UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN

Desarrollo de un prototipo robot humanoide que busque, encuentre y patee una pelota

Por:
Jennifer Dos Reis De Dobrega
Juliana Leon Quinteiro

Realizado con la asesoría de:
Ivette Carolina Martinez

PROYECTO DE GRADO
Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero en Computación

Sartenejas, Noviembre 2014

Resumen

AQUI DEBE IR EL RESUMEN

DEDICATORIA

Agradecimientos

AGRADECIMIENTOS

Índice general

Índice de cuadros

Índice de figuras

Introducción

Ya existen aproximaciones a este enfoque, la mayoría relacionados con: el diseño gráfico como la herramienta *Brain Storm* del programa *After Effects* de *Adobe* [**?**], que muestra variaciones de los elementos en pantalla para sugerir posibles modificaciones; la inteligencia artificial, como la herramienta Gecode [**?**], que permite resolver problemas algebraicos y lógicos para obtener rangos de soluciones posibles; y el desarrollo de videojuegos, como los juegos de la saga *Diablo, Minecraft* [**?**], o *Dwarven Fortress* [**?**], en los que los mapas son generados de forma aleatoria pero mantienen coherencia.

Incluso hay otro tipo de herramientas o que guardan bastante similitud a esta idea. Entre ellas destaca *QuickCheck* [**?**], que es una herramienta en *Haskell* que permite generar casos de prueba de estructuras que se han descrito. Sin embargo, tiene como limitación que sólo está hecha para realizar pruebas y no generar soluciones de salida para aplicaciones en produción.

En el presente trabajo de investigación se explica el proceso y diseño de la herramienta de generación de estas instancias aleatorias. Partiendo en el desarrollo de un lenguaje de programación propio (sección **??**), para luego ahondar en el proceso de resolución de problemas y asignación de valores (sección **??**). Por último se evaluarán los resultados de las pruebas hechas sobre la herramienta y se concluirá acerca de sus posibles mejoras y enfoques para trabajos futuros (sección **??**).

Capítulo 1

Marco teórico

En este capítulo se presentan los conceptos necesarios sobre robótica aplicados en el presente trabajo, posteriormente descripcion basica de aprendizaje de máquinas, así como visión artificial para detección de objetos.

1.1. Robotica

Para definir un lenguaje formal se requiere describir:

- **Robot:** Un agente artificial, activo, cuyo entorno no es el mundo físico. El término activo descarta de esta definición a las piedras, el término artificial descarta a los animales, y el término físico descarta a los agentes de software puros o softbots, cuyo entorno lo constituyen los sistemas de archivos, bases de datos y redes de computo. LIBRO Russell and NORVIG pag 815
- **Robótica:** Es la rama de la tecnología que se encarga del diseño, construcción, operación y aplicación de los robots.

- **Sensores:** Son los encargados de percibir el ambiente que rodea al robot. Según Murphy R.R son dispositivos que miden algún atributo del mundo. Un sensor recibe energía del entorno (sonido, luz, presión, temperatura, etc) y transmite una señal a una pantalla o computador ya sea de forma análoga o digital. (introduction to AI robotics pag 196)

- **Actuador:** Es aquella parte del robot que convierte comandos de software en movimientos físicos. AI Russell and Norving pag 777

- **Servomotor:** Es un motor eléctrico, considerado como actuador, que permite ser controlado tanto en velocidad como en posición.

- **Giroscopio:** Es un sensor utilizado para medir y mantener la orientación, se mide a través del momento angular.

1.2. Robótica Inteligente (Agentes Inteligentes)

1.2.1. Paradigmas de robótica

En la robótica inteligente, según Robin Murphy, existen tres paradigmas en los cuales se clasifica el diseño de un robot inteligente, estos paradigmas pueden ser descritos de dos maneras: la relación entre las primitivas básicas de la robótica, percibir, planificar, actuar o de la forma en que los datos son percibidos y distribuidos en el sistema.

Percibir se refiere al procesamiento útil de la información de los sensores del robot. Planificar, cuando con información útil, se crea un conocimiento del mundo y se generan ciertas tareas que el robot podría realizar. Por último actuar consiste en realizar la acción correspondiente con los actuadores del robot para modificar el entorno.

1.2.1.1. Paradigma Jerárquico

Este paradigma es secuencial y ordenado. Primero el robot percibe el mundo y construye un mapa global. En base al mapa ya percibido y con "los ojos cerrados", el robot planifica todas tareas necesarias para lograr la meta. Luego ejecuta la secuencia de actividades según la planificación realizada. Una vez culminada la secuencia se repite el ciclo percibiendo el mundo, planificando y actuando. (pag 6)

PONER IMAGEN

1.2.1.2. Paradigma Reactivo

El paradigma reactivo omite por completo el componente de la planificación y solo se basa en percibir y actuar. El robot puede mantener un conjunto de pares percibir-actuar, estos son llamados comportamientos y se ejecutan como procesos concurrentes. Un comportamiento toma datos de la percepción del mundo y los procesa para tomar la mejor acción independientemente de los otros procesos. (pag 8)

Estas tareas se encuentran muy relacionadas en el comportamiento, y todas las actividades emergen del resultado de dicho comportamiento en secuencia o concurrentemente.

1.3. Inteligencia Artificial

La inteligencia artificial es un término relacionado con la computación y la robótica que ha tenido varias definiciones, ocho de ellas, las cuales nacieron a finales del siglo XX, se encuentran organizadas en (norving and Russell) bajo cuatro categorías: pensar y actuar de forma humana, pensar y actuar de forma racional. Con ello se puede entender que la inteligencia artificial tiene que ver con lograr que un robot resuelve problemas de manera inteligente, es decir, de manera que parezca que el razonamiento y comportamiento humano

las ha resuelto.

1.3.1. Aprendizaje de Máquinas

El aprendizaje de máquinas es un área de la inteligencia artificial que está relacionada con la pregunta de cómo construir programas de computadora que automáticamente mejoren con la experiencia. Se dice que un programa aprende de la experiencia E con respecto a una tarea T y desempeño P. Si el desempeño en la tarea T, medido por P, mejora con con la experiencia E (Mitchell)

1.3.2. Aprendizaje por reforzamiento

El aprendizaje por reforzamiento es un tipo de aprendizaje de máquinas que se basa en un sistema de recompensas y penalizaciones. Las recompensas se pueden dar en cada estado o una sola vez al llegar al estado final.

El objetivo del agente es aprender de las recompensas para escoger la secuencia de acciones que produzca la mayor recompensa acumulada. (Mitchell)

El agente existe en un entorno descrito por algunos estados S. Puede ejecutar un conjunto de acciones A. Cada vez que ejecuta una acción 'at' en algún estado 'at' el agente recibe una recompensa 'rt'. El objetivo es aprender una política pi : S -¿ A que maximice la suma esperada de esas recompensas con descuento exponencial de las recompensas futuras. (Mitchell) El resultado de tomar las acciones puede ser determinista o no, en el caso de este proyecto no es deterninista, es decir, existen porcentajes de probabilidad de pasar a un estado u otro al tomar una acción en un estado en particular.

1.3.3. Q- learning

Es un método de aprendizaje por reforzamiento

Compara las utilidades esperadas de acciones posibles sin necesidad de saber el resultado por tanto no se necesita un modelo del entorno (Russell)

ACTIVE REINFORCEMENT LEARNING pag 839 russell

Visión de computadoras

1.4. Visión Artificial

Concepto Vision de computadoras (IMPORTANTE). En la inteligencia artificial es importante el reconocimiento de patrones y clasificación de objetos. El reconocimiento de objetos

1.4.1. Filtros

El filtrado de imágenes es una técnica para la transformación de imágenes, que consiste en destacar sus características más relevantes orientadas a un propósito en particular.

Generalmente en la tarea de extracción de información de una imagen se utilizan filtros para descartar zonas o características que no son importantes para el patrón deseado y para determinar el área deseada ya sea por patrones de forma o color.

En la investigación, los algoritmos de filtrado aplicados a las imágenes fueron: Clausura Morfológica y Apertura Morfológica, filtros que aplican las técnicas de erosión y dilatación a las imágenes.

1.4.2. Transformaciones Morfológicas

Las transformaciones morfológicas básicas son llamadas dilatación y erosión, y si Image Morphology que se presentan en una amplia variedad de contextos como la eliminación del ruido, aislamiento de elementos individuales, elementos de unión dispares en en una imagen.

1.4.2.1. Dilatación

La dilatación es una convulsión de alguna imagen (o región de una imagen) , que llamaremos A, con un núcleo que llamaremos B, el núcleo que puede ser de cualquier forma o tamaño, tiene un solo punto de anclaje definido. Muy a menudo, el núcleo es un pequeño cuadrado o disco sólido con el punto de anclaje en el centro. El núcleo puede ser pensado como una plantilla o mascarilla, y su efecto es que para la dilatación de un operador de máximo local sobre la imagen, se calcula el valor de píxel máximo común a B y reemplazamos el píxel de la imagen en el punto de anclaje con ese valor máximo. Esto causa regiones brillantes dentro de una imagen y la hacen crecer. Este crecimiento es el origen del término .ºperador de dilatación".

IMAGEN

1.4.2.2. Erosión

La erosión es la operación inversa a la dilatación. Esta acción del operador es equivalente a la erosión el cálculo de un mínimo local sobre el área del núcleo. La erosión genera una nueva imagen desde la original, utilizando el siguiente algoritmo: como el núcleo B es analizado sobre la imagen, se calcula el mínimo valor del pixel superpuesto por B y se reemplaza el pixel de la imagen con un punto de anclaje de valor mínimo.

IMAGEN

Capítulo 2

Presentación del problema

¿De qué forma obtener resultados no determinísticos podría traer una ventaja? Ciertamente si un resultado es completamente aleatorio su utilidad es muy pobre y el cálculo sería considerado como un desperdicio. Sin embargo, se pudiera definir un objeto con ciertas propiedades y limitarlo para que cumpla con ciertas condiciones. Se tendría entonces una instancia obtenida aleatoriamente de un objeto cuya utilidad sería definida por su especificación y su constitución estaría bajo las limitantes que se establezcan.

Partiendo de esto, se ha planteado el diseño y desarrollo de una herramienta que se base en estos principios no deterministas,para generar soluciones variadas a este tipo de problemas pero que cuyas respuestas se mantengan al margen de la especificación proporcionada. Los beneficios de esta herramienta son muchos, entre ellos podría destacarse:

- Generación de casos de pruebas.
- Creación y llenado de bases de datos con ejemplos consistentes.
- Resolución de problemas.
- Computación gráfica.

- Computación especializada en creación de simuladores.
- Diseño y creación de videojuegos.

Todos los anteriores son los ejemplos de circunstancias más básicas que podrían atacarse con esta herramienta. Dicho esto queda más claro que el objetivo general es diseñar e implementar esta herramienta. De forma que se pueda generar instancias aleatorias de objetos basadas en una definición y cuyos valores satisfagan ciertas condiciones y restricciones definidas como entrada.

Mientras que como objetivos parciales o específicos se postulan los siguientes:

- Diseñar y elaborar un lenguaje de programación que facilite especificar un modelo a crear, incluyendo su estructuración, así como sus restricciones internas.
- Implementar un reconocedor sintáctico escrito en *C++* utilizando las herramientas *Flex* y *Bison*.
- Elaborar un compilador para el lenguaje que resuelva los problemas asociados a restricciones y asocie los valores posibles a las variables.
- Desarrollar un interpretador que sirva como auxiliar al compilador y permita dividir el proceso en partes para facilitar el cálculo cuando se requiere para generar múltiples instancias de un mismo objeto.
- Estudiar las posibles formas de optimizar la herramienta para aumentar su velocidad y mejorar su uso de memoria.
- Investigar y proponer formas de ampliar las funcionalidades de la herramienta y mejorar las ya existentes de forma que trabajos futuros tengan sólidas referencias y guías.

Capítulo 3

Definición de lenguaje

Amorfinator es un lenguaje desarrollado con la finalidad de facilitar la generación de instancias aleatorias de estructuras de datos. Por ende, está diseñado para proporcionar a sus usuarios una sintaxis intuitiva que permita especificar la estructura final deseada, así como las funciones para verificar u obtener valores y también las estructuras auxiliares que sirven como herramientas para obtener los resultados finales.

La sintaxis fue diseñada para aceptar palabras reservadas escritas tanto en ingles como en español, incluso en ambos idiomas al mismo tiempo.

3.1. Estructura general

La estructura general se basa en la idea de centrar toda la importancia en una única estructura de salida, que contendrá la definición del objeto deseado a generar. Para facilitar esto, se decidió diseñar una estructuración por bloques que separen la salida final de otras instancias auxiliares y funciones. Estos bloques son los siguientes:

3.1.1. Estructura de la salida

Esta es la estructura principal que describe cómo se quiere recibir el objeto generado por el lenguaje. Este bloque comienza con la palabra `salida` (`exit`) seguido del nombre de la estructura y luego un bloque con el contenido. Este bloque está entre un conjunto de llaves ({ y }) y está formado por dos partes:

3.1.1.1. Descripción

Esta parte contiene todos los atributos de los objetos (variables), sus valores o "comportamientos". De esto último se explicará en la sección de variables **??**.

