ANÁLISIS, DISEÑO E IMPLEMENTACIÓN DE UN PROTOTIPO DE UN LENGUAJE DE DOMINIO ESPECÍFICO (DSL) INTERNO, ORIENTADO AL MODELADO DE PROBLEMAS DE PROGRAMACIÓN LINEAL

BRAHYAM PINEDA CARDONA
1088304027
JORGE HERNÁN RIVERA MALDONADO
1088304075

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
PEREIRA
2015

ANÁLISIS, DISEÑO E IMPLEMENTACIÓN DE UN PROTOTIPO DE UN LENGUAJE DE DOMINIO ESPECÍFICO (DSL) INTERNO, ORIENTADO AL MODELADO DE PROBLEMAS DE PROGRAMACIÓN LINEAL

BRAHYAM PINEDA CARDONA
1088304027
JORGE HERNÁN RIVERA MALDONADO
1088304075

Trabajo de grado presentado como requisito para optar al título de INGENIERO DE SISTEMAS Y COMPUTACIÓN

Director
Alejandro Rodas Vásquez

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
PEREIRA
2015

NOTA DE ACEPTACIÓN:

_____
_____
_____
_____
_____
_____
_____

FIRMA DEL JURADO

_____

Pereira, 12 de junio de 2015

AGRADECIMIENTOS

En primer lugar queremos agradecer a Dios por brindarnos la posibilidad de estar culminando un paso más en nuestras vidas, a nuestros padres porque sin ellos nada de esto sería posible. Su constante apoyo incondicional y la educación ejemplar que nos han dado desde nuestro primer día ha sido fundamental para nuestro proyecto de vida.

También queremos agradecer a todos y cada uno de nuestros profesores que han sido partícipes de nuestra formación académica, asimismo queremos agradecer a nuestros compañeros de estudio y en general a todas las personas que han influido en nuestra formación, especialmente a los ingenieros Jorge Iván Ríos Patiño y Alejandro Rodas, que nos han compartido su conocimiento y experiencias para poder culminar de forma exitosa esta etapa de la universidad.

DEDICATORIA

Esto va dedicado a todas aquellas personas que siempre confiaron en nuestras capacidades y que intervinieron de alguna manera para que pudiéramos lograr este objetivo tan importante para nosotros, haciendo un reconocimiento especial a nuestros padres, ellos son nuestra principal motivación.

TABLA DE CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.

LISTA DE TABLAS

Pág.

1. PRESENTACIÓN DEL PROYECTO

1.1 Resumen

Este proyecto de grado explica el conjunto de procesos llevados a cabo para la construcción de un Lenguaje de Dominio Específico interno, mediante el establecimiento de ciertas pautas que posibilitan su implementación bajo cualquier dominio. De esta forma se exponen las características que debe tener un lenguaje anfitrión, las técnicas usadas para el análisis del dominio objetivo y los pasos para su implementación.

Se toma como ejemplo de desarrollo el dominio de la programación lineal, dado que su amplio campo de aplicación en diferentes áreas permite evidenciar la productividad que conlleva desarrollar Lenguajes de Dominio Específico internos.

Por lo tanto, este documento pretende exhortar al usuario a desarrollar Lenguajes de Dominio Específico Internos y así mismo, ser una guía para su desarrollo.

1.2 Introducción

Actualmente existe una gran variedad de lenguajes de programación, usados para resolver cualquier tipo de problema computacionalmente soluble. De acuerdo a su enfoque, se dividen en dos tipos: Lenguajes de propósito general (GPL[1]), y lenguajes de dominio específico (DSL[2]). Los lenguajes de propósito general posibilitan la abstracción y representación de cualquier problema en diferentes dominios. Sin embargo, para hacer uso de ellos es necesario conocer conceptos básicos como algoritmia, sintaxis y el paradigma del lenguaje, además de nociones básicas de matemáticas. Esto conlleva a que programar y entender el código sea una tarea difícil para una persona con pocos conocimientos en informática.

Otro problema que poseen los GPL's es que pueden presentar inconvenientes al intentar solucionar un problema de un área concreta, debido a que podría ser más complejo para estos abstraer características específicas del problema, pues las abstracciones que se plantean podrían no estar soportadas nativamente por el GPL, siendo necesario crearlas desde cero. Esto requiere de mayor esfuerzo y tiempo en la implementación. Además, se podría perder eficiencia en la solución si el lenguaje

[1] Por su nombre en inglés General Purpose Language

[2] Por su nombre en inglés Domain Specific Language

no posee el soporte y las librerías necesarias, lo que obligaría a adaptar el dominio del problema a lo que ofrece el GPL, perdiendo flexibilidad.

Como menciona Fowler[3] un Lenguaje de Dominio Específico es un lenguaje de programación de expresividad limitada enfocado en un dominio particular. A diferencia de un lenguaje de propósito general, éste proporciona una abstracción que permite pensar solo en un área específica. Tal abstracción es muy valiosa, gracias a que permite expresar el comportamiento de un dominio mucho más fácilmente que si se piensa en términos de construcciones de menor nivel, es decir, más cercanas al nivel de hardware.

Al igual que los GPL's, los DSL's también son muy usados, sin embargo, el concepto DSL fue adoptado hace poco, incluso muchos desarrolladores consideran que algunos DSLs (como HTML o CSS) no cumplen las características necesarias para considerarse un lenguaje de programación. Estos lenguajes son una buena solución para los expertos de un dominio, ya que manejan una sintaxis muy limitada y similar al dialecto del experto. Esta característica ayuda a que su uso sea más fácil e intuitivo que el uso de un GPL, facilitando la creación de nuevas soluciones, simplificando código y minimizando tiempo, permitiendo a los expertos modelar soluciones fácilmente, eliminando así la barrera de tener que aprender conceptos más complejos para utilizar un GPL.

De acuerdo a la forma en que un DSL se puede implementar, existen dos tipos, interno y externo. Un DSL interno se desarrolla en un lenguaje de propósito general (lenguaje anfitrión) y utiliza las propiedades que este lenguaje ofrece; el DSL externo por otra parte, se construye desde cero, teniendo que implementar cada uno de los módulos que componen un compilador (analizador léxico, analizador sintáctico, analizador semántico, generador de código intermedio, optimizador de código independiente de la máquina y generador de código). También se debe crear el IDE en caso de ser requerido.

1.3 Definición del problema

Día a día entidades de diferentes áreas de la sociedad se deben enfrentar a problemas relacionados con la toma de decisiones, estos problemas tienen que ver con la optimización de bienes para obtener mayores beneficios. Gracias al uso de poderosas y cada vez más sofisticadas herramientas tecnológicas, es posible la sistematización y solución de estos.

[3] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 27.

Sin embargo, muchas de estas herramientas no están enfocadas en el usuario, ya que su funcionalidad es tediosa y poco intuitiva, además su mantenimiento es complejo. Por este motivo, se detectó la necesidad de desarrollar un prototipo de un DSL interno como una alternativa que permita modelar problemas de programación lineal en forma ágil, con una sintaxis cercana a su notación matemática.

Por otro lado, durante la investigación se identificó que los DSLs internos, al ser relativamente nuevos, no cuentan con suficiente documentación en español sobre sus características y las metodologías existentes para su desarrollo, además del potencial que estos poseen para modelar soluciones con alto nivel de usabilidad en áreas específicas. Esta es una de las razones por la cual muchas personas y entidades del ámbito de la optimización desconocen estas bondades y generalmente acuden al uso de otras alternativas como las hojas de cálculos (Excel), las bibliotecas de algoritmos especializados para lenguajes de propósito general, o el uso de entornos de cálculo numérico y/o simbólico como Matlab o Mathematica.

Por lo tanto, se detectó la necesidad de redactar un documento que dé a conocer las principales características y metodologías de desarrollo que hacen que un DSL sea una solución viable y eficiente para muchos profesionales y entidades. Así mismo, que este documento sirva como base para la construcción de DSLs internos que satisfagan de manera ágil sus necesidades.

1.4 Justificación

La optimización está presente en diferentes ámbitos de la sociedad, tales como las grandes industrias, el sector educativo, estadístico, científico, económico, entre muchos otros. Por esta razón, constantemente se busca desarrollar soluciones informáticas que permitan mejorar las soluciones actuales y obtener resultados más óptimos que ayuden a conseguir mayores beneficios para quienes hacen uso de estas herramientas. Para alcanzar este objetivo, existen diferentes técnicas de las ciencias de la administración.

Según Chediak[4], como parte de una investigación sobre practicantes en el gobierno, la industria y la academia, Shannon, Long y Buckles pidieron a administradores en ejercicio que señalaran si estaban familiarizados con los diversos métodos cuantitativos y si habían utilizado o no esos métodos en aplicaciones específicas. En la Tabla 1 se encuentran los resultados de dicha investigación sobre las diferentes técnicas de la ciencia de la administración.

[4] Francisco Chediak. Investigación De Operaciones. 2004, p. 234.

Tabla 1. Clasificación de las técnicas de ciencia de la administración más usadas

MÉTODO	RANGO DE CONOCIMIENTO	USO (%)
Programación Lineal	1	83,8
Simulación	2	80.3
Análisis de redes	3	58,1
Teoría de las colas	4	54,7
Árboles de decisión	5	54,7
Programación según enteros	6	38,5
Análisis de reposición	7	38,5
Programación dinámica	8	32,5
Procesos de Markov	9	31,6
Programación no lineal	10	30,7
Programación de metas	11	20,5
Teoría de los juegos	12	13,7

Fuente: Francisco Chediak. Investigación De Operaciones Volumen 1. Página 234.

Como se puede apreciar de la tabla anterior, la programación lineal es la técnica más usada. Existen lenguajes de dominio específico que permiten modelar problemas de este tipo, sin embargo, la mayoría no presentan una sintaxis intuitiva y cercana a la que manejan los expertos en programación lineal.

Crear un lenguaje de programación de dominio específico, intuitivo y entendible por los expertos del área de la optimización, con una gramática que exprese plenamente el dominio de la programación lineal y ocultando complejidades innecesarias, facilita la implementación y lectura de nuevas soluciones a problemas cotidianos y contribuye a la mejora continua de la productividad, confiabilidad y calidad de dichas soluciones.

Por tal motivo se desarrollará un DSL que cumpla con estas características. El DSL a desarrollar será de tipo interno, gracias a que ofrece ventajas como la facilidad de implementación sobre un lenguaje anfitrión, obteniendo los beneficios derivados de éste, por ejemplo las ayudas y el soporte asociados al uso de un IDE y/o el uso de su compilador o intérprete, evitando así tener que crear uno nuevo, como sucede en caso de desarrollar un DSL externo.

En la Tabla 2 se pueden ver las diferencias entre modelar un problema de programación lineal en el DSL propuesto y modelar el mismo problema en el GPL Java.

Tabla 2. Comparativa entre un DSL y un GPL para modelar un problema lineal.

