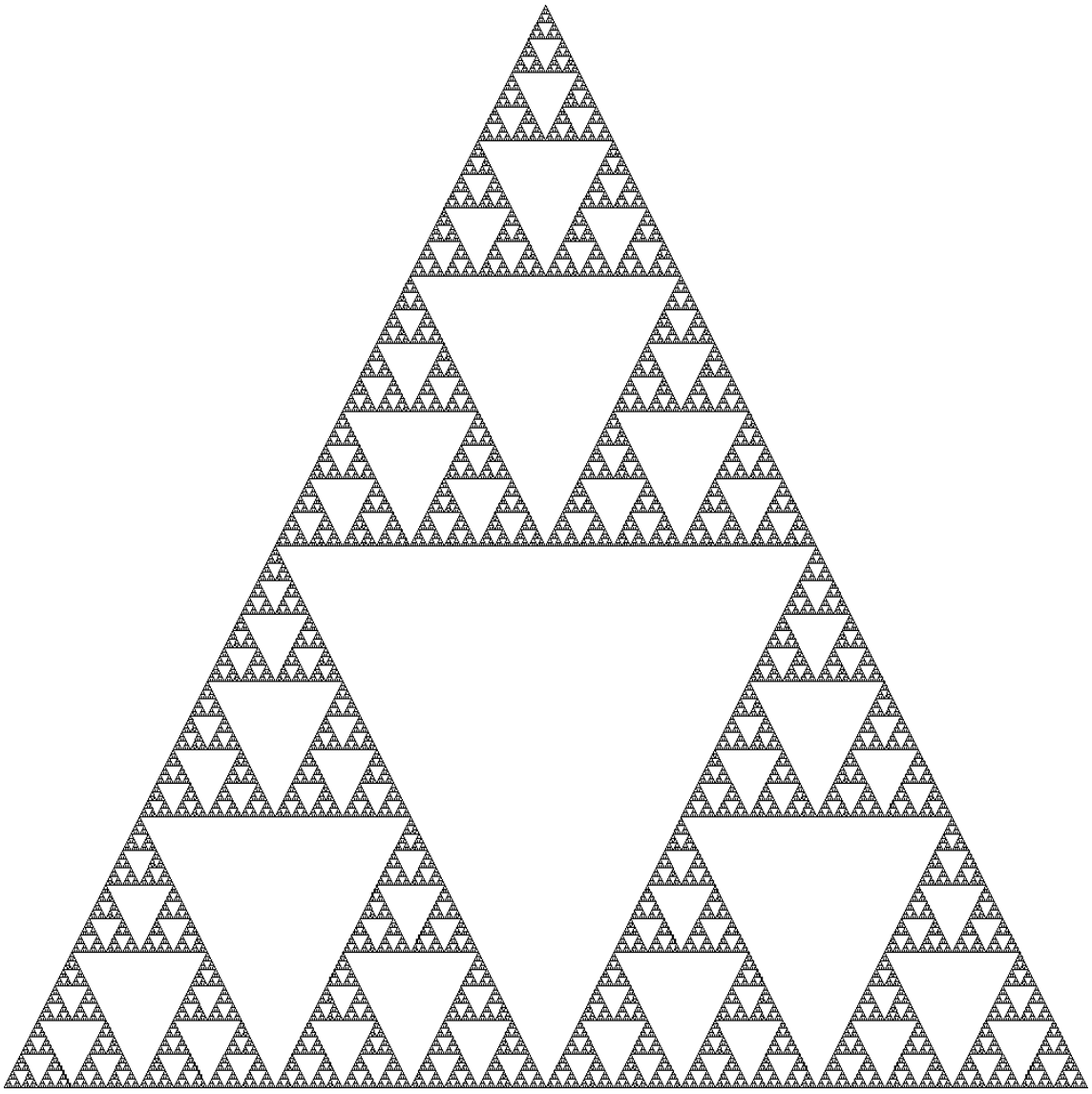
$$x \in [-3.2, 3.2]$$

$$y \in [-2, 2]$$

Triángulo de Sierpinski

El triángulo de Sierpinski es uno de los fractales más famosos por su sencillez y apariencia. La manera de construirlo consiste en dibujar un triángulo y calcular los puntos medios de cada arista. Estos puntos formarán otro triángulo a dibujar. Tras esto se ha dividido el triángulo original en 4. Se repite el proceso indefinidamente para los triángulos de los extremos.

El gráfico resultante con profundidad 9 es el siguiente:



Sección VI

Conclusiones

Sobre el proyecto

El resultado del proyecto ha sido definitivamente positivo, ya que se ha obtenido un lenguaje completamente funcional y útil desde varias perspectivas. Además, se han conseguido explorar unos conceptos interesantes para el campo de desarrollo de lenguajes de programación que se tendrán en cuenta a nivel personal en el futuro.

El interés a nivel teórico de Ulan viene de su capacidad para extender los conceptos más básicos de la sintaxis de manera eficiente y sin necesidad de utilizar paradigmas discutiblemente simplistas como programación orientada a pilas (siendo lenguajes como *Forth* [16] y *Factor* [17] claros ejemplos). Esta tarea ha sido completada de manera satisfactoria con la simplicidad que se buscaba desde un principio, por lo que podría decirse que, en este aspecto, el proyecto ha sido un éxito.