Sintaxis: Su sintaxis es `descripcion` (`description`) seguido del bloque con las variables.

Ejemplo:

```
1 descripcion{
2   int a0 ~ {
3     #> 0;
4     #< 25;
5     # %2 == 0
6   };
7   int a1;
8   list<int> l0;
9 }
```

Figura 3.1: Ejemplo de descripción de objeto

El objeto de la figura **??** tiene:

- Un entero `a0` que es positivo, menor que `25` y es par.
- Un entero cualquiera `a1`.
- Una lista de enteros `l0` con una cantidad aleatoria de elementos.

3.1.1.2. Bloque de restricciones

Es el bloque que contiene el conjunto de restricciones para los atributos del objeto. En esta parte, es posible hacer referencia a varias variables del objeto en una restricción y restringir el comportamiento de las estructuras de datos no básicos del objeto.

Sintaxis: Su sintaxis es `restriccion` (`restriction`) seguido del bloque con las restricciones. Este bloque puede ser especificado como una expresión booleana en *sintaxis por delimitación* (*la sintaxis por delimitación* se explica más adelante en la sección **??**).

Ejemplo:

```
1 restriccion {
2   a > b + length(l0);
3   c ~ normal(a,b);
4 }
```

Figura 3.2: Ejemplo de bloque de restricciones

Este bloque de restricciones de la figura **??** significa que se quiere que el valor de a debe ser mayor que la suma del valor de `b` y el largo de la lista `l0`. Además se indica que se quiere que el valor de `c` tenga un valor aleatorio pero que este sea obtenido aleatoriamente bajo una distribución normal.

Para facilitar la lectura y edición de expresiones booleanas de gran tamaño. Se ha diseñado un tipo de sintaxis llamada *Sintaxis por Delimitación*. Esta se basa en agrupar las expresiones que operan todas bajo conjunciones incluyéndolas en un bloque de { } y las disyunciones entre [ ] y separadas por ;.

3.1.2. Estructuras auxiliares

Sirven para especificar estructuras que no serán retornadas por el lenguaje pero son de utilidad para realizar los cálculos necesarios para obtener los valores finales. Además también pueden formar parte del objeto de salida.

Sintaxis: Su sintaxis es similar a la de la estructura de salida y se diferencian en que en vez de tener como etiqueta `salida` (`exit`) tiene la etiqueta `auxiliar` (`aux`).

Ejemplo:

```
1 aux bar {
2   descripcion {
3     int a2;
4     string a3;
5   }
6 }
```

Figura 3.3: Ejemplo de estructuras auxiliares

Lo que quiere decir la figura **??** es que los objetos que sean del tipo `bar` tienen un entero `a2` y un string `a3` aleatorios.

3.1.3. Funciones

En este bloque se especifica el conjunto de funciones que son definidas por el usuario para realizar cálculos y minimizar la repetición de código. Dentro de una función se puede usar variables aleatorias usando la sintaxis del lenguaje.

Sintaxis: Para especificar un bloque de funciones se debe usar la etiqueta `funcion` (`function`) y luego dentro de unas llaves { } colocar las funciones separadas al final por un `;`. Una función dentro del bloque de funciones tiene la siguiente estructura:

```
1 < tipo >< nombre_funcion > ([< parametros > (, < parametros >)*])
2 [< variables_aleatorias >] =
3 < expresion >
4 |
5 if(< expresion_booleana >)then < expresion >
6 [elseif(< expresion_booleana >) < expresion >]+
7 else < expresion >
```

Figura 3.4: Sintaxis de funciones

Donde cada uno de las declaraciones en **??** significa:

- **tipo:** Es el tipo devuelto por la función, solo soporta tipos básicos.
- **nombre_funcion:** Es el nombre de la función.
- **parametros:** Son el tipo y nombre de la variable que recibe la función.
- **variables_aleatorias:** Son el conjunto de variables aleatorias que se desean tener en la función.
- **expresion:** Es la expresión que devuelve la función cuyo tipo debe ser el mismo que el de la función.
- **expresion_booleana:** Es una expresión que devuelve un valor booleano siempre.

Ejemplo:

```
1 funciones {
2   bool par (int i) = if (i %2 == 0) then true else false
3 }
```

Figura 3.5: Ejemplo de funciones

Aquí en la figura **??** se tiene la especificación de una función que recibe un número y verifica si es par, en caso de serlo devuelve `true` y en caso contrario devuelve `false`.

3.2. Tipos de datos

Los siguientes son los tipos de datos definidos en **Amorfinator**:

3.2.1. Datos Básicos

- **booleano (bool):** Admite solo los valores `true` y `false`.
- **entero (int):** Admite valores numéricos enteros (Z) entre -2^{31} y 2^{31}.

- **floatante (float):** Admite valores numéricos reales entre -1.18e-38 y 3.4e38.
- **caracter (char):** Representa caracteres del alfabeto UTF-8.

3.2.2. Datos Complejos

- **Double (Double):** Admite valores numéricos reales con mayor precisión y mayor rango de representación. Con números entre -2.23e-308 y 1.79e308.
- **Palabra (String):** Representa una cadena de caracteres.

3.2.3. Estructuras Básicas de datos

- **Vector2:** Representa una tupla de un mismo tipo de dos elementos.
- **Vector3:** Representa una tupla de un mismo tipo de tres elementos.
- **Vector4:** Representa una tupla de un mismo tipo de cuatro elementos.
- **Lista<*Tipo*>:** Representa una lista de un mismo tipo de datos. Esta también permite operaciones para los tipos Pila, Cola y Arreglo.

3.3. Expresiones

Una expresión es una pequeña estructura que comprende valores y operadores y que tiene un valor, este valor puede o no estar determinado al especificar la expresión dependiendo de si las variables que la componen están o no instanciadas. Permiten definir comportamientos de las variables, relaciones entre varias de ellas o simplemente calcular un valor.

Sintaxis: Una expresión puede ser un *string* complejo, una variable, acceso a una variable, una operación binaria de dos expresiones, una expresión puede estar dentro de uno más

paréntesis, puede tener un operador unario, una función que devuelva un valor, puede ser un numero, booleano, lista, caracter, variable `#`, vectores de 2, 3 y 4 dimensiones.

Ejemplos:

```
2 + (3 * sqrt(42))
[1,2,3,4]
"Hola Mundo"
length([1,2,3,4])
```

Figura 3.6: Ejemplo de expresiones

3.3.1. Expresiones Matemáticas

Son expresiones compuestas por varios operadores matemáticos de tipo operación y uno de tipo comparación. Es el equivalente a representar una ecuación algebraica. En este tipo de ecuaciones es posible utilizar variables o funciones que representen valores numéricos y el resultado de esta expresión es una expresión lógica que es la evaluación de el operador de comparación.

Sintaxis: La forma en que se escriben las expresiones matemáticas no es diferente a la forma en que se escriben en otros lenguajes de programación, simplemente se basa en un conjunto de valores que operan entre ellos mediante un conjunto de operadores matemáticos, algunos valores son desconocidos y son representados como variables.

Ejemplos:

- **5 + 3 > 2** En este caso el operador de comparación es el > y el resultado de evaluar esta expresión es `true` ya que el lado `5+3` da como resultado `8` y es mayor que `2`.
- **x == 34 / 2** Para esta ecuación el operador de comparación sería el signo `==`, dado que existe una ecuación y solo una variable, es posible obtener un valor único para la misma. En este caso el valor de `x` sería `17`. Luego de obtener el valor, la ecuación

se evaluaría como cierta (`true`). Si en el caso contrario no pudiese asignarse un valor a una variable porque todas las asignaciones contradicen otra expresión o restricción, entonces la expresión no podría resolverse y la solución general que se estaría verificando en el momento sería descartada. Existe también la posibilidad de que esta expresión esté bajo un contexto de negación, en cuyo caso la ecuación completa deberá retornar `false` y por ende el valor de `x` estaría siendo obligado a ser diferente de `14`. Esto sería equivalente a haber planteado la ecuación de la siguiente forma: `x != 34 / 2`.

- **2x - y == 2** En este ejemplo el operador de comparación que también se podría considerar principal es el signo `==`. En este caso la primera parte de la ecuación está formada por una operación con las variables $x\&y$. Esta ecuación no puede tener un valor booleano final hasta que se hayan conseguido los valores de ambas variables involucradas. Si se consiguiese el valor de una de ambas variables sería posible conseguir el valor de la otra utilizando un despeje por métodos matemáticos. De lo contrario los posibles valores soluciones de ambas variables formarían un conjunto infinito.

3.3.2. Expresiones Booleanas

Las expresiones booleanas representan un argumento lógico cuyos valores finales posibles son `cierto` o `falso` (`true` o `false`) y están compuestas por sub expresiones que devuelven un valor boleeano unidas por operadores unarios y binarios booleanos o funciones. Una expresión booleana podría estar fácilmente compuesta por expresiones matemáticas enlazadas por operadores lógicos. Un ejemplo de esto sería como el mostrado en la figura **??**.

```
1 (x + z == 10 && x > 5 && x - z == 4) || (x == 4 && z == 2)
```

Figura 3.7: Ejemplo de expresiones booleanas

Para este caso una sola expresión booleana permite declarar dos sistemas de ecuaciones

en los que solo hay una asignación posible para cada par de variables. En la primera parte el único valor para que la expresión puede resultar cierta es `x = 7` y `z = 3`, mientras que para la segunda los valores son evidentes. Ambos casos tienen la misma prioridad para ser considerados y dado que los valores posibles dan conjuntos disjuntos se podría obtener cualquiera de las soluciones pero no ambas al mismo tiempo.

Existe otra sintaxis creada para facilitar la escritura de las estructuras booleanas y es la que se trata en la sección **??**.

3.4. Variables

Las variables se asemejan a las existentes en C++, con la diferencia de que existe la posibilidad de especificar las restricciones sobre ellas. Para esto se crearon 4 formas posibles de declarar una variable:

1. Variable de valor constante, es decir una variable a la cual se le asigna el valor desde su declaración, usando el símbolo =.

2. Variable aleatoria por defecto, los detalles se encuentran en la sección **??**.

3. Variables con restricciones propias, en este caso se usa el símbolo $\sim$, los detalles de las restricciones se tratan en la sección **??**.

4. Variables por elección, este último usa los signos unidos $=\sim$ para poder expresar que la variable toma uno de los valores contenidos dentro de la lista. Los detalles se encuentran en la sección **??**.

3.4.1. Variable aleatoria por defecto

Esta forma de definición consiste en declarar una variable que tomará un valor de su tipo correspondiente de forma pseudoaleatoria por el lenguaje.

Sintaxis: La forma de definir una variable aleatoria por defecto es simplemente declarar una variable sin asignarle ningún valor.

Ejemplos:

```
1 int numero;
2 char letra;
3 Vector2 coordenadas;
```

Figura 3.8: Ejemplo de variable aleatoria por defecto

En cualquiera de estos casos de la figura **??** la variable recibirá un valor aleatorio siempre que no exista una restricción que la condicione o que la relacione con otro valor. En el caso de no estar asignada y no estar restringida una variable obtendrá un valor aleatorio obtenido automáticamente por el lenguaje según sus propias librerías.

3.4.2. Variables con restricciones propias

Esta forma de definición consiste en declarar una variable sin asignarle un valor al igual que en la forma anterior, pero restringiendo las posibles asignaciones de valores tanto como se quiera.

Sintaxis: La sintaxis de este tipo de definición variable necesita primero la especificación del tipo de variable, luego se define el nombre de la variable y finalmente se define un conjunto de restricciones que se quieren para esta variable. Este bloque se especifica así:

1. **∼:** Esto representa que la variable no esta instanciada sino que es aleatoria y tiene restricciones propias.

```
1 int par ~{
2   # >= 0;
3   # <= 30;
4   # %2 == 0;
5 };
```

Figura 3.9: Ejemplo de variable aleatoria con restricciones propias

2. Restricción: Es una expresión booleana o una función con retorno booleano. Solo puede involucrar a la misma variable que es la que se está definiendo. Para facilitar la escritura en este contexto se puede referir a esta variable con el símbolo `#`. En el caso de querer utilizar varias restricciones se abre un bloque definido por los símbolos { y } y las restricciones terminan con `;` que sirve también a modo de separación.

Ejemplo:

En la figura **??** la variable solo puede ser mayor o igual a `0`, menor que `30` y par.

3.4.3. Variables por elección

Estas variables no tienen un valor fijo asignado, y solo puede obtener alguno de los valores especificados.