Código en el DSL interno	LinearProgramming
	  .variables('x','y')
	  .max('15x + 10y')
	  .subjectTo(
	  '1/3x + 1/2y <= 100',
	  '1/3x + 1/6y <= 80')

Código en el GPL Java	package Optimization;
	public class Main{
	  public static void main(String[] args){
	  LinearProgramming model1 = new LinearProgramming();
	
	  model1.addVariable("x");
	  model1.addVariable("y");
	
	  model1.objectiveFunction("max", "15x + 10y");
	  model1.addRestriction("1/3x + 1/2y <= 100");
	  model1.addRestriction("1/3x + 1/6y <= 80");
	  }
	}

Fuente: Los autores

En el código de la Tabla 2 se puede apreciar que en el DSL interno la sintaxis es más cercana al lenguaje natural del experto, es intuitiva, se presenta poco ruido sintáctico, y permite modelar el mismo problema en menos líneas de código.

El lenguaje en el que se desarrollará el DSL interno (lenguaje anfitrión) es Ruby. Se eligió este lenguaje gracias a que cuenta con una serie de características que lo hacen un lenguaje idóneo para desarrollarlo: sintaxis flexible, la posibilidad de redefinir operadores (sobrecarga de operadores), es dinámicamente tipado, soporta metaprogramación. Todos estos conceptos serán explicados con más detalle en el capítulo 2.

Por todo lo anterior, la creación de este DSL interno servirá a la comunidad, pues se investigará y se dejarán aportes significativos plasmados como guía para la

creación de DSLs internos. Además se podrá hacer uso del DSL en forma gratuita, tanto en el ámbito educativo como en el profesional. Por último, es importante tener en cuenta que crear software cada vez más cercano al humano, con los conceptos de usabilidad e interacción persona-computador, hará que la brecha en la transferencia de información entre humano y máquina sea cada vez más pequeña.

1.5 Objetivos

1.5.1 Objetivo general

Analizar, diseñar e implementar un prototipo de un Lenguaje de Dominio Específico (DSL) interno, orientado al modelado de problemas de programación lineal.

1.5.2 Objetivos específicos

- Investigar y dejar aportes significativos al campo de investigación de DSLs internos.
- Investigar acerca de la programación lineal y su campo de aplicación.
- Analizar y establecer el dominio del lenguaje.
- Investigar las características que deben tener los lenguajes anfitriones para facilitar la implementación de un DSL interno.
- Analizar requerimientos y metodologías para desarrollar el DSL interno.
- Establecer la sintaxis y las restricciones del dominio adecuadas.
- Implementar el DSL.
- Planificar y desarrollar los casos de prueba.
- Ejecutar y analizar los casos de prueba para comprobar el funcionamiento del DSL.

1.6 Marco de referencia

1.6.1 Marco histórico

“En 1966 Landin publicó The Next 700 Programming Languages en el que presentaba el lenguaje de programación ISWIM, o "If you See What I Mean", o "Si comprendes lo que quiero decir". En el artículo se sentaban las bases para los lenguajes específicos de dominio”[5].

Fortran y Cobol fueron los primeros lenguajes de programación y fueron también los primeros lenguajes de dominio específico, Fortran se usaba principalmente para el desarrollo de aplicaciones científicas y el análisis numérico, mientras que Cobol estaba orientado a aplicaciones financieras y empresariales; es decir que ambos se centraban sólo en resolver problemas matemáticos, así que ambos eran lenguajes de modelado aritmético. Fortran tuvo varias versiones y en la actualidad es un lenguaje de propósito general; Cobol por otra parte, aunque se le añadieron nuevas características, sigue siendo un lenguaje de propósito específico.

A partir de la década que corresponde al año 1960 se crearon e hicieron populares los lenguajes de programación de propósito general (GPL) como C (1972) y Java (1995), que permiten modelar problemas de cualquier dominio; sin embargo, en la actualidad las necesidades de la población son diferentes y se requieren lenguajes centrados en el usuario, es por ello que se están desarrollando DSLs para crear soluciones más óptimas en diferentes áreas. En el área de la optimización se creó la alternativa de los AMLs (Algebraic Modeling Languages). Según Kuip[6] los AML “Indican un problema de programación matemática en una forma que se corresponde estrechamente con su notación algebraica”.

Así mismo, Hermann Schichl[7] expresa que los lenguajes de modelado algebraico son una clase especial de lenguaje declarativo ya que los problemas están representados de forma declarativa. Presentan una clara separación entre la definición del problema y el proceso de solución, y también hay una clara separación entre la estructura del problema y sus datos. Por otra parte, el lenguaje de modelado algebraico es responsable de la creación de una instancia de problema que un algoritmo de solución puede trabajar, estos algoritmos de solución se denominan

---

[5] VIEIRO.NET. Peter J. Landin y los 700 próximos lenguajes de programación. [en línea]. 2015. Disponible en internet. <http://vieiro.net/blog/20130301.html/>

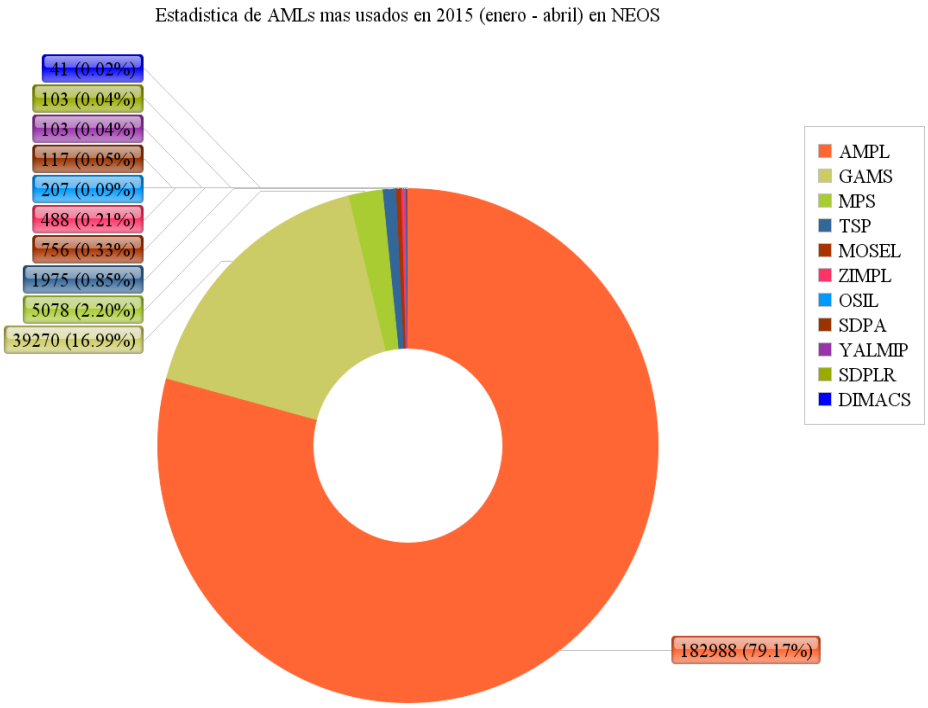
[6] C.A.C. Kuip “Algebraic Languages for Mathematical Programming” European Journal of Operational Research 67 (1993) 25 -51

[7] Hermann Schichl “THEORETICAL CONCEPTS AND DESIGN OF MODELING LANGUAGES FOR MATHEMATICAL OPTIMIZATION” Institut f¨ur Mathematik der Universit¨at Wien - Strudlhofgasse 4, A-1090 Wien 2004

solvers, existen muchos de código abierto y son usados por los más populares lenguajes de modelado algebraico.

Los lenguajes de modelado algebraico más usados hasta el 13 de abril del año 2015 se muestran en la siguiente Figura 1.

Figura 1. Diagrama de AMLs más usados en 2015 (desde enero 1 hasta abril 13) en NEOS



Fuente: Generada por los autores basados en las estadísticas del servidor de AMLs NEOS[8].

A continuación se citarán los dos lenguajes de modelado algebraico más usados según la Figura 1, ya que entre ambos (AMPL y GAMMS) acaparan el 96,06% del mercado (según este indicador).

Chen[9] menciona que el lenguaje de modelado GAMS (General Algebraic Modeling System) fue desarrollado en el Banco Mundial (en el año 1988) para facilitar la

[8] NEOS_SOLVER. Neos Solver Access Statistics [Estadísticas de acceso a Neos Solver]. [en linea]. 13-04-2015. Disponible en internet < http://www.neos-server.org/neos/report.html >

[9] Xueyu Chen, Krishnaraj S. Rao, Jufang Yu and Ralph W. Pike “Comparison of gams, ampl, and minos for optimization” Louisiana State University - Baton Rouge, AL 70803

solución de los modelos de toda la economía-multisectorial que se habían utilizado previamente en el lenguaje FORTRAN. Por otro lado, el lenguaje de modelado AMPL (A Mathematical Programming Language) fue desarrollado en los laboratorios AT&T Bell (en el año 1993) para aplicaciones de comunicación. Estos dos lenguajes ofrecen una manera eficiente y eficaz para resolver problemas de programación matemática a expensas de aprender otro lenguaje de programación.

1.6.2 Marco teórico

1.6.2.1 Tipos de DSLs

Un lenguaje de dominio específico se puede crear de dos formas, a partir de otro lenguaje de programación (DSL interno) y/o desde cero (DSL externo).

Un DSL externo es un lenguaje nuevo, con sintaxis muy próxima al lenguaje natural del experto del dominio, eliminando así la presencia de ruido sintáctico. Este tipo de DSL debe tener su propio compilador o intérprete, lo que genera costo extra en la implementación.

Según Calle Lejdfors[10] Un enfoque para reducir el costo inicial en la implementación de un lenguaje de dominio específico es reutilizando la infraestructura de un lenguaje existente. Un DSL interno, también conocido como interfaz fluida o API fluida, es un lenguaje que se desarrolla en un lenguaje de propósito general, [11]todas las instalaciones y la infraestructura del entorno de acogida pueden apropiarse al DSL, y la familiaridad con las convenciones sintácticas y herramientas de la lengua de acogida pueden ser transferidas al DSL. Sin embargo, no todos los lenguajes de propósito general son óptimos para construir un DSL interno; se recomienda usar aquellos que ofrecen gran flexibilidad, azúcar sintáctica, no sean fuertemente tipados, con posibilidades de formato y metaprogramación; por ejemplo Ruby y Scala. Existen varios patrones para desarrollar un DSL interno como el encadenamiento de métodos y las funciones anidadas.