Como aspecto final remarcable, podría mencionarse que, al contrario de lo que a priori pudiera parecer, el código escrito en este lenguaje tiene una legibilidad notable siempre que se sigan normas de escritura consistentes y razonables (ver validación y resultados), disipando así una de las mayores preocupaciones que se tenían sobre el lenguaje. Si unimos esto con la estabilidad del producto final obtenemos un lenguaje de programación con una funcionalidad a tener en cuenta.

A nivel personal

Esta no ha sido la primera experiencia que he tenido con un proyecto extenso, pero sí ha sido el más grande de todos ellos. Esto es cierto tanto a nivel de código como a nivel de planificación. La tarea de diseñar un lenguaje de programación completo no es baladí, pero si a esto se le suma un lenguaje interno de extracción de datos y el hecho de que el paradigma de programación de conceptos no está demasiado extendido, era necesario ser especialmente cuidadoso con la gestión del proyecto.

Debido a esto, ha sido necesario tener un especial cuidado con la selección de características a añadir, así como tener muy clara tu productividad individual y conocer tus capacidades. Este proyecto ha servido como un ejercicio de introspección y autoanálisis que me ha ayudado a tener mucho más claro de lo que soy capaz. Además, me ha dado ese "empujón" al darme cuenta de que soy capaz de construir un proyecto interesante desde los cimientos. Asimismo, he adquirido unos conocimientos técnicos de gran valor.

A un nivel más teórico, ver cómo tu leguaje evoluciona desde su concepción hasta tener su primera versión estable es una experiencia que abre la mente y ayuda a ser más crítico con tu propio trabajo. Hacer una herramienta tan compleja con la intención de que la usen los demás es gratificante, pero es necesario superar el miedo a la crítica ajena.

A grandes rasgos, ha sido una experiencia enriquecedora en muchos aspectos que definitivamente me ayudará a ser un mejor profesional en el futuro.

Aspectos mejorables

Funcionalidad

Por lo general, la funcionalidad del lenguaje no solo ha llegado hasta el punto esperado desde un principio, sino que ha superado las expectativas en muchos aspectos. Un claro ejemplo de esto son las librerías básicas, que nacieron de la propia necesidad de tener una base de código con la que probar el lenguaje.

Si hubiera que mencionar alguna funcionalidad que se eche en falta, quizás sería el tratamiento con iteradores de Containers. Se considerará añadirlos en futuras versiones para abrir paso a un tratamiento de rangos mucho más eficaz y para abrir paso a un tratamiento funcional de listas.

Rendimiento

El objetivo principal de Ulan no es tener un rendimiento excepcional. Esta es la razón principal por la que todavía no se implementó el algoritmo de compilación. Dicho esto, el rendimiento de los programas descritos en la sección de rendimiento es más que aceptable en la mayoría de los casos. Aun así, se podrían mejorar los mecanismos de abstracción del lenguaje en un futuro para permitir usos más profesionales.

Posibles extensiones

Debido a las restricciones inherentes a un trabajo de final de grado, no se podrán implementar todas las características que se han planeado, por lo que tampoco aparecerán en los siguientes apartados del documento. Algunas de las características descartadas para la primera versión son las siguientes:

- **USDL multinivel:** esta es la característica descartada más importante, ya que se pretendía que se pudiese definir cualquier concepto semántico, pero en la descripción dada habrá huecos. Un lenguaje de descripción multinivel permitiría poder asignarle un significado (semántica) a cualquier árbol de sintaxis abstracta que se desease.
- **Programación por patrones:** reutilizando el lenguaje anterior se podría implementar una sintaxis y estructuras para poder definir funciones mediante reconocimiento de patrones. Un claro ejemplo de esta manera de programar sería Erlang [18].
- **Compilación completa:** actualmente se ofrece un intérprete que realiza una ejecución en tiempo real de código Ulan sin procesado previo. Se pretende implementar un compilador a C++ en un futuro.

Bibliografía

[1] The Glassgow Haskell Compiler - Inlining. [Online]. https://ghc.haskell.org/trac/ghc/wiki/Inlining

[2] Haskell official website. [Online]. https://www.haskell.org/

[3] Wolfram Language official website. [Online]. https://www.wolfram.com/language/

[4] Christophe de Dinechin. (2008) Concept Programming Metrics. [Online]. http://xlr.sourceforge.net/concept/metrics.html

[5] Christophe de Dinechin. XL Language official website. [Online]. http://xlr.sourceforge.net/

[6] Bjarne Stroustrup, *The C++ Programming Language*., 1985.

[7] The C++ Standards Comitee, *The C++ Standard*., 2017.

[8] Code:Blocks official website. [Online]. http://www.codeblocks.org/

[9] CppCheck official website. [Online]. http://cppcheck.sourceforge.net/

[10] Doxygen official webpage. [Online]. http://www.stack.nl/~dimitri/doxygen/

[11] Graphviz official website. [Online]. http://www.graphviz.org/

[12] draw.io. [Online]. https://www.draw.io/

[13] Sublime Text official website. [Online]. https://www.sublimetext.com/

[14] Marco Bodrato. (2007) Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. [Online]. http://www.bodrato.it/papers/#WAIFI2007

[15] Oliver Robert J. Lemke and Kannan Soundararajan. (2016) Unexpected biases in the distribution of consecutive primes. [Online]. https://arxiv.org/abs/1603.03720

[16] FORTH, Inc. official webpage. [Online]. https://www.forth.com/

[17] Factor official webpage. [Online]. https://factorcode.org/

[18] Erlang Reference Manual. [Online]. http://erlang.org/doc/reference_manual/patterns.html