Sintaxis: La forma de definir una elección es declarar un bloque de opciones. Esto se hace incluyendo dentro de un par de `[ ]` el conjunto de valores que puede tomar a variable separados por el símbolo |. Es posible especificar el porcentaje de probabilidad con el que se quiere que sea elegido el valor de entre los posibles valores del conjunto, esto se realiza especificando un número que equivale al porcentaje asignado al un valor, este porcentaje se escribe luego del valor y separado por espacios o tabulaciones.

Ejemplos:

Como se puede observar en la figura **??** en el último ejemplo es posible asignar un valor que represente el porcentaje de probabilidad que tiene una opción de ser elegida. En el caso

```
1 int foo =~ [ 4| 8| 15| 16| 23| 42 ];
2 Caracteres nombre =~ [ "Miguele"| "Jose"| "Kratrin"| "Juan" ];
3 Str clasifica =~[ "Excelente" 20%| "Bueno" 50%| "Malo" 29.5%| "Pesimo" ];
```

Figura 3.10: Ejemplo de variable por elección

de que la suma de todas las opciones de un mismo bloque sea menor a `100` también lanzará un error.

3.5. Restricciones

Son los elementos que permiten filtrar o especificar el comportamiento o los conjuntos posibles de valores que pueden ser asignados a las variables. Existen varias formas de especificar las restricciones:

- En primer lugar, hay restricciones que solamente involucran a una sola variable, mientras que hay otras que relacionan a varias. Para facilitar el orden y formato, las que involucran a solo una variable pueden escribirse en el bloque de definición de la misma y refiriéndose a esta variable con el signo `#`. Estas restricciones pueden también escribirse en el área de restricciones generales para la estructura si se desea.

- Por otro lado las expresiones que involucran a varias variables o a estructuras no básicas deben escribirse necesariamente en el área de restricciones generales.

Las restricciones también pueden clasificarse por el uso, tal y como se muestra en las siguientes subsecciones.

3.5.1. Restricciones booleanas

Estas restricciones vienen siendo expresiones boolenas como las que ya se explicaron, que contienen entre sus términos más de una variable. Sirven para establecer comportamientos

de dependencia entre variables.

3.5.2. Distribuciones estadísticas

Este tipo de restricción permite especificar que el valor de la variable se distribuye basado en una distribución estadística "estándar". Esto limita a que no pueda ser utilizada para verificar un valor, sino para asignar una instancia del conjunto de valores del dominio de la variable obtenida según la distribución.

Sintaxis: La forma de especificar que una variable se debe obtener de una función probabilística es con el símbolo `#` (dependiendo del contexto) seguido del símbolo $\sim$ y luego la función de distribución con sus parámetros correspondientes. Éstas funciones de distribución están especificadas en el lenguaje y corresponden a las distribuciones usuales como: Distribución normal, exponencial, gamma, etc.

Ejemplos:

```
1 nota ~ normal(13,4);
```

Figura 3.11: Ejemplo de variable restringida por distribución estadística

Este caso de la figura **??** representa una variable que se quiere que tenga un valor aleatorio pero que este dentro de los valores que comprende una distribución normal.

3.5.3. Cuantificadores

Permite expresar restricciones para los elementos de una lista.

Sintaxis: Primero se especifica el cuantificador que permite seleccionar los elementos de la lista. Entre estos están las opciones:

- **para todos (for all)**: Incluye a todos los elementos de la lista.

- **para al menos (for al least) N**: Incluye al menos N elementos de la lista.
- **para a lo sumo (for at most)**: Incluye a lo sumo N elementos de la lista.
- **para cualquier (for any)**: Se cumple si para algún elemento de la lista se cumple la proposición.
- **para exactamente (for exactly) N**: Se cumple si la proposición se cumple para exactamente N elementos de a lista.

Luego se especifica la variable con la que se referirá a cada elemento de la lista y de que lista proviene. Para esto se especifica una variable, luego los signos $<-$ seguidos del nombre de la variable lista, todo esto dentro de paréntesis. Por último se escribe el signo $\sim$ y un bloque de restricciones.

Ejemplos:

```
1 for all (i <- l0) ~ {
2   0 <= i <= 100;
3 };
4 for at least 5 (i <- l1) ~ {
5   0 <= i <= 100;
6 };
7 for exactly 10 (i <- l2) ~ {
8   0 <= i <= 100;
9 };
```

Figura 3.12: Ejemplos de variable restringida por cuantificador

Para cada caso de la figura **??** lo que dice cada restricción es:

- Una lista de números llamada `l0` con todos los valores aleatorios entre `0` y `100`.
- Una lista de números llamada `l1` con al menos `5` de los valores aleatorios entre `0` y `100`.
- Una lista de números llamada `l2` con exactamente `10` de los valores aleatorios entre `0` y `100`.

Capítulo 4

Diseño del solver y compilador

En este capítulo se explicará el funcionamiento del proceso de compilación del lenguaje así como varios conceptos importantes relacionados. Este proceso de compilación es también un proceso de resolución de problemas y es el responsable de conseguir las soluciones a las entradas proporcionadas.

4.1. Árbol Sintáctico Abstracto

La estructura obtenida del *Parser* que contiene toda la información posible es el Árbol Sintáctico Abstracto. En este se encuentran tanto las definiciones de variables como las restricciones. Contiene también la estructura principal, las auxiliares y las funciones, sean estas o no utilizadas.

No hubo necesidad de generar una tabla de símbolos ya que a diferencia de otros lenguajes los valores de las variables solo pueden pasar de no asignado a asignado y luego no hay posibles modificaciones. De hecho solo si la variable es constante su valor comienza siendo asignado, de resto no será sino hasta los últimos pasos del proceso de compilación cuando consiga un valor concreto.

El árbol luce como la figura **??** y contiene:

- Las estructuras de la representación.
- La descripción de las variables así como sus posibles restricciones individuales y globales.
- Las funciones particulares que el usuario describió.

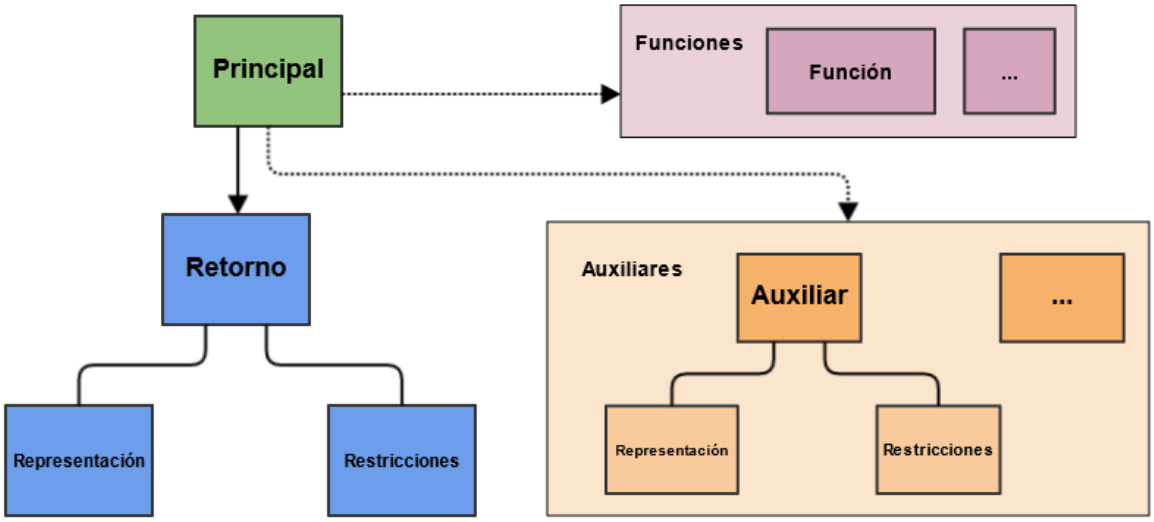

Figura 4.1: Gráfica del AST

4.2. Grafo y generación de estructura de retorno

Dentro del compilador no será suficiente contar con el **AST** para hacer los cálculos necesarios y obtener las soluciones. Para estos se utilizarán dos estructuras:

- Un grafo: Que delimitará los grupos de variables y se encargará de calcular los valores de soluciones finales.

- Una estructura de salida: Que será el equivalente a la estructura especificada. En esta se almacenarán los valores de las variables finales y luego servirá para producir la salida en formato genérico.

4.3. Aclaraciones iniciales sobre el solver

Antes de explicar el funcionamiento del *solver* es necesario primero definir ciertos conceptos clave para entender su funcionamiento.

4.3.1. Sistemas de variables

Hay que tener claro que una parte fundamental del *Solver* es como bien su nombre lo sugiere solucionar problemas, esto debido, a su naturaleza de **CSP**. Para esto hay que analizar y conseguir todos los posibles valores que se le pueda dar a cada variable, en especial a aquellas que dependan de otras variables. Debido a que no todas las restricciones son de tipo algebraico se prefiere llamarlo *sistema de variables* más que el tradicional concepto de *sistema de ecuaciones*.

Entonces, para definir claramente un *sistema de variables* hay que considerar los siguientes puntos:

- Un conjunto de variables forman un sistema si existen restricciones que las involucren entre ellas de forma transitiva.
- Es posible encontrar muchos sistemas dentro de un mismo problema.
- Ya que no existen relaciones entre varios sistemas cada uno es totalmente independiente de los otros y por lo tanto, el proceso de resolución que se explicará más adelante es aplicado a cada sistema independientemente.

4.3.2. Conjuntos

Para poder representar los posibles valores que se pueden asignar a una variable es necesario utilizar un tipo de dato o alguna notación que permita representar conjuntos de datos. En especial grandes conjuntos ordenados de enteros.

Para esto, se ha pensado en utilizar alguna representación no expandida como listas por comprensión o alguna otra diseñada para este fin. Por ejemplo algo como esto: Los números entre -300 y 500 más los números del 550 al 600 sin incluir los pares tendría la siguiente representación $([-300.,500] \cup [550.,600]) - pares()$.

Sin embargo, debido a la complicación de representar los datos de esa forma en el lenguaje principal del *Solver*, lo cual posiblemente llevaría a la implementación de otro sub-lenguaje, se decidió que los conjuntos de valores se representaran como los rangos iniciales de las variables y las listas de restricciones de dominio (que serán explicadas en los siguientes puntos). Cuando llegue el momento serán evaluadas y por ende expandidas, para ser almacenadas en archivos en memoria secundaria y evitar sobrecargar la memoria del programa.

4.3.3. Restricciones

Hay dos tipos de restricciones que se pueden declarar en los modelos de entrada. Cada uno de ellos tiene sus propias implicaciones a nivel de resolución:

4.3.3.1. Restricciones de dominio

Estas son las restricciones más básicas que pueden definirse y son esenciales para definir el dominio de valores que pueden dársele a cada variable, independientemente del resto de las variables. Estas se caracterizan por lo siguiente:

- Se declaran en el área de descripción de las estructuras.

- Permiten, como su nombre lo indica delimitar los valores que pueden obtener las variables sin importar el de las demás.
- Son aquellas referentes a una sola variable.

Adicionalmente hay dos formas distintas de restricciones de dominio:

1. **Restricciones genéricas de dominio**: Se refieren a las restricciones definidas sobre una variable básica en alguna estructura. Estas restricciones definen el comportamiento de dicha variable de forma general y por lo tanto, cada instancia de la estructura que la contiene someterá a la variable en cuestión de la misma forma. Este tipo de restricción es especialmente útil para cuando se instancian muchas objetos de una misma estructura y se quiere que el comportamiento de sus variables sea el mismo.
2. **Restricciones propias de dominio**: Son las restricciones que se asocian a una variable de tipo no básico o estructural, estas restricciones por ende hacen referencia a alguna de las variables básicas internas de la estructura. La diferencia entre este tipo de restricción con la anteriormente mencionada es que para estas las restricciones, sólo afectan la instancia particular asociada a la variable que se está restringiendo y por ende, las demás variables del mismo tipo no se verán afectadas. Este caso es de especial utilidad cuando existen varias variables de un mismo tipo de estructura, pero se quiere que cada una tenga un comportamiento distinto de las demás (tanto como se quiera y respetando las restricciones genéricas a su tipo y las restricciones de rango que se explicarán a continuación).

4.3.3.2. Restricciones de rango

Un problema sería muy sencillo de resolver si sus variables no se relacionaran entre sí. De hecho si no se relacionasen bastaría con calcular los dominios de cada variable y decir

que esos son definitivamente los valores finales que puede tomar cada una. Sin embargo este tipo de problemas no es realmente un reto y por otro lado no permitiría definir estructuras complicadas e interesantes. Por ejemplo si se quisiera generar figuras humanas se quisiera que sus partes contaran con algo de simetría o que se rijan bajo ciertas proporciones, desde las más simples como las proporciones de alto y ancho hasta cosas más complicadas como definir que mientras más rojizo es el color del pelo más pecas podría tener en la cara. Incluso para cosas tan simples como calcular una distancia. es necesario relacionar varias variables y para esto hay que definir un tipo de restricción particular.