[10] Calle Lejdfors “Techniques for implementing embedded domain specific languages in dynamic languages” 2006

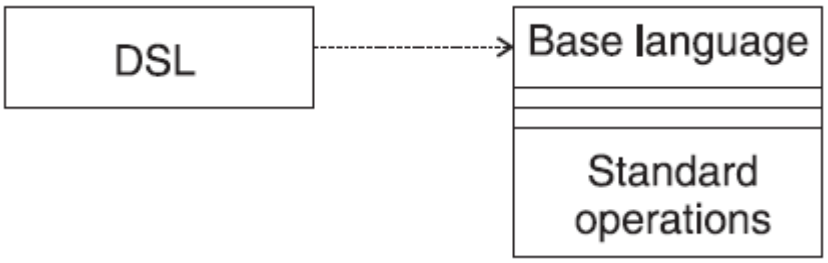
[11] Jeremy Gibbons and Nicolas “WuFolding Domain-Specific Languages: Deep and Shallow Embeddings” Department of Computer Science, University of Oxford

1.6.2.2 Patrones estructurales de DSLs

De acuerdo a la arquitectura que poseen los DSLs, Diomidis Spinellis propuso patrones que permiten su reutilización, estos patrones son los siguientes:

A cuestas[12]: Este patrón estructural utiliza las capacidades de un lenguaje existente como base de alojamiento de un nuevo DSL[13].

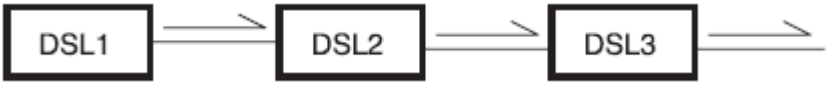
Figura 2. Patrón a cuestas



Fuente: Notable design patterns for domain-specific languages. p. 3.

Tubería[14]: Este patrón de comportamiento resuelve un problema mediante la composición de DSLs. A menudo, un sistema puede ser mejor descrito usando una familia de DSLs[15].

Figura 3. Patrón tubería



Fuente: Notable design patterns for domain-specific languages. p. 4.

Procesamiento léxico[16]: Este patrón creacional ofrece una forma eficiente para diseñar e implementar DSLs, el diseño del DSL se orienta hacia la traducción léxica

[12] Por su nombre en inglés Piggyback

[13] Diomidis Spinellis "Notable design patterns for domain-specific languages" Department of Information and Communication Systems, University of the Aegean, GR-83 200 Karlovasi, Greece. 14 February 2000, p. 3.

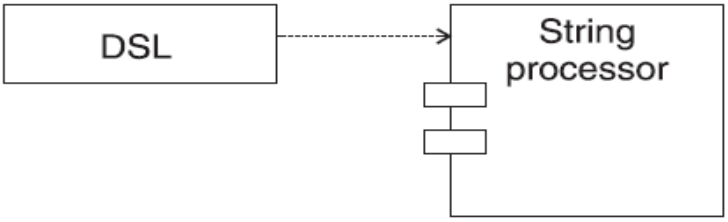
[14] Por su nombre en inglés Pipeline

[15] Spinellis, Op. Cit., p. 4.

[16] Por su nombre en inglés Lexical processing

mediante la utilización de una notación basada en pistas léxicas tales como la especificación de elementos de lenguaje (por ejemplo, variables) utilizando caracteres especiales de prefijo o sufijo[17].

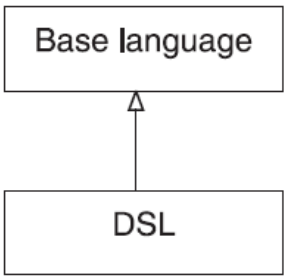
Figura 4. Patrón procesamiento léxico



Fuente: Notable design patterns for domain-specific languages. p. 5.

Extensión del lenguaje[18]: Este patrón creacional se utiliza para agregar nuevas características a un lenguaje existente. A menudo un lenguaje existente puede cumplir eficazmente una nueva necesidad, con la adición de algunas nuevas características a su funcionalidad. En este caso, un DSL puede ser diseñado e implementado como una extensión del lenguaje base[19].

Figura 5. Patrón extensión del lenguaje



Fuente: Notable design patterns for domain-specific languages. p. 5.

[17] Diomidis Spinellis “Notable design patterns for domain-specific languages” Department of Information and Communication Systems, University of the Aegean, GR-83 200 Karlovasi, Greece. 14 February 2000, p. 4.

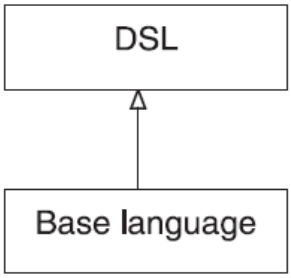
[18] Por su nombre en inglés Language extension

[19] Spinellis. Op. cit., p. 5.

.

Socialización del lenguaje[20]: Este patrón elimina características del lenguaje base para formar un DSL. El DSL es diseñado e implementado como un subconjunto de un lenguaje existente. Cuando algunas de las características específicas de un lenguaje existente lo hacen inadecuado para una aplicación dada, el diseño de un DSL siguiendo este patrón puede resultar en un lenguaje maduro que satisface los requisitos dados[21].

Figura 6. Patrón socialización del lenguaje



Fuente: Notable design patterns for domain-specific languages. p. 6.

Transformación fuente a fuente[22]: Este patrón permite la implementación eficiente de los traductores DSL. Cuando el DSL no puede ser diseñado por los patrones extensión del lenguaje, socialización o a cuestas, a menudo es posible aprovechar las facilidades proporcionadas por herramientas lingüísticas existentes utilizando una técnica de transformación de origen a origen. El código fuente DSL se transforma a través de un adecuado proceso de traducción superficial o profunda en el código fuente de un lenguaje existente[23].

Figura 7. Patrón transformación fuente a fuente

Fuente: Notable design patterns for domain-specific languages. p. 6.

[20] Por su nombre en inglés Language specialisation

[21] Diomidis Spinellis "Notable design patterns for domain-specific languages" Department of Information and Communication Systems, University of the Aegean, GR-83 200 Karlovasi, Greece. 14 February 2000, p. 5.

[22] Por sus siglas en inglés: Source-to-source transformation

[23] Spinellis. Op. cit., p. 6.

Representación de estructura de datos[24]: Este patrón permite la especificación declarativa y de dominio específico de datos complejos. Estructuras complicadas son mejor expresadas usando un lenguaje que su representación subyacente. El diseño de un DSL para representar los datos ofrece una solución atractiva para el problema. El patrón es usado cada vez que una estructura de datos no trivial necesita ser inicializada con datos. Es particularmente aplicable a la inicialización de las estructuras de datos cuyos elementos se interrelacionan como árboles, grafos, vectores de punteros para inicializar estáticamente elementos de estructuras, arreglos de apuntadores a funciones y elementos de texto en varios idiomas[25].

Figura 8. Patrón representación de estructura de datos

Fuente: Notable design patterns for domain-specific languages. p. 6.

Sistema frontal[26]: La configuración y adaptación de un sistema a menudo pueden ser relegados a un front-end DSL. Hacer el sistema programable mediante este patrón estructural ofrece un mecanismo declarativo, mantenible, organizado y abierto para su configuración y adaptación. Los sistemas con pocas opciones de configuración y los sistemas cuyo funcionamiento no se puede especificar de manera adecuada mediante algunos argumentos o una interfaz gráfica de usuario normalmente se benefician con la adición de un DSL front-end[27].

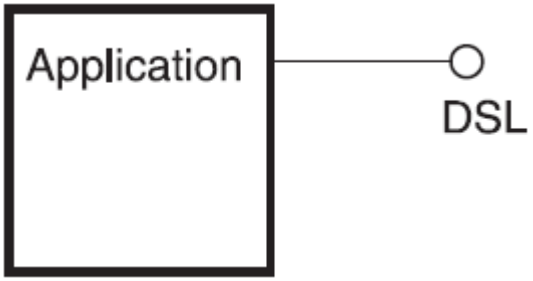
[24] Por su nombre en inglés Data structure representation

[25] Diomidis Spinellis “Notable design patterns for domain-specific languages” Department of Information and Communication Systems, University of the Aegean, GR-83 200 Karlovasi, Greece. 14 February 2000, p. 6.

[26] Por su nombre en inglés System front-end

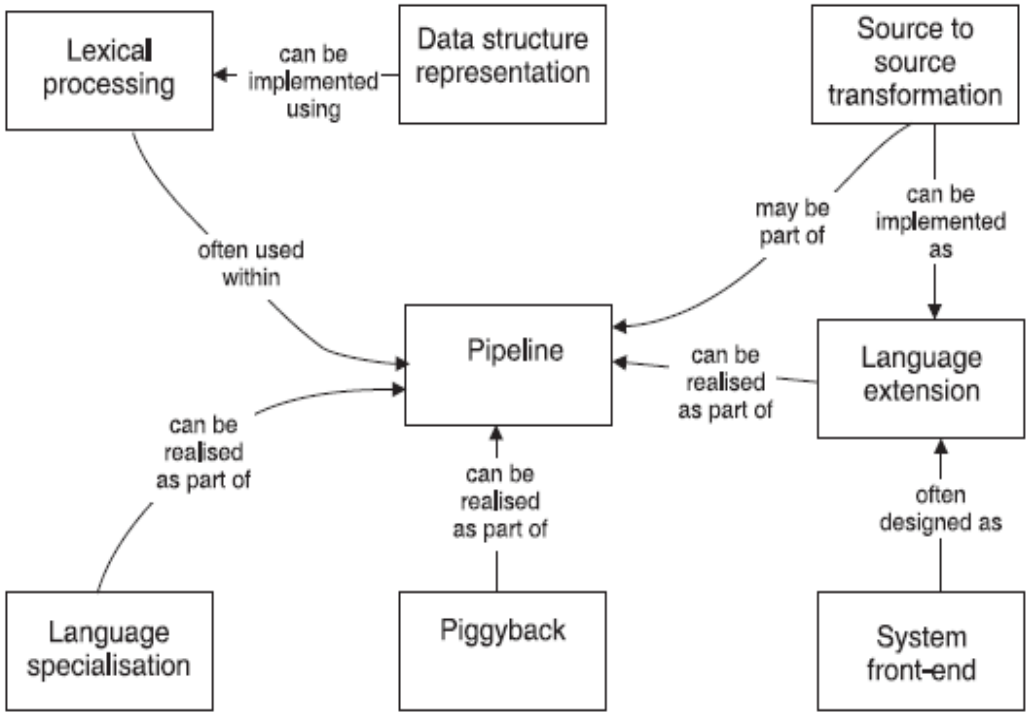
[27] Spinellis. Op. cit., p. 6.

Figura 9. Sistema frontal



Fuente: Notable design patterns for domain-specific languages. p. 7.

Figura 10. Relación entre patrones de DSL



Fuente: Fuente: Notable design patterns for domain-specific languages. p. 7.

1.6.2.3 Etapas del desarrollo de DSLs

Sebastian Günther[28] postuló un proceso genérico para desarrollar DSLs, el cual se compone de los siguientes pasos:

[28] Sebastian Günther. Agile DSL-Engineering with Patterns in Ruby. Faculty of Computer Science, University of Magdeburg. 2009, p. 3.

1. Analizar el dominio y crear un modelo de dominio.

2. Definir los requisitos del lenguaje. Esto incluye la sintaxis concreta, el tipo de DSL (gráfico, textual) y los criterios generales de generación de código (lenguaje anfitrión, framework).

3. Definir e implementar los generadores de código necesarios. Esto incluye el análisis del lenguaje anfitrión, el lenguaje objetivo, y el mapeo entre sus expresiones.

4. Generar el código de la aplicación, comprobar la exactitud de las transformaciones de código y usar el código en producción.

Según Mernic[29] en la fase de análisis del desarrollo del DSL, el dominio del problema se identifica y el conocimiento del dominio se recopila. La mayoría de veces, el análisis de dominio se hace de manera informal, pero a veces se utilizan metodologías de análisis de dominio tales como DARE (Domain Analysis and Reuse Environment)[30], DSSA (Domain-Specific Software Architectures)[31], FODA (Feature-Oriented Analysis Domain)[32], ODM (Organization Domain Modeling)[33] y DDD (Domain-Driven Design)[34].

Debasish Ghosh[35] explica que después de definir el problema del dominio (definir el modelo), los elementos del problema del dominio se deben mapear en los artefactos apropiados en la solución de dominio, es decir que se debe pasar del gráfico que representa al modelo del dominio a código en algún lenguaje de programación. El proceso mencionado se representa gráficamente en la Figura 11.

---

[29] Marjan Mernik, Jan Heering and Anthony M. Sloane "When and How to Develop Domain-Specific Languages" ACM 2005, p. 8.

[30] [W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain analysis and reuse environment. Annals of Software Engineering, 5:125–141, 1998.

[31] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures. ACM SIGSOFT Software Engineering Notes, 20(5):27–37, 1995.

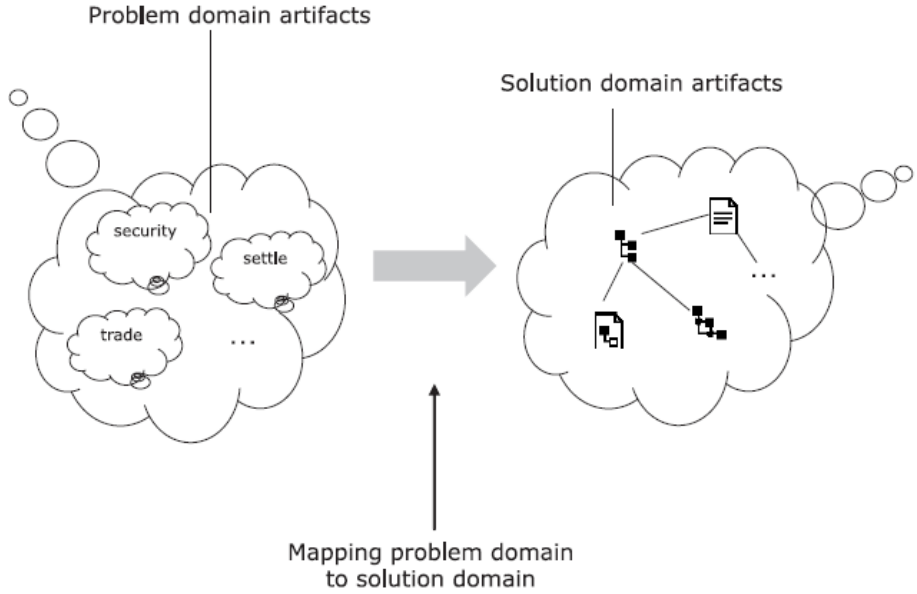
[32] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[33] M. Simos and J. Anthony. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In Proceedings of the Fifth International Conference on Software Reuse, pages 94–102. IEEE Computer Society, 1998.

[34] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[35] Debasish Ghosh. DSLs in action. Manning Publications Co, 2011, p. 4.

Figura 11. Problema del dominio y solución del dominio

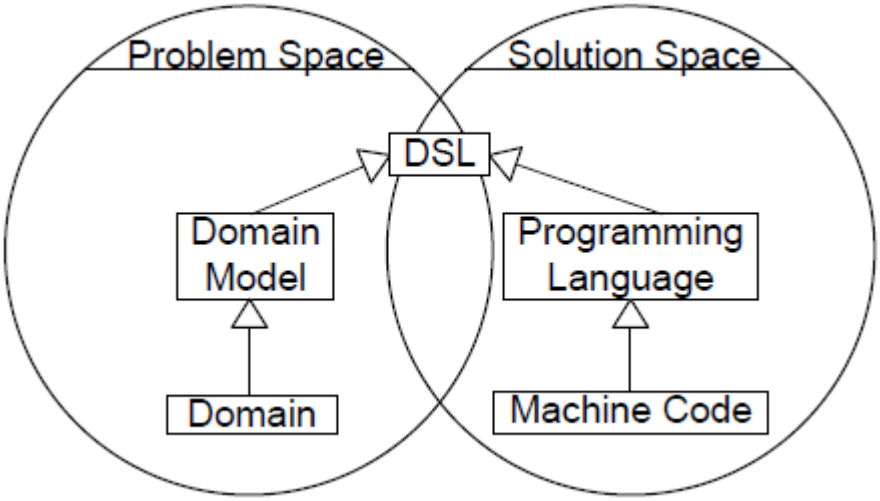


Fuente: Debasish Ghosh. DSLs in action. Manning Publications Co, 2011. p. 5.

Dicho de otra forma, “en el espacio del problema, los conceptos del dominio y sus relaciones se representan como un modelo de dominio. En el espacio de soluciones, los conceptos de dominio se representan típicamente como objetos y métodos de un lenguaje de programación, u otras construcciones adecuadas. Esto facilita probar, construir y desplegar aplicaciones”[36].

[36] Sebastian Günther. Development of Internal Domain-Specific Languages: Design Principles and Design Patterns, 2011, p. 2.

Figura 12. Espacio del problema y espacio de solución



Fuente: Sebastian Günther. Development of Internal Domain-Specific Languages: Design Principles and Design Patterns, 2011. p. 2.

Tal como se aprecia en la Figura 12 “el DSL cierra la brecha entre el espacio del problema y el espacio de soluciones, proporcionando abstracciones que unen ambos espacios”[37].

1.6.3 Marco conceptual

Los conceptos claves desarrollados en el presente proyecto son los siguientes:

Lenguaje de propósito general (GPL): Son lenguajes de programación dedicados a resolver problemas de diversos propósitos. Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

Lenguaje de dominio específico (DSL): Es un lenguaje de programación orientado a resolver un problema en particular, presenta las herramientas necesarias para dar solución a este, posee un alto nivel de abstracción para el dominio del problema, y ofrece notaciones y construcciones específicas de dominio las cuales permiten expresar la solución más claramente que los lenguajes de propósito general (GPL).

[37] Sebastian Günther. Development of Internal Domain-Specific Languages: Design Principles and Design Patterns, 2011, p. 2.

Compilador: Un compilador es un programa de software que se encarga de traducir un código escrito en un lenguaje de programación a otro lenguaje (generalmente lenguaje máquina) que entiende una máquina y es capaz de interpretarlo y ejecutarlo. Dentro de las funciones de un compilador están las de analizar la sintaxis, revisar la semántica y optimizar el código.

Optimización: Consiste en determinar un valor máximo o un valor mínimo en una función de una variable, teniendo en cuenta que incluye un conjunto de variables relacionadas con el problema, es decir, la optimización permite obtener los mejores valores entre un conjunto de elementos.

Programación lineal: Es una rama de la programación matemática orientada a la resolución de problemas de optimización, donde la función objetivo define la maximización o minimización de un modelo de programación lineal. Además cuenta con un conjunto de restricciones que limitan o reducen el grado en el que se puede buscar el objetivo. Su función principal es hacer uso eficiente de los recursos de la forma más óptima buscando alcanzar el objetivo deseado.

Entorno de desarrollo integrado (IDE): Se define como un programa informático compuesto por un conjunto de herramientas de programación. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien puede utilizarse para varios. Entre las herramientas se encuentra un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

Solver: Es la implementación en algún lenguaje de programación de un algoritmo de resolución de problemas de programación lineal cómo Simplex o Garfinkel.

Ruby: Es un lenguaje de programación interpretado, reflexivo y orientado a objetos que combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk. Este lenguaje fue diseñado con el objetivo de mejorar la productividad y la diversión del desarrollador, siguiendo los principios de una buena interfaz de desarrollo.

Dominios: "Aquellos campos del saber caracterizados por la homogeneidad de su objeto de conocimiento, una común tradición histórica y la existencia de comunidades de investigadores, nacionales o internacionales"[38].

[38] Universitat de Barcelona. CATÁLOGO DE ÁREAS DE CONOCIMIENTO. [en línea]. 2015. <http://www.ub.edu/farmacia/doctorat/pdf/areas_conocimiento.pdf>

Modelo: Como menciona[39] un modelo es una simplificación, es una interpretación de la realidad que abstrae los aspectos relevantes para la solución del problema en cuestión e ignora detalles superfluos, por tanto, esa área temática a la que el usuario aplica el programa es el dominio del software. Así pues, un modelo adecuado le da sentido a la información y la enfoca en un problema.

Ruido sintáctico: Son elementos en el lenguaje que afectan a la forma de la información, elementos que desde la perspectiva del programador son superfluos y que pueden hacer tediosa la tarea de codificar, por ejemplo, el punto y coma (;) que utilizan lenguajes como C y Java para especificar el fin de una sentencia.

Azúcar sintáctica: Se refiere a algunas características de la sintaxis de un lenguaje de programación que no afectan a su funcionalidad, pero que facilitan expresar algunas construcciones de una forma más clara o concisa, o en un estilo alternativo.

Interfaz fluida: Es el patrón por medio del cual se logra que la sintaxis del DSL sea muy próxima al lenguaje natural. La interfaz fluida es lo que diferencia a una API de un DSL.

Alfabeto: Según Kelley[40] un alfabeto es el conjunto de símbolos que se utiliza en un lenguaje determinado para construir las “palabras” o “cadenas” que pertenecen a éste. El conjunto tiene que ser no vacío y finito.

Palabra: “Es una secuencia finita de símbolos de determinado alfabeto”[41].

Lenguaje: “Es un conjunto de “palabras” escritas sobre algún alfabeto en cuestión”[42].

Token: Es un par que consiste en un nombre de token y un valor de atributo opcional, el nombre del token es un símbolo abstracto que representa un tipo de unidad léxica.

Expresión regular: Sirve para describir a todos los lenguajes que pueden construirse de operaciones aplicadas a los símbolos de cierto alfabeto.

---

[39] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003, p. 24.

[40] Dean Kelley. Teoría de autómatas y lenguajes formales. Prentice Hall, 1995. p. 29.

[41] Íbid., p. 29.

[42] Íbid., p. 29.

Gramática: Sirve para describir en forma sistemática la sintaxis de las construcciones de un lenguaje de programación, cómo las expresiones o las instrucciones. Si se utiliza una variable sintáctica llamada instrucción y una variable llamada expresión para denotar la siguiente producción:

Instrucción → if (expresión) instrucción else instrucción

Lo anterior especifica la estructura de una instrucción condicional.
En otras producciones se define lo que es expresión y lo que es instrucción.