Ya que el resultado de estas restricciones condiciona directamente las posibles respuestas finales, se ha decidido nombrarlas como **restricciones de rango**. Y estas restricciones cumplen con las siguientes propiedades:

- Son aquellas que relacionan algunas variables con otras.
- Se especifican en el área de restricciones.
- Permiten delimitar el rango y por lo tanto filtran el conjunto de vectores de resultados posibles.
- Son realmente las únicas que presentan dificultad para calcularse, ya que dependerán de algún método de resolución. Sea este *backtracking* o métodos numéricos, cosa que será tratada más adelante.
- Casi siempre cuando se hable de restricciones en este documento se referirá a este tipo de restricciones.

Al igual que para las restricciones de dominio, las restricciones de rango pueden afectar de forma genérica a un mismo tipo de estructura (restricciones genéricas de rango) o pueden afectar solo a una instancia particular (restricciones propias de rango). El razonamiento es

prácticamente el mismo que ya se mencionó para las restricciones de dominio así que se omitirá repetir esta explicación.

4.3.3.3. Relación dominio-rango

Para poder conseguir valores solución es necesario cumplir con dos condiciones: la primera implica que la variable en cuestión pueda tomar el valor sugerido; la segunda que el valor que tome la variable en conjunto con el de las demás cumplan con todas las restricciones de rango. Ambas condiciones son absolutamente necesarias o los valores no tendrían sentido dentro del problema que se quiere resolver. Ahora bien es importante analizar de que forma se afectan las restricciones de dominio y de rango entre ellas:

En primer lugar, las restricciones de dominio son propias para cada variable y por ende son totalmente independientes. Es decir que una restricción de dominio en una variable no afectará los valores posibles de otras variables y viceversa (no se considera el efecto de las restricciones de rango). Entonces las restricciones de dominio pueden calcularse todas al mismo tiempo o unas antes que otras.

Otra observación es que entre las restricciones de dominio y de rango tampoco hay relación, es decir que se podría decidir evaluar primero las restricciones de dominio y luego utilizar estos valores para y filtrar aquellos que cumplan las de rango; por otro lado, podría también evaluarse los de rango y conseguir los vectores solución para luego filtrar las soluciones viables de las que no por los valores válidos de dominio.

Por último, las restricciones de rango dentro de un mismo sistema de variables deben y necesitan ser evaluadas al mismo tiempo siempre. Ahora bien, si estas restricciones no se encuentran en los mismos sistema de variables, pueden evaluarse en cualquier orden al igual que pasaría con las restricciones de dominio.

- Las restricciones de dominio y las de rango son totalmente independientes entre sí y

solo se consideran simultáneamente a la hora de buscar el o los resultados finales.

- Las restricciones de dominio pueden ser aplicadas en orden arbitrario (exceptuando los casos en los que involucran funciones -no consideradas por ahora-). Al ser aplicadas van acortando el conjunto de valores que puede tomar la variable.

4.4. Árbol de Salida

Para devolver la estructura deseada ya estando inicializada hace falta primero construir la estructura y expandir todas sus sub-estructuras. Para esto se utiliza el modelo que ya se construyó en el **AST**.

Este árbol contiene variables que tomarán valores una vez se hayan resuelto todos los sistemas de variables del problema y una vez completo se aplicará una función para convertir a este en un *string* con formato genérico ya sea un *XML* o un *JSON*. La estructura de salida será representada por un árbol y cuya estructura es similar al de la figura **??**.

- Cada elemento del árbol es o un nodo estructura (que actúa como contenedor) o un tipo básico que representa una de las variables de la estructura.

- Los contenedores son las expansiones de las estructuras auxiliares que son contenidas por la estructura de retorno o incluso las que son contenidas por otras estructuras auxiliares.

- Mientras se resuelven los sistemas las variables cuyos valores no sean asignados constantes por defecto no habrán sido inicializadas.

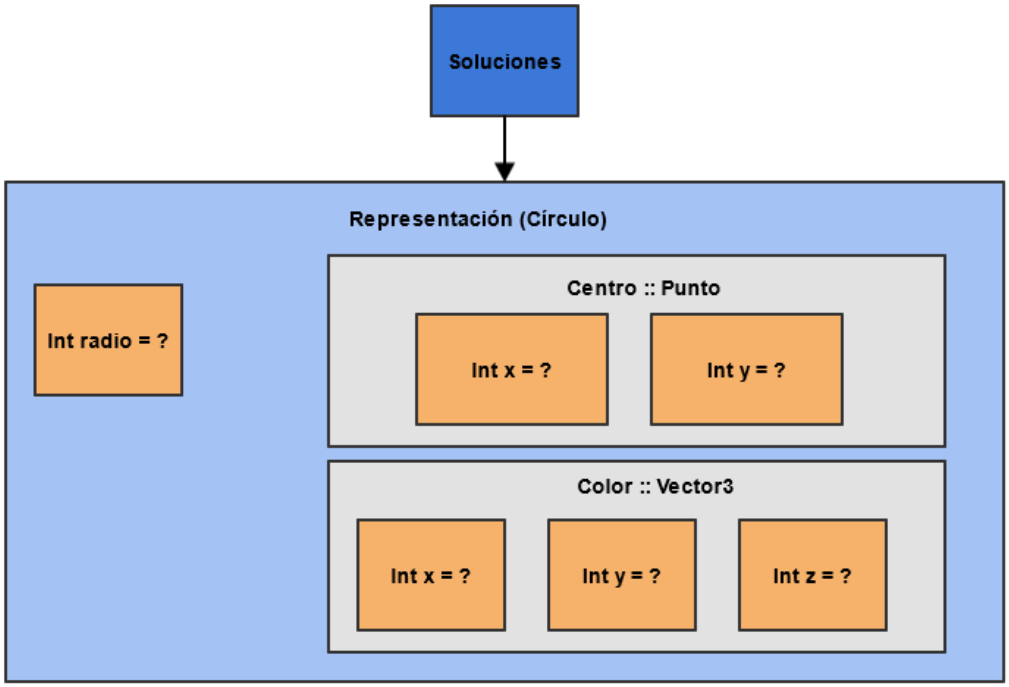

Figura 4.2: Gráfico de Árbol de salida

4.5. Grafo de resolución

Los grafos son una excelente herramienta para representar datos relacionados entre sí. Su forma permite realizar operaciones rápidamente y evitar redundancias. En este caso, es necesario el uso de un grafo por la facilidad que ofrece para representar todas las partes de los sistemas de variables. En este caso, el grafo del compilador se utiliza para almacenar la información necesaria con la que se realizará el cálculo de rangos solución.

La estructuración del grafo es la siguiente: los nodos serán utilizados para almacenar los datos referentes a las variables, mientras que los arcos tendrán las restricciones de rango. De esa forma cada variable tiene como alcance solo a las demás variables que forman parte de su sistema de variables.

El grafo ofrece una ventaja cuando se manejan estructuras dinámicas como las listas, ya

que puede modificarse sin mucha dificultad, no obstante, se ha descartado la generación de estructuras dinámicas por ahora debido a que el manejo de estas supera el alcance definido en un comienzo para este trabajo (para más información sobre esto se puede consultar la sección **??**).

Como por ahora no se está trabajando con estructuras "dinámicas" el grafo inicial será el grafo necesario para representar las variables de la estructura de retorno y sus estructuras auxiliares hijos.

4.5.1. Variables (Nodos)

Las variables serán representadas de la siguiente forma:

- Una variable que denota el tipo de la variable representada.
- Un rango de valores según el tipo de variable.
- Una referencia a la misma variable en el árbol de salida (sección **??**).
- Una lista con referencias a las restricciones que involucran a la variable.

4.5.2. Restricciones (Arcos)

Las restricciones tienen la siguiente representación:

- Son una estructura que contiene un entero con la cantidad de variables que involucra.
- Una lista con referencias a dichas variables y con su ubicación dentro de la restricción.

4.6. Proceso de resolución

Luego de haberse generado el **AST** en la fase de reconocimiento sintáctico, se procede a generar la instancia del objeto en cuestión. Para esto se deben instanciar las partes del

objeto al mismo tiempo en que se verifica el cumplimiento de las restricciones. A esta parte se le denominará etapa de resolución, la cual a su vez estará también compuesta por partes:

4.6.1. Parte primera: Generación de la estructura de salida

En esta primera parte se procede a construir la estructura que representa el objeto a retornar sin instanciar. Para esto se realizan los siguientes sub-pasos:

1. Se crea el árbol de salida copiando los elementos de la estructura de salida del **AST**.
2. Se expanden las estructuras auxiliares que sean requeridas recursivamente.
3. Se inicializan las variables y se instancian solo si están declaradas como constantes desde la entrada.
4. Cada variable no constante que es incluida en el árbol también es creada como un nodo en el grafo.
5. Mientras se crean los nodos se operan los rangos con las restricciones de dominio correspondientes.
6. Luego se incluyen las restricciones de rango y se asignan las referencias desde los nodos hacia y desde estas.

Un resumen de lo que se genera en esta etapa es lo que se muestra en la figura **??**.

4.6.2. Parte segunda: Cálculo del dominio

En esta parte se obtendrán los posibles valores iniciales que puede adoptar cada variable. Para esto es fundamental la noción de conjunto que se había establecido. El cálculo del dominio se efectúa de la siguiente forma:

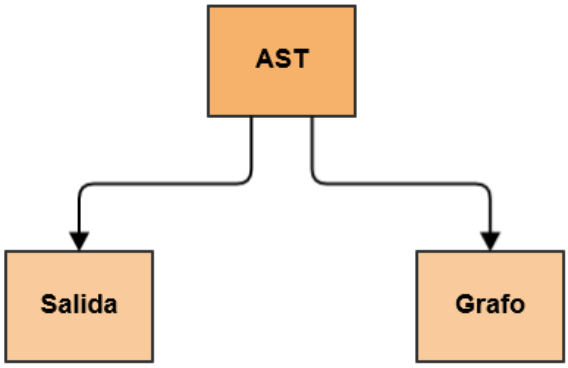

Figura 4.3: Compilación parte 1

1. Para cada variable no instanciada, de tipo básico encontrada en el **AST** y que haya sido creada también en la estructura de salida, se crea un nodo en el grafo. Las variables de tipo estructura no serán consideradas para esto, pues sólo son referencias para las de tipo básico.

2. Si la variable es de un tipo básico, entonces a su nodo en el grafo se le añaden las restricciones propias.

3. Luego de desplegar la estructura de salida esta misma se recorre en orden inverso. Entonces las restricciones propias de las variables estructurales se añaden a la variable referenciada.

4. En este mismo recorrido inverso se añaden las restricciones no propias como arcos del grafo.

5. En este punto cada variable cuenta con una lista de todas las restricciones propias a si mismas. Luego, para cada uno de los nodos en el grafo se crea un hilo para que se encargue de el cálculo de su dominio.

6. El hilo, cuyo funcionamiento será explicado en breve deberá recibir un archivo con las restricciones.

En resumen de lo que se estudia en esta etapa es lo que se muestra en la figura **??**.

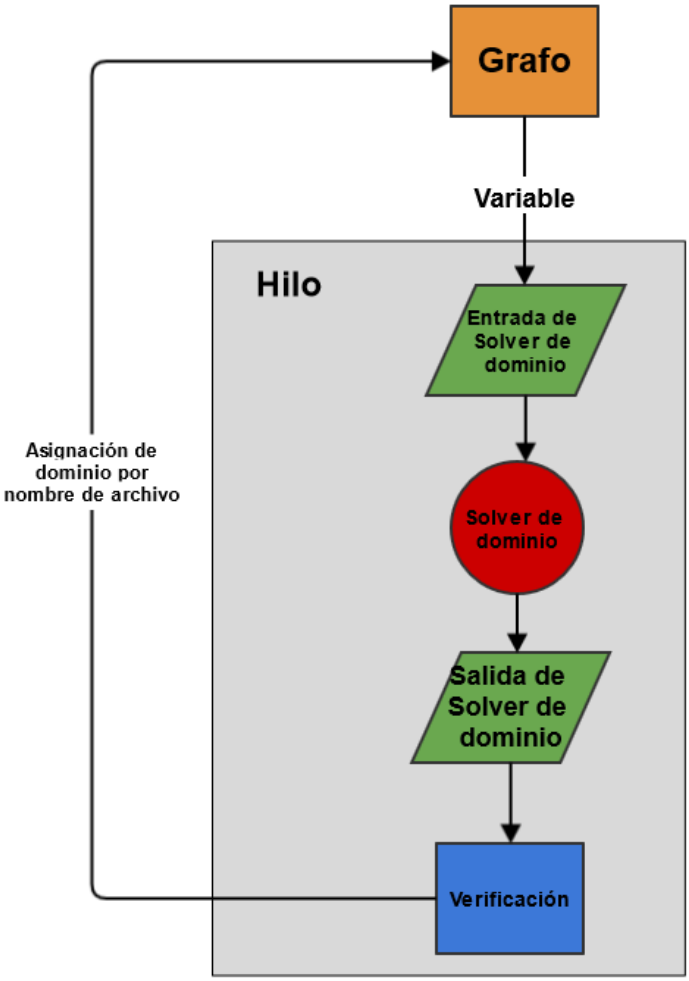

Figura 4.4: Compilación parte 2

4.6.2.1. Hilo para el cálculo de dominio

Para cada variable se creará un hilo con el que se delimitará su dominio. Estos hilos funcionarán de la siguiente forma:

- El hilo escribirá en un archivo identificado con un nombre clave para la variable, todas las restricciones afines a la misma, estructuradas de forma que el programa solucionador de dominio escrito en *Prolog* pueda interpretarlo como un predicado.
- Luego de escrito el archivo el hilo, espera a que el programa en *Prolog* al cual se le llamará a partir de ahora "Solver de dominio" o "Solucionador de dominios" (sección **??**) termine su ejecución.
- El *solver* de dominio habrá escrito un archivo con el resultado obtenido. Ahora el hilo deberá abrirlo y revisar si hubo un error. Si lo hubo, el hilo se encargará de la interrupción del programa, de lo contrario añadirá el nombre del archivo a la información contenida en los nodos.
- El hilo termina su ejecución.

A partir de este punto el programa esperará a que todos los hilos terminen su ejecución. Con lo que todos los dominios estarán ya calculados.

4.6.2.2. *Solver* de dominio o solucionador de dominio

Este es un programa escrito en el lenguaje de programación *Prolog* cuya función es utilizar *backtracking* sobre los valores comprendidos en el rango de la variable, para conseguir todos los valores posibles que cumplan con todas las restricciones. La especificación del funcionamiento de este programa y sobre el uso de *Prolog* serán tratadas en otro punto.

4.6.3. Parte tercera: Cálculo de rangos

La dinámica para el cálculo de rangos es similar a la del cálculo de dominios, solo que adicionalmente se deben definir y separar los sistemas de variables antes de resolverse. Esto se hace de la siguiente forma:

1. El primer paso es aplicar el algoritmo de Roy Warshall [**?**] al grafo para hallar la clausura transitiva de los nodos y así poder separarlos por grupos. Estos grupos definen cuales son los sistemas de variables.

2. Si existe algún grupo con una sola variable entonces se considerará que este sistema está resuelto y se modificará el nombre del archivo de dominio asociado a esta para ser utilizado como rango del sistema.

3. Para cada sistema se crea un hilo que actuará de forma idéntica a los que trabajan los dominios, con la excepción de que estos transcribirán las restricciones de forma diferente y llamarán a otro programa en *Prolog* distinto para que resuelva las rangos. Este programa será llamado "Solver de rangos" o "Solucionador de rangos" (sección **??**).

4. Al recibir respuesta por parte del *Solver* el hilo revisa si el archivo contiene algún error o si el rango es vació. Si todo es correcto escribe la dirección del archivo en la representación del sistema.

Luego de este punto el grafo es innecesario y puede ser olvidado o destruido. De la misma forma , no hace falta almacenar los archivos de dominio. En la sección referente a la optimización se hablará más de esto. Un resumen de lo que se genera en esta etapa es lo que se muestra en la figura **??**.

4.6.3.1. Solver de rangos o Solucionador de rangos

Este programa actúa de forma similar al *Solver* de dominios, su diferencia con este reside en que este primero toma como entrada los archivos correspondientes a la variables del sistema y adicionalmente recibe un archivo escrito por el hilo quien lo llamó, que contiene la

especificación de las restricciones que tiene el sistema. Por último este programa retorna la dirección del archivo que escribe con la información de los posibles rangos del sistema.

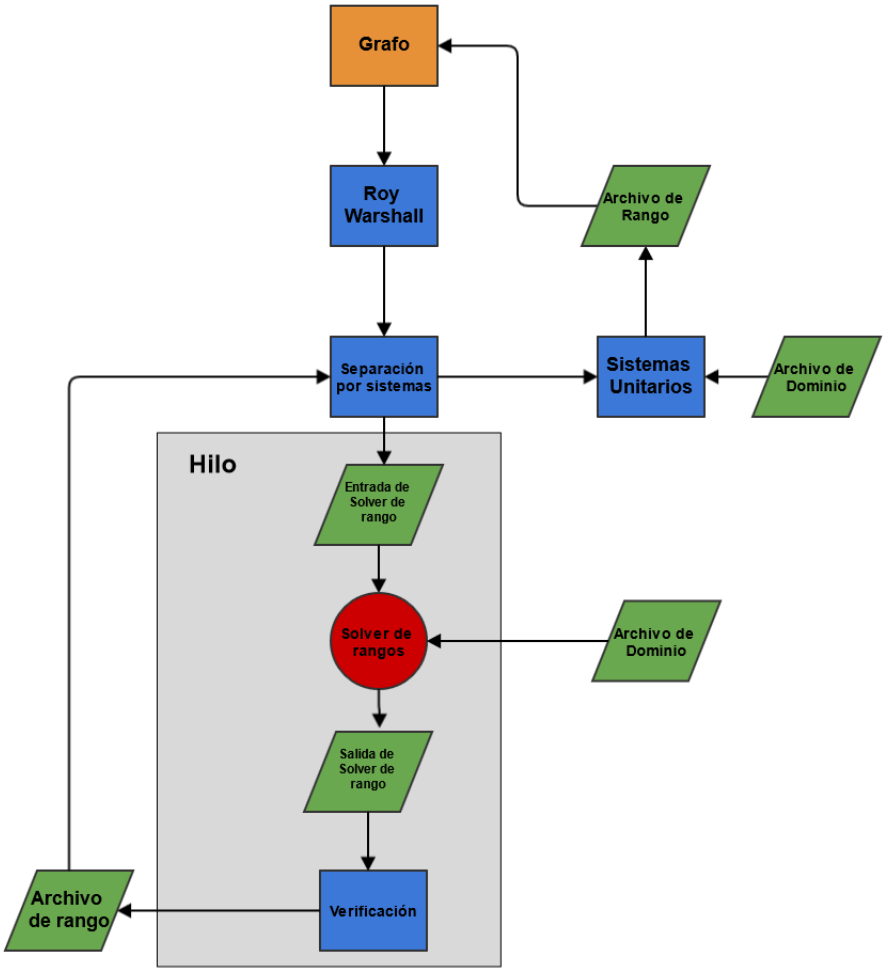

Figura 4.5: Compilación parte 3

4.6.4. Parte cuarta: multiplicación de rangos

Una vez conseguidos los rangos para cada sistema solo falta unir estos resultados con los de los demás sistemas para conseguir soluciones finales. Esta parte del compilador es

relativamente simple puesto a que sólo hay que hacer la multiplicación de los vectores de rango. Para esta operación se debe decidir de forma aleatoria una solución de cada sistema y unirlos. Una forma gráfica de verlo es tal y como luce en la figura **??**.

A partir de este punto no hace falta almacenar los archivos de rangos.

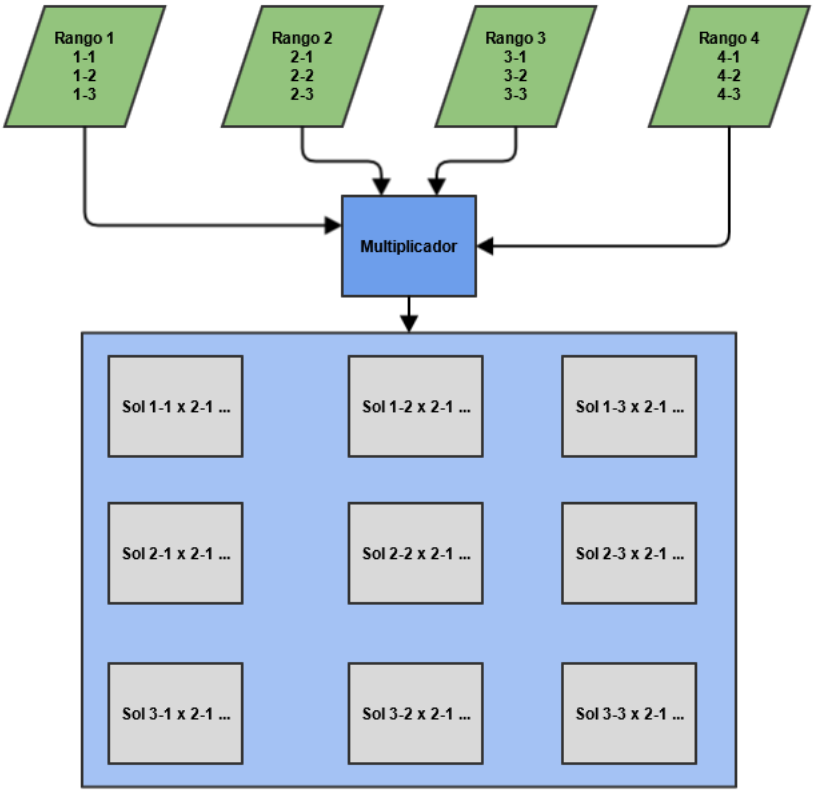

Figura 4.6: Compilación parte 4

4.6.5. Parte quinta: Instanciación de valores

Una vez obtenidas las soluciones finales, resta únicamente instanciar la estructura de salida con los valores de las variables de cada solución y reducir la salida de las soluciones solicitadas por el usuario. Esta salida se realiza en un formato genérico en *JSON* para que pueda ser interpretado y utilizado por otro programa tal y como se muestra en la figura **??**.

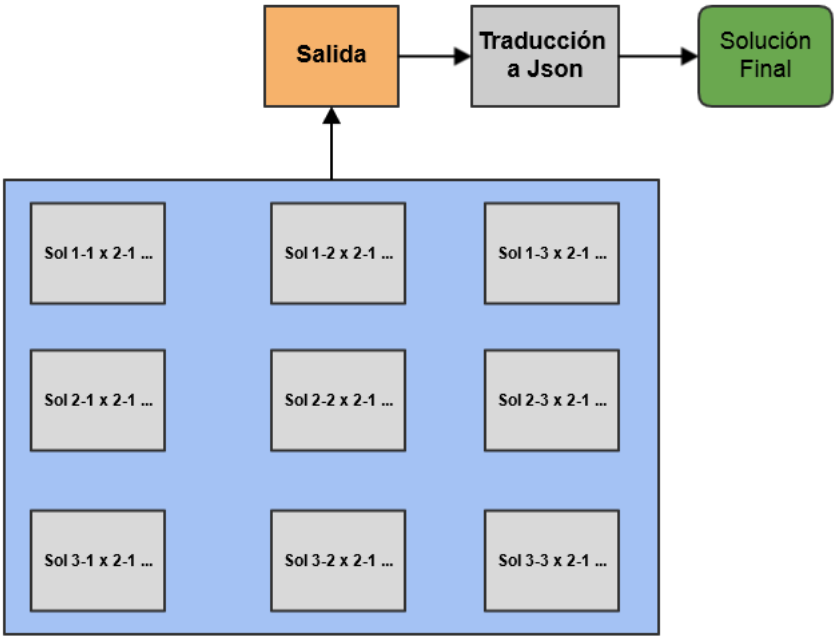

Figura 4.7: Compilación parte 5

Capítulo 5

Implementación y resultados

En el presente capítulo se explicará la implementación realizada según la especificación presentada los capítulos **??** y **??**. Así como también se presentarán los resultados obtenidos para los ejemplos dispuestos en el apendice **??**.

5.1. Implementación

La implementación del *Parser* fue realizada en *C++* usando las herramientas *Flex* [**?**] y *Bison* [**?**] para el reconocimiento del programa, estos corresponden al *Lexer* y *Parser* discutidos en la sección **??**.

La herramienta de **CSP** (discutida en la sección **??**) usada fue *Prolog* [**?**]. *Prolog* es un lenguaje basado en el paradigma de programación lógica. Basa su funcionamiento en el razonamiento de expresiones lógicas y la unificación de términos para satisfacer tales expresiones, usando el mecanismo de inferencia conocido como resolución y una estrategia de *backtracking* para poder explorar los distintos espacios de búsqueda. Esto permitió que la escritura del mecanismo de resolución por *backtracking* sea muy simple y no tener que ahondar en el *Solver* para poder dedicar más esfuerzo a los objetivos principales del proyecto. La comunicación

con *Prolog* se logra a través de archivos y se realizan llamadas al sistemas operativo para ejecutarlo (que será descrito más adelante en esta sección) desde la herramienta. Los archivos intermedios generados se encuentran por defecto en un directorio temporal y son de 4 tipos diferentes:

- El archivo de dominio de valores discutido en la sección **??**. Una explicación con más detalles se encuentra en el apéndice **??**.
- El archivo correspondiente a una variable independiente en el sistema es reconocida en la herramienta y se deja sólo para optimizar cálculos. Una explicación con más detalles se encuentra en el apéndice **??**.
- Cuando una variable se encuentra relacionada con otras o tiene al menos una restricción se devuelve la solución en un archivo de que se llama rango. Una explicación con más detalles se encuentra en el apéndice **??**.
- Por último se encuentran los archivos de *Prolog* que contienen las instrucciones para la resolución del sistema y generación de archivo de salida. Una explicación con más detalles se encuentra en el apéndice **??**.