Lenguaje regular: Según Kelley[43] es una notación para especificar lenguajes, constituyen el menor conjunto de lenguajes sobre un alfabeto. Las expresiones regulares permiten representar de manera simplificada un lenguaje regular.

Metodologías ágiles: Las metodologías ágiles son métodos usados en la ingeniería de software que se basan en el desarrollo iterativo e incremental, que se basan en la adaptabilidad de cualquier cambio como medio para aumentar las posibilidades de éxito de un proyecto. En esta metodología los individuos y sus interacciones son más importantes que los procesos y las herramientas, se enfatiza en las comunicaciones cara a cara en vez de la documentación y los contratos. Asimismo, se da más importancia al software que funciona por encima de la documentación exhaustiva, y la respuesta a los cambios repentinos debe ser rápida y correcta, en vez de seguirse un plan cerrado.

1.7 Diseño metodológico

1.7.1 Hipótesis

Los usuarios necesitan un software que les permita plasmar problemas de programación lineal de forma sencilla.

1.7.2 Tipo de investigación

La investigación realizada para el proyecto actual es de tipo cuantitativo, ya que es posible definirlo, limitarlo y es determinístico.

[43] Dean Kelley. Teoría de autómatas y lenguajes formales. Prentice Hall, 1995, p. 48.

1.8 Metodología

1.8.1 Etapa preliminar

- **Investigación sobre desarrollo de DSLs internos:** Se investigan las diferentes metodologías, patrones, recomendaciones y requisitos para desarrollar DSLs internos.

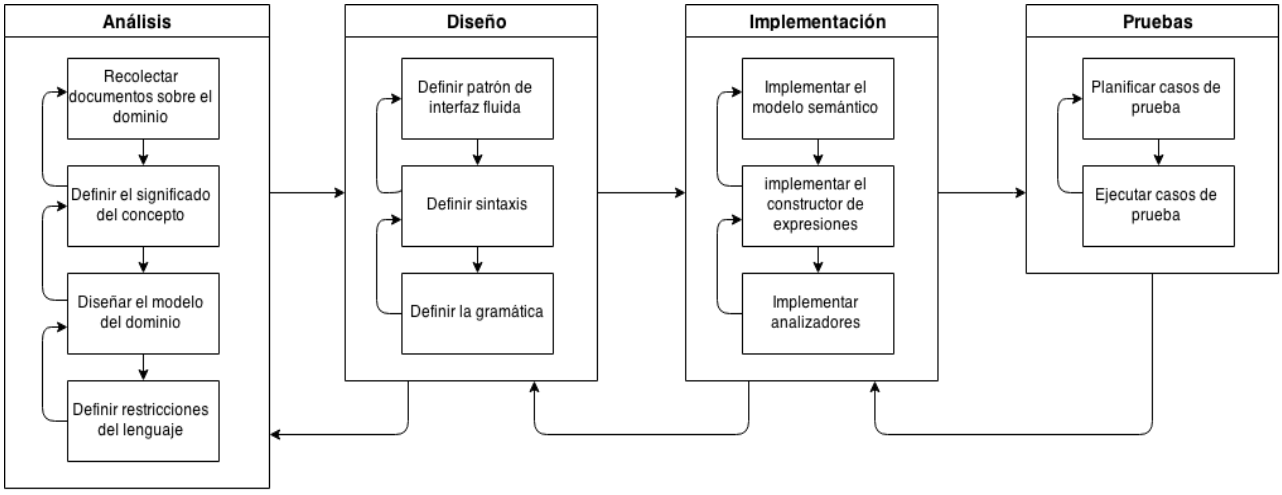
- **Investigación sobre lenguajes anfitriones:** Se investigan las características que hacen que un lenguaje de programación sea idóneo para ser un lenguaje anfitrión de un DSL interno.

- **Investigación sobre Ruby:** Se investiga y se aprende a desarrollar en el lenguaje de programación Ruby.

1.8.2 Etapa de desarrollo

Para desarrollar el DSL se creó una metodología basada en las mencionadas en el marco teórico. Fue necesario crearla ya que las existentes no son completas y dejan muchos aspectos importantes sin definir. La misma fue pensada en usarse bajo la metodología de desarrollo ágil llamada programación extrema, se realiza bajo un proceso iterativo en el cual cada una de sus fases se explica en el capítulo correspondiente. La estructura de esta metodología se plasma en la Figura 13.

Figura 13. Metodología de desarrollo de DSL



Fuente: Los autores.

1.8.1 Metodología de desarrollo

La metodología de desarrollo elegida para este proyecto fue la programación extrema.

La programación extrema (XP[44]) es una metodología ágil usada en la ingeniería de software para el desarrollo de aplicaciones. Fue formulada en el año 1999 por el ingeniero de software estadounidense Kent Beck en su libro Extreme Programming Explained: Embrace Change. Asimismo, esta metodología le da prioridad a las personas, y no a los procesos. Los cambios de requisitos en medio del desarrollo de un proyecto son un aspecto natural, inevitable e incluso deseable. Por eso es crucial poder adaptarse a esos cambios de la forma más rápida y efectiva posible y la metodología XP pone un empeño importante en este aspecto, enfatizando en la adaptabilidad en vez de la previsibilidad. Esta metodología reconoce la importancia de las decisiones de diseño, pero se resiste fuertemente al diseño inicial. En lugar de ello, pone un esfuerzo admirable en la comunicación y mejora de la capacidad del proyecto para cambiar de rumbo rápidamente. Con esa capacidad de reacción, los desarrolladores pueden utilizar la "cosa más simple que podría funcionar" en cualquier etapa de un proyecto y luego refactorizar, haciendo muchas mejoras pequeñas al diseño, llegando así en última instancia a un diseño que se ajuste a las necesidades reales del cliente[45].

La XP es considerada la metodología de desarrollo ágil más destacada en el desarrollo de software, gracias a que pretende adoptar las mejores características de las demás metodologías de desarrollo ágil, y aplicarlas de manera dinámica durante el ciclo de vida del software. En consecuencia, XP cuenta con 12 principios que guían el desarrollo y se agrupan en 4 categorías:

Retroalimentación a escala fina

1. **El principio de pruebas**: Este principio define un periodo de pruebas de aceptación del programa. Acá se definen las entradas del sistema y los resultados esperados. Se recomienda automatizar estas pruebas para agilizar las pruebas, mediante las simulaciones en ambientes de prueba. Un ejemplo de esto es JUnit para Java.

2. **Proceso de planificación**: Este principio se basa en que el usuario escribe sus necesidades, definiendo lo que debe realizar el sistema. Para ello, se crea un documento llamado “Historias de Usuario”. Cada historia de usuario

[44] Del inglés eXtreme Programming

[45] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003, p. preface.

se corresponde con un requisito del sistema y usualmente se pueden crear entre 20 y 80 historias de usuario, dependiendo de la complejidad del sistema. Con las historias de usuario se define el llamado Plan de Liberación, el cual define los tiempos de entrega de la aplicación. Estos tiempos generalmente son de una a tres semanas por historia de usuario. Este proceso también debe contar con reuniones periódicas para identificar problemas y proponer soluciones, identificando los de mayor trascendencia.

3. **El cliente en el lugar**: Este principio permite determinar los requerimientos, definir la funcionalidad, indicar las prioridades y responder las dudas de los programadores. Debido a esta fuerte interacción cara a cara entre cliente y programadores se disminuye tanto el tiempo de comunicación como la cantidad de documentación y así agilizar el proceso de desarrollo.

4. **Programación en parejas**: Este es uno de los principios más fuertes y en el que la mayoría de gerentes de desarrollo pone sus dudas. Consiste en que el grupo de desarrollo se organiza en parejas, que escriben el código compartiendo un solo computador. El intercambio de ideas entre dos programadores permite mejorar las aplicaciones, de manera consistente con el mismo o incluso menor costo.

Proceso continuo en lugar de por lotes

5. **Integración continua**: Este principio permite implementar las nuevas características del software en conjunto de manera continua, es decir, los programadores pueden reunir su código y reconstruir el sistemas varias veces al día, lo que permite un rápido progreso ya que se reducen los problemas de integración comunes ocasionados en proyectos largos que manejan la metodología en cascada.

6. **Refactorización**: Los programadores pueden construir en primera instancia una funcionalidad y seguido a eso ir mejorando el diseño del sistema a lo largo de todo el proceso de desarrollo. Se evalúa continuamente el diseño y se van recodificando lo que sea necesario para mejorar el sistema sin cambiar su funcionalidad. La finalidad es mantener un sistema que realice lo que el cliente necesita con el mínimo código posible, sin renunciar a la calidad del software.

7. **Entregas pequeñas**: Se va desarrollando funcionalidades sencillas que rápidamente se van actualizando permitiendo que el verdadero valor de negocio del producto sea evaluado en un ambiente real. Cada entrega debe ser máximo de dos o tres semanas.

Entendimiento compartido

8. **Diseño simple**: Este principio pretende enfatizar en la sencillez, apostando a elegir el programa más sencillo que cumpla con todos los requerimientos del cliente, ni más ni menos. Esto permite eliminar redundancias y rejuveneces los diseños obsoletos de forma sencilla.

9. **Metáfora**: Se desarrolla al inicio del proyecto y consiste en definir una historia de cómo es el funcionamiento del sistema completo. Esta historia es básicamente una descripción breve de cómo trabaja el sistema, en vez de usar los tradicionales diagramas y modelos UML.

   Existen también las tarjetas CRC (Clase, Responsabilidad y Colaboración) que también ayudan al equipo a definir actividades durante el diseño del sistema. Cada tarjeta representa una clase y define sus responsabilidades y las colaboraciones entre otras clases, en términos de programación orientada a objetos.

10. **Propiedad colectiva del código**: Este principio defiende la idea de que el código no es de nadie en particular, sino que es de todos los que participaron en la codificación. Esto contrasta con lo tradicional, en el cual a cada programador se le asigna un conjunto de código. El argumento para este principio es que mientras más gente participe en una porción de código, menos errores se producirán.

11. **Estándar de codificación**: Así como en principios anteriores se defiende la idea de que todos pueden trabajar en el mismo código, este define las reglas para escribirlo y documentarlo y la comunicación entre diferentes piezas de código desarrollado por diferentes equipos. De esta forma es más fácil entender el código en su totalidad y cualquiera puede hacer modificaciones. En consecuencia, al final el código parece que hubiera sido escrito por una sola persona.

Bienestar del programador

12. **La semana de 40 horas**: XP sostiene que un programador cansado escribe código de menor calidad, por lo que minimizar las horas extras y mantener a los programadores frescos, generará un código de mayor calidad.