Finalmente las instancias generadas son del formato *JSON* [**?**] y se devuelven por defecto en un directorio que contendrá todas las salidas obtenidas por la herramienta.

Para cada uno de los ejemplos se crearon programas a manera de librerías, en los que se reconocieron las soluciones obtenidas por la herramienta para convertirlas en objetos dentro de un lenguaje o contexto específico. Estos programas fueron implementados en los lenguajes de programación *Python* [**?**] y *C#* [**?**] debido a la facilidad que ofrecen a la hora de interpretar los archivos en formato *JSON* y por las necesidades de librerías para mostrar o utilizar los objetos generados.

5.2. Pruebas realizadas a nivel del Solver

Para las pruebas se estructuraron y probaron varios problemas como entrada para la herramienta. Estos problemas fueron los considerados más importantes en cuanto a utilidad e interés para las pruebas.

En el caso de los ejemplos contenidos en el apéndice **??** y otros que tenían un nivel de complejidad parecido en cuanto cantidad de variables involucradas y restricciones, lograron generar resultados satisfactorios y muy rápido, tal y como se muestra en la tabla **??**.

Instancias	1	100	1000	10000
	real 0m0.064s	real 0m0.166s	real 0m0.262s	real 0m1.194s
Base de Datos	user 0m0.056s	user 0m0.060s	user 0m0.152s	user 0m0.924s
apéndice **??**	sys 0m0.004s	sys 0m0.028s	sys 0m0.028s	sys 0m0.192s
	real 0m0.099s	real 0m0.114s	real 0m0.490s	real 0m6.043s
Computación Gráfica	user 0m0.092s	user 0m0.100s	user 0m0.368s	user 0m5.184s
apéndice **??**	sys 0m0.000s	sys 0m0.012s	sys 0m0.060s	sys 0m0.744s
	real 0m1.059s	real 0m1.190s	real 0m1.224s	real 0m1.785s
HTML	user 0m1.004s	user 0m1.044s	user 0m1.068s	user 0m1.460s
apéndice **??**	sys 0m0.044s	sys 0m0.048s	sys 0m0.072s	sys 0m0.208s

Cuadro 5.1: Comparación de resultados

Sin embargo, una de las pruebas demostró que un sistema sencillo con pocas variables, pero con muchos valores posibles en el dominio, puede hacer que la herramienta interrumpa su ejecución. Por ejemplo si se tienen 3 variables con 1000 valores posibles en el dominio por cada una, se generaran $1000 \times 1000 \times 1000$ vectores solución al comenzar la ejecución, los cuales no caben en la memoria de *Prolog*. Aunque, si se contara con un sistema de evaluación perezosa como el de *Haskell* no pasaría esto ya que se generarían vectores solución a medida que se necesitan.

Otras pruebas que ocasionan problemas son aquellas que tienen muchas restricciones. Debido a que *Prolog* tardaba mucho en su ejecución, estas producen un cierre inesperado del programa. Esto ocurre porque busca evaluar las restricciones en un orden que probablemente

es poco conveniente para el *solver*.

Es responsabilidad del programador el escribir una descripción de las estructuras que permita un mejor desempeño de la herramienta, ya que no especificar el dominio de una variable, o tener restricciones en un orden poco conveniente de revisión, repercute negativamente sobre el desempeño de la herramienta, al igual que en cualquier otro lenguaje, el colocar en determinado orden instrucciones o hacer mal uso de ciertos recursos, suelen tener resultados no deseados.

Los resultados generados abarcan la totalidad de las respuestas posibles. Es decir, para cada ejecución de la herramienta, todas las instancias válidas como solución fueron generadas. Por ende se puede afirmar que la herramienta cumple con la condición de completitud en cuanto a los resultados generados.

Todos los resultados cumplen con las restricciones sobre las cuales están fundamentados y delimitados. Por lo que no se producen resultados "erróneos" o no válidos.

Por otro lado, la herramienta no es capaz de detectar repeticiones entre las respuestas proporcionadas. Así que algunas respuestas tienen más probabilidad de aparecer que otras en el proceso de selección aleatoria.

Capítulo 6

Optimización

Durante el proceso de compilación hay varios momentos en los que se pueden llevar a cabo algunos procesos que permitan optimizar algunas partes y conseguir mejoras en el tiempo de ejecución y en el espacio utilizado tanto en memoria como en disco. Aunque muchas de estas decisiones son automatizadas, hay otras que dependen de las necesidades del usuario y de su conocimiento sobre el uso de las soluciones que desea generar.

6.1. Optimización Manual o dependiente

En esta parte se analizarán las decisiones que puede tomar el usuario (programador) que permiten conseguir mayor velocidad en los cálculos o menor uso de memoria.

6.1.1. A nivel del Parser o reconocedor sintáctico

A nivel de el reconocimiento sintáctico hay muy pocas decisiones que puede tomar un programador para optimizar el funcionamiento del mismo o el compilador. Esto, obviando claro la fuerte dependencia que existe entre los recursos utilizados, con respecto al nivel de complejidad del modelo de entrada.

La cantidad de variables y la cantidad de valores que puede tener inicialmente cada una, afecta proporcionalmente la cantidad de memoria utilizada por el programa. Evidentemente las restricciones de dominio de la variable disminuyen la cantidad de valores que puede tomar.

Por otro lado la cantidad de restricciones de rango que involucren mismos sistemas de variables, incrementan considerablemente la complejidad de los cálculos necesarios, para determinar los rangos de solución de los sistemas.

6.1.2. Partes de compilación

Las partes en la compilación están diseñadas para permitir segmentar el proceso y hacer cada parte independiente del resto. Esto permite interrumpir la compilación y concluirla más adelante desde la última parte completada.

Mediante un flag en el comando de compilación es posible especificar hasta que parte de la compilación se desea llegar y luego con otro flag se puede continuar desde este punto sin tener que repetir todas las partes anteriores. A está opción que continua el proceso basado en los archivos generados por el compilador es llamado instanciador y tiene la particularidad de que no conserva los archivos que genera luego de obtener las soluciones finales.

Dependiendo del tipo de instancia a conseguir, el problema a resolver o contexto donde se solicita el uso de la herramienta, es más conveniente compilar hasta cierta parte:

6.1.2.1. Luego del reconocimiento sintáctico

Compilar hasta esta parte es recomendable solo en un caso:

- Para probar la correctitud de la entrada: se puede saber si tiene errores sin la necesidad de obtener resultados.

6.1.2.2. Luego de la primera parte

Hasta este punto solo se han generado la estructura de salida y el grafo:

- Para problemas pequeños en donde no se quiera almacenar datos: si un problema tiene variables con una gran cantidad de valores posibles en su dominio y tiene restricciones de considerable complejidad, entonces es probable que no se desee almacenar ningún resultado intermedio de los cálculos.

6.1.2.3. Luego de la segunda parte

Hasta este punto se conserva: la estructura de salida, el grafo y los archivos de descripción de dominio para cada variable. Esta optimización es preferible para el siguiente caso:

- Para problemas con dominios particularmente difíciles de resolver: de este modo el calculo queda almacenado para no repetirlo en las próximas corridas.

6.1.2.4. Luego de la tercera parte

Hasta este punto se conservan: la estructura de salida y los archivos de descripción de rango para cada sistema de variables. Esta optimización es preferible para los siguientes casos:

- Para problemas con rangos difíciles de resolver: de la misma forma que con el caso anterior el cálculo de los rangos quedan almacenados para no repetirlos luego.
- Para problemas con rangos pequeños: aún si los dominios son muy permisivos, si las restricciones de rango limitan mucho las soluciones de los sistemas de variables, conviene almacenar estas soluciones.

Esta parte es la considerada predilecta para problemas con muchas soluciones finales, ya que almacena las soluciones de los sistemas antes de ser multiplicados, acción que indiscutiblemente requerirá mucho más espacio de almacenamiento.

6.1.2.5. Luego de la cuarta parte

Hasta este punto se conservan: la estructura de salida y los archivos con las soluciones finales sin formato. Esta optimización es preferible para el siguiente caso:

- Cuando se necesita obtener resultados rápidamente: luego de esta parte los resultados ya están calculados y solo falta darles formato para que sean interpretados por el destino. Dejar la compilación hasta este punto es favorable si no importa ocupar mucho espacio almacenando estos resultados, a cambio de obtenerlos rápidamente.

6.1.2.6. Luego de la quinta parte

Hasta este punto se cuenta ya con todos los archivos solución del problema.

- Luego de esta parte no hace falta ejecutar el instanciador ya que las soluciones ya fueron generadas, lo que resta es elegir mediante algún método la solución deseada e interpretarla.

- Este caso es preferible si se desea contar directamente con las soluciones y no ejecutar la herramienta. Esto es considerablemente útil si se requieren los problemas durante la ejecución de otro programa y no se quiere que el cálculo de estas soluciones repercuta en el uso de recursos para el cálculo.

- Esta forma de compilación calcula todos los resultados y almacena cada uno en un archivo con un formato genérico. Esto representa en la mayoría de los casos un gasto grande de memoria. Por esto, este caso no debe tomarse a la ligera y solo ser usado si realmente es preferible gastar espacio de almacenamiento en vez de hacer los cálculos para hallar las soluciones. O si se sabe que se van a necesitar muchas soluciones en diferentes momentos y no vale la pena calcular las soluciones una y otra vez.

6.2. Optimización automatizada

Se diseñaron varios mecanismos para la optimización a tiempo de corrida que no dependieran de el uso explícito de opciones por parte del usuario. Estos mecanismos no fueron implementados por motivos de complejidad y de tiempo. Sin embargo, se considera de fundamental importancia explicarlos para que sean implementados en un futuro y mejoren en desempeño de la herramienta lo más que se pueda. Sin más que decir se presentan a continuación los mecanismos de optimización automáticos que se diseñaron:

6.2.1. Diferenciación de tipos de problemas

Se han catalogado los problemas de entrada en dos tipos, esta distinción se basa en la forma en que están estructuradas las entradas y la facilidad que ofrecen para ser resueltos por un método u otro. Antes de continuar sería conveniente explicar estos métodos de resolución:

6.2.1.1. Método de resolución Algebraico

Es posible obtener los valores solución a un sistema de variables planteando, tratando este como un sistema de ecuaciones tradicional y resolviéndolo por métodos numéricos. Dado que a nivel de *Solver* todas las variables son manejadas como flotantes y enteros es viable conseguir los vectores solución de estos sistemas para cualquier sistema consistente.

Además este método tiene como ventaja la posibilidad de separar completamente el cálculo del dominio de las variables respecto al de rango de los sub-sistemas. Por ende es posible calcular los dominios o los rangos en cualquier orden, incluso simultáneamente, de forma tal que se puedan intercambiar las partes dos (sección **??**) y tres (sección **??**) dependiendo de la forma del problema.

6.2.1.2. Método de resolución por *Backtracking*

A tiempo de publicación de este documento el método por *backtracking* es el único completamente implementado y por lo tanto el mecanismo por defecto utilizado para conseguir soluciones a los sistemas de variables. Este se basa en la utilización del método algorítmico de *backtracking* para buscar todas las posibles soluciones a la asignación de valores de para variables.

Es sabido que este método tiene numerosas desventajas en cuanto a eficiencia se refiere, en especial por que recorre todo el espacio de búsqueda para conseguir soluciones viables y no es capaz de distinguir entre caminos viables de los que no. Aún así y dado a la genericidad de la herramienta y de los posibles problemas proporcionados, se considera que es una opción por defecto bastante aceptable. Adicionalmente, este método puede ser mejorado y convertirse en un *backtracking* no cronológico [**?**] (si el problema en cuestión lo permite).

6.2.2. Método de búsqueda especializado o híbrido

Sería recomendable establecer un sistema que discrimine los tipos de problema y basado en esto, utilice ciertas heurísticas para mejorar el desempeño del cálculo, incluso es posible utilizar métodos de aprendizaje de máquina para reconocer patrones y asignarlos a las mejores formas de resolución para los mismos [**?**].

Existen varias herramientas de software que se especializan en la resolución de problemas matemáticos pero que también permiten incluir cálculos para problemas lógicos y/o alfabéticos. Varios incluso, permiten especificar los sistemas basados en satisfacción de restricciones. El problema, es que estas herramientas no calculan todas las soluciones posibles y solo consiguen una de ellas o que solo calculan soluciones basadas en las relaciones entre variables y no considerando los dominios.

Estos pasos faltantes pueden ser solucionados por esta herramienta, haciendo que se cal-

culen todas las soluciones posibles (o el número solicitado) y haciendo un filtrado de los vectores solución basándose en las restricciones de dominio. Este planteamiento se comenzó a trabajar durante la elaboración de la herramienta. Sin embargo, se decidió no continuarlo porque implicaba modificar el software de resolución utilizado, llegando entonces a un nivel de complejidad muy grande y al mismo tiempo enfocando demasiado tiempo en un tema ajeno al objetivo principal del trabajo de investigación.

6.2.3. Decisión sobre el uso del método de resolución

Podría considerarse que un método es siempre mejor que otro, pero eso es totalmente relativo al tipo de problema que se resuelve. Entonces, lo más prudente sería poder discernir entre uno y otro tomando en cuenta las características específicas del problema a resolver y posiblemente las experiencias previas que se tengan para estas características específicas.

Considerando lo tratado en la sección **??** es claro que cada sistema de variables es independiente y por lo tanto su resolución es indiferente del resto de los sistemas, por esto, cada sistema de variables puede resolverse con un método de resolución indiferentemente del resto. Haciendo entonces la decisión de elegir un método u otro un problema particular de cada sistema y no del problema de entrada en general.

Entonces sería recomendable contar con múltiples métodos de resolución en vez de uno solo y poder utilizar uno u otro dependiendo de la entrada y más específicamente las propiedades de cada sistema de variables en particular.

Dado que actualmente solo está implementado el sistema de resolución por *backtracking* utilizando *Prolog*, no es necesario decidir entre métodos de resolución. Ahora bien lo ideal sería contar con una gran cantidad de métodos para que pueda optimizarse la resolución de cualquier sistema.

Una vez más es notoria la importancia de establecer un sistema que se encargue de

asignar automáticamente un método de resolución a cada sistema de variables. Idealmente un sistema que implemente aprendizaje de máquinas para que se puedan abarcar gran cantidad de posibilidades y pueda adaptarse a las necesidades del usuario.

6.2.4. Recomendaciones para el diseño del sistema de decisión

Hay una serie de consideraciones que deben tomarse en cuenta, para diseñar e implementar el sistema que distingue que método utilizar a la hora de resolver cada sistema de variables.

6.2.4.1. Comparación de métodos de resolución

Cada uno de los métodos de resolución tiene sus ventajas y desventajas en general o frente a algún tipo de sistema en particular:

El método por *backtracking* evalúa todas las posibles asignaciones de valores del vector solución y mantiene solo las que satisfagan las restricciones de rango. Tiene como ventaja que no depende de la complejidad de la interrelación de las restricciones de rango. Esto, por otro lado es una desventaja si el dominio es muy grande y las restricciones de rango son relativamente simples.

Mientras tanto, el método numérico resulta ser más efectivo para problemas en los que los tamaños de los dominios sean grandes y el sistema de ecuaciones resultante de las restricciones de rango sea relativamente complejo. Esto es debido a que en este caso la obtención de valores posibles de rangos es primeramente independiente de los dominios, es decir, se obtienen directamente vectores solución y luego se filtran los verdaderamente válidos según las posibilidades de asignación por los dominios. Entonces no hace falta revisar cada valor de los dominios cruzándose con los demás para calcular los rangos. Esto incide en disminuir de forma exponencial la cantidad de operaciones que se deben realizar (esto comparando con el método de *backtracking* con un sistema que tenga gran cantidad de posibles dominios y

muchas variables).

La eficiencia de un sistema híbrido depende del método utilizado para hallar las soluciones, por ende no es posible generalizar los beneficios o desventajas de este tipo de método de resolución.

Dado esto último por los momentos solo es posible hacer una comparación entre el método numérico contra el de *backtracking*. Para esto, es necesario considerar ciertos factores. Se describirán a continuación los que fueron considerados como más importantes:

Amplitud: Evalúa la cantidad de dominio total multiplicando todos los espacios de dominio de las variables. Esta medida permite saber cuan poco preferible es utilizar *backtracking* para resolver el problema, ya que mientras mayor sea el dominio total de las variables mucho mayor será la cantidad de operaciones necesarias para llevar a cabo el *backtracking*.

Disipación: Evalúa la cantidad promedio de variables involucradas en las restricciones de rango. Este indicador permite aportar información referente a la complejidad del sistema de ecuaciones, y por lo tanto, lo preferible que resultaría resolver el sistema mediante métodos numéricos.

Proporción de limitación: Indica la proporción entre la cantidad de variables en un sistema, en relación a la cantidad de restricciones de rango que este contiene. Permite al igual que el factor de disipación tener información sobre la complejidad del sistema de ecuaciones.

En general si el factor de amplitud es muy grande en relación a la facilidad de resolver el sistema por método numérico, entonces no vale la pena utilizar el método por *backtracking*. Ahora bien, si la cantidad de dominios entre las variables es pequeño, indiferentemente de la complejidad del sistema de restricciones, es preferible utilizar el método de *backtracking*.

Capítulo 7

Conclusiones y recomendaciones

Logrando los objetivos planteados, se consiguió diseñar e implantar primero un lenguaje que facilitará la especificación de los problemas; y luego la herramienta en si (Parser, compilador e instanciador) que permite generar instancias aleatorias, de objetos descritos como entradas, cumpliendo las especificaciones de los mismos, así como también las restricciones que contienen. A partir de esta herramienta se lograron obtener soluciones a los problemas planteados para pruebas. Se pudieron obtener grupos de una, varias o incluso todas las soluciones posibles para las pruebas.

El lenguaje diseñado es cómodo y bastante expresivo en comparación con la mayoría de las librerias de **CSP** o con la implementación de un algoritmo procedural en un entorno imperativo, por lo que resulta fácil acostumbrarse a su sintaxis. Además el diseño del lenguaje permite añadir nuevas funcionalidades sin tener que hacer demasiados cambios para que se mantenga funcional.

El tiempo de ejecución de la herramienta fue para todos los ejemplos bastante breve, en donde la escritura en disco fue la causa de la mayor demora en la corrida.

Además, se lograron también buenos resultados al interpretar las respuestas como objetos finales. Para esto se probaron casos que abarcan una gran cantidad de áreas de la compu-

tación, desde las más abstractas como las bases de datos hasta las más ilustrativas como la computación gráfica.

Se analizaron las principales formas para optimizar la herramienta, se implementaron las que involucran la intervención y se especificaron las automáticas para ser implementadas en trabajos futuros.

Ahora bien, la herramienta es funcional y cumple con las expectativas iniciales, pero quedan muchas funcionalidades adicionales por ser implementadas. Para eso, a continuación se proporcionará una lista de recomendaciones para posibles trabajos futuros que reimplementen o continúen con este ambicioso proyecto:

- Utilizar un lenguaje que permita modificar fácilmente la estructura del lenguaje diseñado y de la herramienta: Esto permitirá que se puedan integrar nuevas funciones rápidamente.
- Implementar el funcionamiento de las listas: Con esto se reduciría en gran cantidad la cantidad de variables de igual comportamiento que se deben especificar como entrada.
- Implementar el funcionamiento de las funciones: Con esto se pueden aplicar restricciones mucho más complejas y que permitan expresar mejor algunos problemas.
- Añadir la posibilidad de recibir parámetros a la corrida de la herramienta: Con esto podrían pedirse respuestas para objetos similares pero con diferencias sin tener que reescribir la entrada por completo.
- Implementar la posibilidad de describir restricciones mediante disyunciones: Con esto se aumentan las posibles descripciones de los objetos pero significa aumentar en gran medida la complejidad del proceso de resolución.
- Implementar el sistema de decisión de método de resolución para los subsistemas: actualmente esto no representa una mejora significativa, pero considerando la implemen-

tación de las demás recomendaciones, este se vuelve fundamental para mantener la herramienta eficiente.

- Mejorar el motor de resolución de sistemas en *Prolog* o substituirlo con uno más complejo y eficiente. Adicionalmente, implementar otros métodos de resolución, en especial el de tipo numérico, o bien crear un motor de resolución dedicado a este tipo de problemas.

Luego de finalizado el proyecto se descubrieron una gran cantidad de posibles usos para la herramienta. Estos seguramente, son sólo una minúscula fracción de las posibilidades que se pueden generar con este proyecto. Cada una de estas son en realidad una gran cantidad de soluciones a un mismo problema. Sin tener que crear un programa específico sino un pequeño pero representativo modelo de lo que se quiere.

Se tiene la seguridad de que este trabajo puede ser la piedra angular de nuevas formas de diseño y generación de contenido. Que sean accesibles a personas de múltiples especialidades y que resuelvan cualquier cantidad de problemas. Faltan aún muchas mejoras para poder catalogar la herramienta como un producto finalizado y mucho menos un producto comercial. Aún así se tiene la expectativa de que este proyecto será continuado y mejorado. ¿Quién sabe? Quizá sirva como base para lo que en un futuro conozcamos como creatividad artificial...

Apéndice A

Archivos intermedios

A.1. Representación de Dominio

El archivo contiene los valores posibles para la variable que se codificó, el formato es valor y punto. En el caso de los *flotantes* y *doubles* el valor va entre comillas dobles. Un ejemplo del nombre del archivo para un nodo con `id 5` es: `X5.dom`. Un ejemplo del contenido del archivo suponiendo que es de tipo entero y tiene valores posibles `10`, `15` y `20`, es como el de la figura **??**.

```
1 10.
2 15.
3 20.
```

Figura A.1: Dominio Enteros

A.2. Representación de Rango

El archivo que representa el rango para una variable independiente en el problema, tiene en cada linea un valor correspondiente a la variable que se codificó. El nombre del archivo

por ejemplo para el nodo con el `id 10` es: `s_i_10.ran`. El contenido del archivo para una para una variable tipo *string* que tenga valores posibles `hola`, `hello` y `hallo`, luce como la figura **??**.

```
1 hola
2 hello
3 hallo
```

Figura A.2: Rango Independiente String

El archivo que representa el rango para un sistema, tiene los `id` de los nodos que están involucrados, todos estos se encuentran al inicio del archivo separados por saltos de linea. Luego el símbolo `#` y los valores que satisfacen el sistema separados por espacio, el orden en el que se encuentran corresponde al `id` de las variables que se encuentran al principio del archivo. El nombre del archivo por ejemplo para el sistema `0` es: `s_0.ran`. El contenido del archivo para un sistema con una variable tipo float con `id 5` y otra entera con `id 9` y vectores solución `15.0 10` y `12.0 10`, luce como la figura **??**.

```
1 5
2 9
3 #
4 15.0 10
5 12.0 10
```

Figura A.3: Rango de Sistema

A.3. Sistemas en *Prolog*

Los archivos de *Prolog* generados para resolver los sistemas lucen como el de la figura **??**.

En la sección del main lo que hace es: primero cargar un conjunto de funciones auxiliares, luego cargar los valores que estén en los archivos de dominio **??**. Al ya tener todos estos valores

```
1  probar(Vector):-
2   nth0(0,Vector,X_4),
3   nth0(1,Vector,X_8),
4   !,
5   (X_4 > 11),
6   (X_4 < X_8).
7
8  main:-
9   consult(include/prolog/funciones_csp),
10  inicializarEnteros('temp/X4.dom', X4),
11  inicializarEnteros('temp/X8.dom', X8),
12  solucionar([X4,X8],Vectores),
13  escribirVectores(Vectores, [4,8], 'temp/s_0.ran').
```

Figura A.4: Sistema para *Prolog*

cargados, se generan todas las tuplas posibles, a estos se les llamará "vectores solución", si cumplen las restricciones se imprimen estos vectores en el formato que se describió en **??**.

Lo que hace es que para el predicado probar, se toma el vector que se quiere verificar que cumpla la restricción y luego tomar de esa lista de valores que esta en `Vector` y comprobar que `X_4` y `X_8` cumplan la restricción que se codificó.

Apéndice B

Ejemplos del lenguaje

Estos ejemplos quieren ilustrar el uso de la herramienta en las distintas áreas de la computación.

B.1. Caso 1: Base de datos

En este caso se quiere crear objetos que representan a 5 personas para una base de datos cuya única restricción es que sus cédulas aparezcan de mayor a menor. Los datos que tiene cada persona pueden ser:

- La cédula que va desde 1000 a 1100.
- El nombre que puede ser alguno de estos: Juan, Pedro, Marco, Jose, Isaac, Tony, Alexis, Erick, Hancel, Alfredo o Carlos.
- Y finalmente el nombre de la bebida que le gusta, que puede ser alguna de estas: Agua, Té, Pepsi, CocaCola o Nestea.

```
1 salida personas {
2  descripcion {
```

```
3    persona p1;
4    persona p2;
5    persona p3;
6    persona p4;
7    persona p5;
8   }
9   restriccion {
10   p1.cedula > p2.cedula;
11   p2.cedula > p3.cedula;
12   p3.cedula > p4.cedula;
13   p4.cedula > p5.cedula;
14  }
15 }
16
17 aux persona {
18  descripcion {
19   int (1000, 1100) cedula;
20   Str nombre =~ ["Juan" | "Pedro" | "Jose" | "Isaac" | "Tony" |
21              "Alexis" | "Erick" | "Hancel" | "Alfredo" | "Carlos" | "Marco"];
22   Str bebida =~ ["Agua" | "Te" | "Pepsi" | "CocaCola" | "Nestea"];
23  }
24 }
```

B.2. Caso 2: Computación gráfica

Para este ejemplo se quiere crear escenarios con casas aleatorias y los datos de estas son:

- Número de pisos que pueden ser entre 1 y 3.
- Altura de los pisos que debe ser un flotante entre 2 y 3.