2. LENGUAJE ANFITRIÓN

2.1 Introducción

Como plantea Fowler[46], cuando se implementa un DSL interno la sintaxis que se puede crear está limitada a la expresividad que posee el lenguaje anfitrión, por lo tanto cualquier expresión que se utilice debe ser válida para este lenguaje. Es por ello que se recomienda elegir como lenguaje anfitrión a un lenguaje de programación dinámicamente tipado, con soporte para tipado pato[47] y metaprogramación.

A continuación se exponen dichas características y la forma en la que ayudan tanto al desarrollador a implementar el DSL, cómo al usuario del DSL a utilizarlo más fácilmente, gracias a que permiten crear una sintaxis más clara.

2.1.1 Tipado dinámico

Según Ghosh[48], un programa escrito en un lenguaje dinámicamente tipado no contiene anotaciones de tipo (char, int); por naturaleza es visualmente menos ruidoso y expresa la intención del programador, esto conduce a una mejor legibilidad del código, que es uno de los atributos principales que diferencia cualquier API típica de un DSL, ya que como indica Ghosh[49] en un DSL se espera que el lenguaje fluya sin problemas, sin ninguna complejidad innecesaria. Un DSL implementado en un lenguaje de programación estáticamente tipado como Java, tiene alta probabilidad de conectar muchas anotaciones de tipo innecesarios en sus abstracciones, mientras que en un lenguaje dinámicamente tipado no es necesario proporcionar anotaciones de tipo, por lo que la intención del programador es más clara de lo que es en una implementación alternativa en un lenguaje estáticamente tipado, es por ello que en general, los lenguajes dinámicamente tipados ofrecen una sintaxis más concisa, lo que se traduce en una mayor legibilidad del DSL y su aplicación.

---

[46] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional. 2010, p. 67.

[47] Por su nombre en inglés Tipado pato.

[48] Debasish Ghosh. DSLs in action. Manning Publications Co, 2011. p. 130.

[49] Ibid., p. 130.

2.1.2 Tipado pato

“Tipado pato se refiere a la tendencia de algunos lenguajes a centrarse menos en la clase de un objeto, y dar prioridad a su comportamiento: qué métodos se pueden usar y qué operaciones se pueden hacer con él”[50]. Es llamado "Tipado pato" porque está basado en el Test del Pato (Duck Test): “Si camina como un pato, nada como un pato y hace quack, podemos tratarlo como un pato.” James Whitcomb Riley.

Es decir que siempre hay que tratar a los objetos por su funcionalidad, en vez de hacerlo por las clases de las que proceden o los módulos que incluyen. Esta es la razón por la que el tipado pato es lo que permite a lenguajes con tipado dinámico, omitir las anotaciones de tipo.

A continuación se puede apreciar la diferencia entre un lenguaje con tipado dinámico y un lenguaje con tipado estático, o lo que es lo mismo un lenguaje con tipado pato y otro sin tipado pato.

Figura 14. C# Lenguaje sin tipado pato

```
class Pato
{
  public void quack() { ... }
  public void nadar() { ... }
}
class Cisne
{
  public void nadar() { ... }
}
...
void patoNadando(Pato pato)
{
  pato.nadar();
}
...
patoNadando (new Pato()); // Legal
patoNadando (new Cisne()); // Ilegal
```

Fuente: Los autores.

[50] Ruby Tutorial. Tipado pato. [en línea]. 2015. Disponible en internet. <http://rubytutorial.wikidot.com/duck>

En la Figura 14 se aprecia una implementación en el lenguaje C# (el cual posee tipado dinámico) de un falso intento por implementar tipado pato al intentar que objetos tanto de la clase Pato como de la clase Cisne puedan acceder al método patoNadando (tratar al Cisne como si fuera un pato), sin embargo, pese a que la clase Pato y Cisne poseen el método nadar, el parámetro del método patoNadando solo permite objetos de tipo Pato, es por ello que el segundo caso de prueba es ilegal pues intenta utilizar como parámetro del método patoNadando un objeto de tipo Cisne.

Figura 15. Implementación de tipado pato en C#

```
void patoNadando(anatido pato)
{
  pato.nadar();
}
...
patoNadando(new Pato()); // Legal
patoNadando(new Cisne()); // Legal
```

Fuente: Los autores.

Por otra parte, en la implementación de la Figura 15 los objetos de tipo Pato al igual que los objetos de tipo Cisne son parámetros legales para el método patoNadando, sin embargo, para poder hacer esto se tuvo que implementar la interfaz anatido[51]. La esencia de los lenguajes con tipado pato es evitar usar interfaces y deducir si con el objeto enviado como parámetro del método se puede ejecutar la petición.

A continuación se aprecia la implementación del ejemplo de tipado pato visto anteriormente, pero en el lenguaje Ruby, el cual posee tipado dinámico.

[51] Los Anátidos integran el grupo de aves donde se estudian los patos, los gansos, los cisnes, las yaguasas y otras aves semejantes.

Figura 16. Implementación de tipado pato en Ruby

```
class Pato
  def quack
    ...
  end

  def nadar
    ...
  end
end
class Cisne
  def nadar
    ...
  end
end
...
def patoNadando (pato)
  pato.nadar
end
...
patoNadando (Pato.new)
patoNadando (Cisne.new)
```

Fuente: Los autores.

Cómo se puede observar en la Figura 16 los lenguajes que poseen tipado pato, no es necesario implementar interfaces, ya que el compilador o intérprete la deduce. Dicho en otras palabras, no hace chequeo de tipos en tiempo de compilación, sino que revisa si el tipo del objeto enviado tiene el atributo que se necesita en el método que lo llamó y si no lo tiene, se produce un error en tiempo de ejecución.

Por lo tanto es necesario entender los conceptos de tiempo de compilación y tiempo de ejecución. El tiempo de compilación es la etapa en la cual se traduce el código escrito en el lenguaje de programación a código fuente, validando las restricciones léxicas, sintácticas y semánticas. El tiempo de compilación finaliza creando un archivo ejecutable[52], de esta forma durante el tiempo de ejecución la computadora ejecuta el archivo ejecutable.

El tipado pato es posible en lenguajes interpretados ya que no presentan tiempo de compilación[53], por ende, las instrucciones no pasan por un analizador de tipos y es en tiempo de ejecución que se verifica si se cumple la regla del tipado pato.

[52] El tiempo de compilación se presenta solamente en lenguajes compilados.

[53] Porque al ser interpretados utilizan un intérprete y no un compilador.

En consecuencia mediante esta propiedad no es necesario declarar interfaces de forma estática o tener jerarquías de herencia para implementar polimorfismo, es por ello que Ghosh[54] afirma que el tipado pato ayuda a escribir código que está libre de restricciones, un efecto inmediato de esto es que la implementación del DSL se torna más concisa, pero al mismo tiempo sus intenciones son claras.

2.1.3 Metaprogramación

Tal como mencionan Flanagan y Matsumoto[55], metaprogramación es escribir programas (o frameworks) que ayudan a escribir otros programas, de ese modo, la metaprogramación es un conjunto de técnicas para extender la sintaxis de un lenguaje de programación con el objetivo de facilitar la programación.

La metaprogramación permite a un programa hacer reflexión del código, esto significa que permite a un programa analizar su código en tiempo de ejecución, es decir, podemos saber en tiempo de ejecución que clase y método estamos utilizando, también, cual es el conjunto de métodos de dicha clase. Además si se invoca un método que no existe, éste se puede crear en tiempo de ejecución y realizar las acciones que se requieran, de igual forma se puede cambiar el comportamiento de un método existente, en tiempo de ejecución. En consecuencia, cómo expresa Perrotta[56] al usar metaprogramación usted está limitado sólo por usted mismo.

Por esta razón “la metaprogramación está estrechamente ligada a la idea de escribir DSLs”[57], de esta manera los lenguajes suelen utilizar invocaciones de métodos como si fueran palabras clave, extendiendo así el lenguaje para resolver una tarea específica.

2.2 Determinación de lenguaje anfitrión

Así pues, Ghosh[58] afirma que los lenguajes de programación que poseen las características anteriormente descritas y por tanto son idóneos lenguajes anfitriones de DSLs son Ruby, Groovy y Clojure.

---

[54] Debasish Ghosh. DSLs in action. Manning Publications Co, 2011. p. 133.

[55] David Flanagan y Yukihiro Matsumoto. The Ruby Programming Language. O’Reilly, 2008. p. 266.

[56] Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010. p. 14.

[57] Flanagan and Matsumoto, Op.cit., p. 266.

[58] Ghosh, Op. Cit. p. 134.

Durante la investigación realizada se determinó que el lenguaje idóneo para implementar el DSL es Ruby, pues es el más usado de los tres, “es sin duda el lenguaje en el que se puede usar metaprogramación de la manera más amigable”[59], cuenta con vasta bibliografía, además “una buena parte del reciente ímpetu detrás de los DSL internos proviene de la comunidad Ruby, cuyo lenguaje tiene muchas características que fomentan la implementación de DSLs”[60].

De esta manera, a continuación se expondrán las características de Ruby que lo hacen un lenguaje ideal para implementar DSLs. Así mismo, como mencionó su creador Yukihiro “Matz” Matsumoto, “Es un lenguaje ideal para ahondar en la productividad y también en la diversión de la comunidad de desarrollo”.

2.3 Ruby

Tal como manifiesta Ghosh[61] Ruby es un lenguaje orientado a objetos y sigue la noción habitual de cualquier otro lenguaje orientado a objetos para definir una clase, Ruby tiene su propio modelo de objetos el cual tiene artefactos como clases, objetos, métodos de instancia, métodos de clase y métodos singleton; se puede pasar un hash como un argumento de un método, posee soporte para expresiones regulares, lo cual es muy útil en la coincidencia de patrones y procesamiento de texto y se pueden implementar bloques, los cuales se utilizan para implementar lambdas y cierres en Ruby.

Sin embargo, quizás la característica más apreciada de Ruby es la metaprogramación, con la cual se puede manipular el lenguaje para satisfacer las necesidades, en lugar de adaptarse al lenguaje cómo es, es por ello que “la metaprogramación y los DSLs tienen una estrecha relación en el mundo Ruby. Para construir un DSL interno, se debe alterar el lenguaje mismo, y hacerlo requiere muchas de las técnicas de Ruby. Dicho de otra manera, metaprogramación ofrece los ladrillos necesarios para construir DSLs”[62]. Según Flanagan y Matsumoto[63] Ruby es un lenguaje altamente dinámico; puede insertar nuevos métodos en las clases en tiempo de ejecución (“incluso una clase principal como Array”[64]), crear alias para los métodos existentes, e incluso definir métodos en objetos individuales (métodos

[59] Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010. p. 20.

[60] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 67.

[61] Ghosh. Op.cit., p. 91.

[62] Perrotta. Op. cit., p. 255.

[63] Flanagan and Matsumoto. Ghosh. Op. cit., p. 134.

[64] Perrotta, Op. cit. p. 13.