- Ancho y profundidad de la casa que esta entre 10 y 20.
- Un booleano que indique que si la casa tiene chimenea o no.

```
1 salida casa {
2  descripcion {
3   int (1,3) pisos;
4   float (2,3) alturaPiso;
5   float (10,20) ancho;
6   float (10,20) profundidad;
7   bool chimenea;
8  }
9 }
```

B.3. Caso 3: HTML

Para este caso lo que se quiere es probar distintas alturas de *header*, *content* y *footer* dentro de una página web, la única retricción que que tienen en común es que tengan el mismo ancho y los valores que admite estan entre 750 y 800. Las restricciones en particular cada una de estas partes son:

- *Header* tiene un alto que puede ser alguno de estos valores 100, 150, 200, 250 y 300.
- *Content* tiene un alto que puede ser entre 500 y 800.
- *Footer* tiene un alto que puede ser entre 50 y 100.

```
1 salida html {
2  descripcion {
3   header  parte1;
4   content parte2;
```

```
5     footer  parte3;
6    }
7    restriccion {
8     parte1.width == parte2.width;
9     parte1.width == parte3.width;
10   }
11  }
12  aux header {
13   descripcion {
14    int height =~ [100 | 150 | 200 | 250 | 300];
15    int (750,800) width;
16   }
17  }
18  aux content {
19   descripcion {
20    int (500,800) height;
21    int (750,800) width;
22   }
23  }
24  aux footer {
25   descripcion {
26    int (50,100) height;
27    int (750,800) width;
28   }
29  }
```

Apéndice C

Especificación formal de la sintaxis

Para reconocer el lenguaje propuesto en la sección **??**, se crearon las gramáticas que están en la sección **??** y los tokens correspondientes a estas últimas se encuentran en la sección **??**.

C.1. Gramáticas

```
1  principal:
2    definicion TKEXIT TKID TKLBRACE descripcion restriccion TKRBRACE definicion
3
4  definicion:
5    definicion TKAUXILIAR TKID TKLBRACE descripcion restriccion TKRBRACE
6    |definicion TKFUNCTIONS TKLBRACE funciones TKRBRACE
7    |
8
9  descripcion:
10   TKDESCRIPTION TKLBRACE declaracion_variable TKRBRACE
11
12 restriccion:
13   TKRESTRICTION lista_bloque_restricciones porcentaje
```

```
14    |
15
16  lista_bloque_restricciones:
17    bloque_restricciones
18    |lista_bloque_restricciones operador_logico_simple bloque_restricciones
19
20  bloque_restricciones:
21    operador_unario bloque_restricciones
22    |TKLBRACE lista_sub_bloque_restricciones TKRBRACE
23    |TKLBRACKET lista_sub_bloque_restricciones TKRBRACKET
24
25  lista_sub_bloque_restricciones:
26    lista_sub_bloque_restricciones_operados porcentaje TKSEMICOLON
27    |lista_sub_bloque_restricciones lista_sub_bloque_restricciones_operados
        porcentaje TKSEMICOLON
28
29  lista_sub_bloque_restricciones_operados:
30    sub_bloque_restricciones
31    |lista_sub_bloque_restricciones_operados operador_logico_simple
        sub_bloque_restricciones
32
33  sub_bloque_restricciones:
34    bloque_restricciones
35    |expresion
36    |distribucion
37    |cuantificador
38    |operador_unario cuantificador
39
40  declaracion_variable:
41    anulable tipo_variable rango_random TKID asignacion TKSEMICOLON
```

```
42    |declaracion_variable anulable tipo_variable rango_random TKID asignacion
          TKSEMICOLON
43
44  rango_random:
45    TKLPARENTHESES negativo TKINTVALUE TKCOMMA negativo TKINTVALUE
          TKRPARENTHESES
46    |TKLPARENTHESES negativo TKFLOATVALUE TKCOMMA negativo TKINTVALUE
          TKRPARENTHESES
47    |TKLPARENTHESES negativo TKINTVALUE TKCOMMA negativo TKFLOATVALUE
          TKRPARENTHESES
48    |TKLPARENTHESES negativo TKFLOATVALUE TKCOMMA negativo TKFLOATVALUE
          TKRPARENTHESES
49    |
50
51  negativo:
52    TKHYPHEN
53    |
54
55  anulable:
56    TKIGNORE
57    |
58
59  tipo_variable:
60    TKBOOL
61    |TKINT
62    |TKCHAR
63    |TKFLOAT
64    |TKSTRING
65    |TKDOUBLE
66    |TKVECTOR2
67    |TKVECTOR3
```

```
68    |TKVECTOR4
69    |TKLIST TKLESSTHAN tipo_variable TKMORETHAN
70    |TKID
71
72  asignacion:
73    TKEQUAL expresion
74    |TKEQUAL TKTILDE TKLBRACKET opciones TKRBRACKET
75    |TKTILDE lista_bloque_restricciones porcentaje
76    |
77
78  expresion:
79    tipos_basicos
80    |variable_mixta
81    |operador_unario expresion
82    |expresion operador_binario expresion
83    |TKLPARENTHESES expresion TKRPARENTHESES
84    |llamada_funcion
85    |TKLBRACKET elemento_lista TKRBRACKET
86    |TKLBRACKET TKRBRACKET
87    |TKLPARENTHESES expresion TKCOMMA expresion TKRPARENTHESES
88    |TKLPARENTHESES expresion TKCOMMA expresion TKCOMMA expresion TKRPARENTHESES
89    |TKLPARENTHESES expresion TKCOMMA expresion TKCOMMA expresion TKCOMMA
          expresion TKRPARENTHESES
90    |TKNULL
91    |expresion operador_logico expresion
92    |expresion operador_binario_matematico_logico expresion
93
94  variable_mixta:
95    variable_acceso
96    |variable_acceso TKLBRACKET expresion TKRBRACKET accesos
97    |TKNUMBERSIGN
```

```
98    |TKNUMBERSIGN TKDOT variable_acceso
99    |TKNUMBERSIGN TKLBRACKET expresion TKRBRACKET accesos
100   |TKNUMBERSIGN TKDOT variable_acceso TKLBRACKET expresion TKRBRACKET accesos
101
102 variable_acceso:
103   TKID
104   | variable_acceso TKDOT TKID
105
106 accesos:
107   accesos TKDOT variable_acceso TKLBRACKET expresion TKRBRACKET
108   |
109
110 operador_unario:
111   TKHYPHEN
112   |TKNEGATE
113
114 operador_binario:
115   TKHYPHEN
116   |TKPLUS
117   |TKASTERISK
118   |TKSLASH
119   |TKPERCENT
120   |TKCARET
121
122 opciones:
123   tipos_basicos porcentaje
124   |opciones TKBAR tipos_basicos porcentaje
125
126 tipos_basicos:
127   TKSTRINGVALUE
128   |TKTRUE
```

```
129     |TKFALSE
130     |TKINTVALUE
131     |TKFLOATVALUE
132     |TKDOUBLEVALUE
133     |TKCHARVALUE
134
135 porcentaje:
136     TKINTVALUE TKPERCENT
137     |TKFLOATVALUE TKPERCENT
138     |
139
140 llamada_funcion:
141     TKID TKLPARENTHESES parametro_funcion TKRPARENTHESES
142
143 parametro_funcion:
144     expresion
145     |parametro_funcion TKCOMMA expresion
146
147 operador_logico:
148     operador_logico_simple
149     |TKAND
150     |TKOR
151
152 operador_logico_simple:
153     TKEQUIVALENT
154     |TKIMPLICATION
155     |TKCONSECUENCE
156     |TKDISTINCT
157
158 operador_binario_matematico_logico:
159     TKLESSTHAN
```

```
160     |TKMORETHAN
161     |TKLESSEQUALTHAN
162     |TKMOREEQUALTHAN
163     |TKEQUAL
164
165 elemento_lista:
166     expresion
167     |elemento_lista TKCOMMA expresion
168
169 distribucion:
170     variable_mixta TKTILDE llamada_funcion
171
172 cuantificador:
173     TKFOR operador_cuantificador TKLPARENTHESES lista_variables TKRPARENTHESES
            TKTILDE lista_bloque_restricciones
174
175 operador_cuantificador:
176     TKANY
177     |TKALL
178     |TKATMOST TKINTVALUE
179     |TKATLEAST TKINTVALUE
180     |TKEXACTLY TKINTVALUE
181
182 lista_variables:
183     TKID TKFROM variable_mixta
184     |lista_variables TKCOMMA TKID TKFROM variable_mixta
185
186 funciones:
187     funcion_firma
188     |funciones funcion_firma
189
```

```
190 funcion_firma:
191   tipo_variable TKID TKLPARENTHESES firma_parametros TKRPARENTHESES TKEQUAL
          variables_aleatorias expresion_funciones
192
193 firma_parametros:
194   par_tipo_nombre
195   |
196
197 par_tipo_nombre:
198   tipo_variable TKID
199   |par_tipo_nombre TKCOMMA tipo_variable TKID
200
201 variables_aleatorias:
202   TKVARIABLES TKLBRACE declaracion_variable TKRBRACE TKIN
203   |
204
205 expresion_funciones:
206   expresion
207   |TKIF TKLPARENTHESES expresion TKRPARENTHESES TKTHEN expresion_funciones
          else_if TKELSE expresion_funciones
208
209 else_if:
210   else_if TKELSEIF TKLPARENTHESES expresion TKRPARENTHESES TKTHEN
          expresion_funciones
211   |
```

C.2. Tokens

Los tokens que se definieron para el lenguaje se clasificaron en 3 grupos.

Símbolo	Token	Símbolo	Token
{	TKLBRACE	!	TKNEGATE
}	TKRBRACE	;	TKSEMICOLON
[	TKLBRACKET	*	TKASTERISK
]	TKRBRACKET	<	TKLESSTHAN
(	TKLPARENTHESES	>	TKMORETHAN
)	TKRPARENTHESES	=	TKEQUAL
∼	TKTILDE	!=	TKDISTINCT

Cuadro C.1: Tokens de símbolos 1

Símbolo	Token	Símbolo	Token
#	TKNUMBERSIGN	,	TKCOMMA
-	TKHYPHEN	+	TKPLUS
/	TKSLASH	%	TKPERCENT
∧	TKCARET	|	TKBAR
.	TKDOT	$	TKIGNORE
==	TKEQUALEQUAL	===	TKEQUIVALENT
==>	TKIMPLICATION	<==	TKCONSECUENCE
&&	TKAND	||	TKOR
<=	TKLESSEQUALTHAN	>=	TKMOREEQUALTHAN
< −	TKFROM		

Cuadro C.2: Tokens de símbolos 2

Expresión regular	Token
$0\|[1-9][0-9]*$	TKINTVALUE
$0.[0-9]+\|[1-9][0-9]*.[0-9]+$	TKFLOATVALUE
$0.[0-9]+\|[1-9][0-9]*.[0-9]+$	TKDOUBLEVALUE
[a-zñá-úä-üA-ZÑÁ-ÚÄ-Ü_]+	TKID
“.*”	TKSTRINGVALUE
’.’	TKCHARVALUE

Cuadro C.3: Tokens de valores

Expresión regular inglés	Expresión regular español	Token
exit	salida	TKEXIT
functions	funciones	TKFUNCTIONS
aux	auxiliar	TKAUXILIAR
description	descripcion	TKDESCRIPTION
restriction	restriccion	TKRESTRICTION
entero	int	TKINT
bool	bool	TKBOOL
char	caracter	TKCHAR
float	flotante	TKFLOAT
Double	Doble	TKDOUBLE
Str	Caracteres	TKSTRING
vector2	vector2	TKVECTOR2
vector3	vector3	TKVECTOR3
vector4	vector4	TKVECTOR4
list	lista	TKLIST
for	para	TKFOR
any	algun	TKANY
all	todos	TKALL
at most	a lo sumo	TKATMOST
at least	al menos	TKATLEAST
exactly	exactamente	TKEXACTLY
variables	variables	TKVARIABLES
in	en	TKIN
if	si	TKIF
then	entonces	TKTHEN
elseif	si_en_vez	TKELSEIF
else	si_no	TKELSE
null	nulo	TKNULL
true	verdadero	TKTRUE
false	falso	TKFALSE

Cuadro C.4: Tokens de palabras reservadas