Singleton). Además, tiene una rica API para la reflexión. Reflexión, también llamada introspección, simplemente significa que un programa puede examinar su estado y su estructura. La API de reflexión en realidad va más allá y permite a un programa alterar su estado y estructura. Un programa de Ruby puede configurar dinámicamente nombres de variables, invocar nombres de métodos e incluso definir nuevas clases y nuevos métodos; del mismo modo puede evaluar una cadena o un bloque de código dinámicamente en tiempo de ejecución.

Perrotta[65] expresa que si se quiere escribir un programa de Ruby que se conecta a un sistema externo, como un servicio web o un programa Java, con metaprogramación se puede implementar un contenedor que toma cualquier llamada al método y la envía al sistema externo. Posteriormente si alguien añade métodos al sistema externo, no se tiene que cambiar el contenedor Ruby; el contenedor soportará los nuevos métodos de inmediato.

"La API de reflexión de Ruby, junto con su naturaleza generalmente dinámica, sus estructuras de control de bloques e iteradores, y su sintaxis de paréntesis opcionales, hace que sea un lenguaje ideal para metaprogramación"[66] y por ende para desarrollar DSLs.

A continuación se muestra cómo Ruby solventa las características de lenguaje dinámicamente tipado, Tipado pato y metaprogramación vistas en los anteriores apartados.

2.3.1 Tipado dinámico y legibilidad en Ruby

Según Flanagan y Matsumoto[67] La sintaxis de Ruby es orientada en la expresividad. "La unidad básica de sintaxis en Ruby es la expresión, el intérprete de Ruby evalúa las expresiones y produce valores. Las expresiones más simples son las expresiones primarias, las cuales representan directamente los valores ejemplo de esto son los números y las cadenas"[68].

Las estructuras de control como if que en otros lenguajes son llamados declaraciones, en Ruby son expresiones. Esto indica que las estructuras de control

[65] Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010. p. 13.

[66] David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. O'Reilly, 2008. p. 266.

[67] Ibid., p. 4.

[68] Ibid., p. 34.

se tratan igual que otras expresiones más simples como una asignación de valor, por tanto, se puede escribir código como el siguiente:

```
minimum = if x < y then x else y end
```

Por otra parte los operadores de Ruby se escriben de forma similar a cómo se escriben en lenguajes de programación como C, Java y JavaScript. A continuación se muestran algunos operadores de Ruby.

Figura 17. Algunos operadores de Ruby

```
1 + 2                    # => 3: addition
1 * 2                    # => 2: multiplication
1 + 2 == 3               # => true: == tests equality
2 ** 1024                # 2 to the power 1024: Ruby has arbitrary size ints
"Ruby" + " rocks!"       # => "Ruby rocks!": string concatenation
"Ruby! " * 3             # => "Ruby! Ruby! Ruby! ": string repetition
"%d %s" % [3, "rubies"]  # => "3 Rubies": Python-style, printf formatting
max = x > y ? x : y      # The conditional operator
```

Fuente: David Flanagan y Yukihiro Matsumoto. The Ruby Programming Language. O'Reilly, 2008. p. 5.

Como se observa en la Figura 17, en Ruby se pueden realizar operaciones entre objetos de tipos diferentes, por medio del tipado dinámico Ruby reconoce en tiempo de ejecución que por ejemplo una cadena puede realizar la operación de multiplicación con un entero, produciendo la misma cadena repetida las veces que corresponde al valor del entero.

Por otra parte, la asignación de valor a una variable se realiza de forma simple y se puede cambiar en cualquier momento el tipo de valor de la variable, es decir que la variable puede cambiar de tipo en cualquier momento. En la Figura 18 se aprecia claramente lo sugerido.

Figura 18. Asignación de diferentes tipos de objetos a una variable

```
x = 1        # x es un entero con valor 1
x = "Uno"    # x es una cadena con valor "Uno"
```

Fuente: Los autores.

Los parámetros de los métodos son dinámicos pues se le pueden enviar varios objetos de diferentes tipos a un método mediante la siguiente instrucción.

Figura 19. Número de parámetros variable

```
def metodo(*parametros)
  parametros.each do |parametro|
    print parametro
  end
end
...
metodo('Hola', 3)  #Hola3
metodo( )          #
```

Fuente: Los autores.

Utilizando el asterisco (*) como primer carácter del nombre del parámetro, Ruby identifica que el número de parámetros es variable. Ruby también permite que se le asignen valores por defecto a los parámetros de la siguiente forma:

Figura 20. Valores por defecto en un método

```
def metodo(param1='Hola', param2='mundo')
  print param1 + ' ' + param2
end
...
metodo()          #Hola mundo
metodo('Chao')  #Chao mundo
```

Fuente: Los autores.

Por si fuera poco, Ruby permite definir parámetros opcionales, esto se utiliza cuando el método requiere que algunos parámetros sean obligatorios y otros opciones, para hacer esto hay que tener en cuenta que los parámetros opcionales deben ir después de los parámetros no opcionales, ya que los parámetros se interpretan de izquierda a derecha, entonces si se ponen de primero los parámetros no opcionales, no se podría saber dónde acaban los parámetros no opcionales y donde empiezan los opcionales, en la siguiente imagen se muestra cómo los parámetros se interpretan de izquierda a derecha.

Figura 21. Parámetros opcionales

```
def metodo(param1='Hola', param2=param1*2, *param3)
  print param1 + ' ' + param2
end
...
metodo()  #Hola HolaHola
```

Fuente: Los autores.

Otra virtud del dinamismo de los tipos en Ruby es que los condicionales y los ciclos no solamente esperan valores booleanos (true y false) o los 1 y 0 en C, sino que, en Ruby el valor vacío (nil) es tratado como el booleano false, y cualquier otro valor es tratado como el booleano true.

Figura 22. Condicional en Ruby

```
if x
  puts 'verdadero'
  else
    puts 'falso'
end
...
x = 2    #verdadero
x = nil  #falso
```

Fuente: Los autores.

2.3.2 Tipado pato en Ruby

En la sección anterior (2.3.1) se demostró cómo Ruby puede realizar operaciones entre diferentes tipos de datos, al igual que puede cambiar el tipo de dato de un objeto en cualquier momento, esto lo hace por medio del Tipado pato. Cabe recordar que en la sección 2.1.2 se mencionó que el Tipado pato es la tendencia de algunos lenguajes a centrarse menos en la clase de un objeto, y dar prioridad a su comportamiento, de este modo cuando en Ruby se escribe la instrucción: 'Hola'*3, el resultado es 'HolaHolaHola', ya que aunque estos dos objetos no son de la misma clase, por medio de Tipado pato el intérprete de Ruby verifica que ésta operación entre objetos de distintas clases sea válida y la realiza.

Así mismo, lo que ocurre en Ruby cuando se realiza una operación aritmética en un número es que se ejecuta el método coerce(), el cual realiza la modificación de tipo pertinente para que los cálculos funcionen como se espera.

Figura 23. Tipado pato en número

```
1.coerce(2)       #[2, 1]
1.coerce(2.0)     #[2.0, 1.0]
1.0.coerce(2)     #[2.0, 1.0]
1.0.coerce(2.0)   #[2.0, 1.0]
```

Fuente: Los autores.

En la Figura 23 se observa que por medio de Tipado pato el método coerce convierte al objeto de clase Fixnum en un objeto de clase Float dentro de un vector, siempre que en la instrucción aparezca un objeto de clase Float. Sin embargo, en Ruby casi todo se puede cambiar, de esta forma, se puede redefinir el método coerce para que tenga el comportamiento que se desee.

En efecto, mediante Tipado pato además de crear métodos que pueden ser accedidos por objetos de diferentes clases, también se pueden redefinir los métodos de las clases (incluidas las básicas) de Ruby para extender la flexibilidad del lenguaje cuando el problema lo amerite.

2.3.3 Metaprogramación en Ruby

En los lenguajes compilados como C++ los métodos y variables tienen valor en un espacio de memoria solo en tiempo de compilación, una vez finaliza el período de compilación los espacios de memoria se liberan y se pierde la información relacionada con el programa, por esa razón “no se pueden solicitar a una clase sus métodos de instancia, ya que, en el momento de hacer la solicitud, la clase se ha desvanecido”[69]. Por otra parte, en los lenguajes interpretados como Ruby la metaprogramación es posible, ya que en tiempo de ejecución “la mayoría de las construcciones del lenguaje todavía están ahí”[70], de esa forma, se pueden consultar valores y construcciones del programa en ejecución.

[69] Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010. p. 15.
[70] Ibid., p. 15.

Figura 24. Ejemplo de metaprogramación en Ruby

```ruby
class Greeting
  def initialize(text)
    @text = text
  end

  def welcome
    @text
  end
end

my_object = Greeting.new("Hello")
my_object.class                          # => Greeting
my_object.class.instance_methods(false)  # => [:welcome]
my_object.instance_variables             # => [:@text]
```

Fuente: Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010. p. 16.

En la Figura 24, se puede observar cómo se pueden preguntar al sistema cosas como el nombre de la clase del objeto llamado my_object, sus métodos y sus variables en tiempo de ejecución.

A continuación se expondrán otros métodos de Ruby para manejo de metaprogramación.

2.3.3.1 Métodos de metaprogramación para manejo de objetos, clases y módulos

"Los métodos reflexivos más utilizados son aquellos para determinar el tipo de un objeto, su clase y los métodos a los que responde"[71]. Algunos de ellos se encuentran en la Tabla 3.

Tabla 3. Métodos de metaprogramación para evaluar clases y módulos

Métodos	Comportamiento
o.class	Retorna la clase del objeto o.
c.superclass	Retorna la superclase de la clase c.
o.instance_of? c	Determina si el objeto o es instancia de la clase c.
o.is_a? c	Determina si el objeto o es instancia de la clase c, o de

[71] David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. O'Reilly, 2008. p. 266.

	cualquiera de sus subclases. si c es un módulo, este método prueba si la clase de la cual es instancia o (o cualquiera de sus ancestros), incluye al módulo.
o.kind_of? C	Es un sinónimo de is_a?.
c === o	Para cualquier clase o módulo c, determina si o.is_a?(c).
o.respond_to? name	Determina si el objeto o tiene un método público o protegido con el nombre especificado.
B < A	Retorna true si el módulo o clase B incluye al módulo A.
B.include?(A)	Retorna true si el módulo o clase B incluye al módulo A.
A.included_modules	Retorna los módulos incluidos en la clase o módulo A.
A.ancestors	Retorna los ancestros de la clase o módulo A.
o.extend(M)	Extiende al objeto o, haciendo que los métodos del módulo M sean métodos singleton del objeto o.
M.nesting	Retorna un vector que expresa el anidamiento de módulos en el cual se encuentra el módulo M.

Fuente: Los autores.

2.3.3.2 Métodos de metaprogramación para manejo de variables y constantes

Para usar metaprogramación en las variables y constantes, las clases “Kernel, Object y Module definen métodos reflexivos para listar los nombres (como cadenas) de todas las variables globales definidas, variables locales definidas actualmente, todas las variables de instancia de un objeto, todas las variables de clase de una clase o módulo y todas las constantes de una clase o módulo”[72]. Estos métodos son: global_variables, instance_variables, class_variables, constants y local_variables.

2.3.3.3 Métodos de metaprogramación para manejo de métodos

Para usar metaprogramación en métodos, la clase Object define métodos para listar los nombres de los métodos definidos en el objeto. Algunos de ellos son:

[72] David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. O’Reilly, 2008. p. 271.

Tabla 4. Métodos para listar nombres de métodos

Métodos	Comportamiento
o.methods	Lista los métodos públicos del objeto o.
o.public_methods	Sinónimo de methods.
o.public_methods(false)	Excluye métodos heredados.
o.protected_methods	Lista los métodos protegidos del objeto o.
o.private_methods	Lista los métodos privados del objeto o.
o.private_methods(false)	Excluye métodos privados heredados.
def o.single; 1; end	Define un método singleton.
o.singleton_methods	Lista los métodos singleton

Fuente: Los autores.

Además si se quiere definir un nuevo método de instancia de alguna clase o módulo, se puede usar el método define_method, para hacerlo debe hacerse algo como lo mostrado en la Figura 25.

Figura 25. Métodos define_method

```
# Add an instance method named m to class c with body b
def add_method(c, m, &b)
  c.class_eval {
    define_method(m, &b)
  }
end

add_method(String, :greet) { "Hello, " + self }

"world".greet   # => "Hello, world"
```

Fuente: David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. O'Reilly, 2008. p. 274.

De acuerdo a la Figura 25 para la creación de este nuevo método es necesario pasar como argumentos la clase de la cual será parte, el nombre que tendrá y el bloque de instrucciones que ejecutará, no obstante, si lo que se quiere es eliminar dinámicamente un método existente, se puede hacer uso de remove_method y de undef_method.

Otro método útil para manipular métodos es el alias_method o simplemente alias, mediante estos métodos es posible darle sinónimos a un método existente, funciona

como un diccionario en el cual un método puede tener diferentes nombres. Por otra parte, si se llama a un método no existente, se puede usar method_missing para "capturar y manejar invocaciones arbitrarias en un objeto"[73].

Por todas estas propiedades que permiten enriquecer la expresividad de Ruby es que Flanagan y Matsumoto[74] afirman que el objetivo de la metaprogramación en Ruby es a menudo la creación de DSLs.

[73] David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. O'Reilly, 2008. p. 284.

[74] Ibid., p. 296.

3. PROGRAMACIÓN LINEAL

3.1 Introducción

La programación lineal (PL) es una de las técnicas usadas en la investigación de operaciones para la resolución de problemas de optimización. Fue desarrollada hacia finales de la II Guerra Mundial, con la aparición de la investigación de operaciones[75]. Es importante aclarar que el término de programación en este contexto no hace referencia a la programación informática, sino más bien se refiere a la acción de planificar y organizar, por lo tanto en muchas ocasiones esta técnica es denominada optimización lineal, para evitar posibles confusiones. Los problemas de programación lineal pretenden optimizar el uso eficiente o la asignación de recursos limitados para alcanzar un objetivo deseado[76]. Por optimizar se refiere al proceso mediante el cual se busca la obtención de los mejores valores dentro de un conjunto de valores posibles dada una función objetivo, atendiendo y respetando un conjunto de limitantes o restricciones impuestas sobre esas variables. En otras palabras, un problema de optimización consiste en maximizar o minimizar una función real compuesta por un conjunto de variables o incógnitas que representan elementos de la vida real eligiendo sistemáticamente valores de entrada para esas variables, teniendo como limitante el dominio de valores que pueden tomar dichas variables. Esta aclaración aplica en su totalidad a todos los problemas de programación lineal, siempre que las restricciones y la función objetivo sean de carácter lineal.

Todos los valores dados a las variables de decisión que cumplan con las condiciones dadas en las restricciones, son llamadas soluciones factibles. A la hora de determinar la solución de un problema puede suceder que este tiene una única solución, varias posibles soluciones, o que no haya ninguna solución que cumpla las condiciones dadas. Las mejores soluciones factibles siempre son denominadas soluciones óptimas. Con esta aclaración se puede concluir que una solución factible no siempre es una solución óptima, mientras que una solución óptima siempre es una solución factible.

Un problema de programación lineal debe contar con los siguientes elementos:

- **Variables de decisión**: Son el conjunto de elementos para los que se desea hallar los valores óptimos. Se representan usualmente mediante el uso de

[75] KOLMAN, Bernard; BECK, Robert E. Elementary Linear Programming with Applications. Orlando, Florida. Elsevier, 2014. p. 2.

[76] WILEY, Jhon & Sons. Encyclopedia of Statistical Sciences. 2006. p. 1.

letras como la x y un sufijo consecutivo (x_1, x_2, …, x_i, ..., x_n), pero esta representación es abierta a la elección de cada persona.

- **Función objetivo**: Esta es la función principal ya que tiene una estrecha relación con la pregunta general que se desea responder. Calcula la cantidad que será maximizada o minimizada conforme se van asignando valores a las variables de decisión. Con esta función se determina si una solución es factible y si es así permite determinar si es óptima. Se representa de la siguiente manera:

$$min: z = \sum_{i=1}^{n} k_i x_i \quad (1)$$

$$max: z = \sum_{i=1}^{n} k_i x_i \quad (2)$$

k_i= Es el coeficiente i-ésimo que expresa el costo de la variable x_i.
x_i = Es la variable de decisión i-ésima que se desea optimizar.

- **Restricciones**: Son el conjunto de ecuaciones o inecuaciones lineales que determinan el conjunto de soluciones factibles, es decir indican el dominio de posibles valores que pueden llegar a tomar las variables para hallar una solución válida. Estas restricciones se denominan comúnmente restricciones principales y se representan de la siguiente manera:

$$\sum_{i=1}^{n} a_i x_i \leq b_i \quad (3)$$

$$\sum_{i=1}^{n} a_i x_i \geq b_i \quad (4)$$

a_i= Es el coeficiente i-ésimo técnico conocido que acompaña la variable de decisión x_i. Generalmente es denominado coeficiente tecnológico.
x_i = Es la variable de decisión i-ésima que se desea optimizar.
b_i = El valor límite de la restricción, el cual debe ser respetado para que los valores de las variables de decisión sean factibles.

Estas inecuaciones (3) y (4) pueden transformarse a ecuaciones mediante adiciones o sustracciones adecuadas de variables estacionarias x_{n+1} no negativas, por ejemplo para (3) tendríamos:

$$\sum_{i=1}^{n} a_i x_i + x_{n+1} = b_i \quad (5)$$

y para (4) tendríamos:

$$\sum_{i=1}^{n} a_i x_i \; - \; x_{n+1} \; = \; b_i \qquad (6)$$

Para cada inecuación debe existir una variable estacionaria que tiene como valor la diferencia entre el lado izquierdo y el derecho de la desigualdad[77].

Se debe tener en cuenta también que las variables de decisión generalmente no pueden tomar valores negativos, por lo que existe una restricción implícita que es la restricción de no negatividad:

$$x_i \; \geq 0 \qquad (7)$$

En el documento actual se trabaja con dos clases especiales de problemas, el problema máximo estándar y el problema mínimo estándar. En estos problemas todas las variables son restringidas a ser no negativas, y todas las restricciones principales son desigualdades. En el caso del problema máximo estándar, las restricciones principales toman la forma de la inecuación (3), mientras que en el caso del mínimo estándar, las restricciones principales toman la forma de la inecuación (4). Tal como afirma Ferguson[78] todos los problemas lineales pueden ser traducidos a una de estas dos clases estándar. Por ejemplo, si una restricción de un problema mínimo estándar tiene una desigualdad con $\leq$, se puede cambiar a $\geq$multiplicando ambos lados de la restricción por (-1) e invirtiendo el signo de la desigualdad. En el caso del máximo estándar ocurre exactamente al contrario. Así mismo un problema máximo estándar puede ser transformado a mínimo estándar multiplicando la función objetivo y todas las restricciones por (-1). Pueden surgir también otros dos casos para la transformación.

1. **Algunas restricciones pueden ser igualdades**. Una restricción de igualdad de la forma (8) puede ser eliminada resolviendo esta restricción para algún $x_j$para el cual $a_{ij} \neq 0$y sustituyendo esta solución en las otras restricciones y en la función objetivo donde aparezca x_j. Así se elimina una restricción y una variable del problema.

$$\sum_{j=1}^{n} a_{ij} x_j \; = \; b_i \qquad (8)$$

2. **Algunas variables pueden no ser restringidas a ser no negativas.** Una variable no restringida x_j, pueden ser reemplazados por la diferencia entre dos variables no negativas, $x_j = u_j - v_j$, donde ambas variables u_j y v_j

[77] WILEY, Jhon & Sons. Encyclopedia of Statistical Sciences. 2006. p. 1.
[78] FERGUSON, Thomas S. Linear Programming, a concise introduction. p. 8.

deben ser positivos. Esto agrega una variable y dos restricciones de no negatividad al problema.

A la hora de resolver un problema lineal, el método más conocido y usado es el Simplex, desarrollado por Dantzig en los años cuarenta[79]. Conocer y entender el funcionamiento de este método iterativo ayuda a la comprensión de las diferentes formas de resolución de programas lineales. Sin embargo, dada la complejidad que representa resolver un problema con muchas variables de decisión y muchas restricciones mediante Simplex, debido a su gran cantidad de iteraciones, se han ido desarrollando mejoras del método y han ido surgiendo otros métodos más sofisticados. Otro factor influyente en el avance de las técnicas ha sido el gran avance tecnológico de las últimas décadas, permitiendo la construcción de computadores cada vez más potentes, que permiten resolver problemas de mayor complejidad en menor tiempo.

En el caso en que un problema sea de únicamente dos variables de decisión, se podría usar el método gráfico para resolverlo. Este método consiste básicamente en graficar en un plano cartesiano las rectas que representan cada una de las restricciones del problema. Una vez graficadas, se procede a hallar los vértices formados en las intersecciones entre las rectas. Si la figura formada es cerrada, significa que existe una solución o conjunto de soluciones óptimas, mientras que si es abierta, podría darse el caso de que no existan soluciones factibles para el problema dado. Normalmente, la solución óptima para estos problemas se encuentra en uno de los vértices de la región de soluciones factibles.

Existen problemas que cuentan con un número de vértices de la región factible muy grande y el cálculo de la solución óptima sería bastante complejo. Para solucionar esta complejidad se creó el método simplex. Este método va escogiendo ciertos puntos y va trabajando con ellos, sin necesidad de probar todos. Este procedimiento parte de un vértice cualquier y mediante algoritmos se va trasladando a vértices adyacentes que mejor el valor de la función objetivo, obteniendo la solución óptima en un número de iteraciones bastante inferior al de evaluar cada uno de los vértices.

No se va a profundizar en la forma de resolver problemas lineales, dado que no es el objeto principal del proyecto. Por otro lado, el modelado de un problema de este tipo si es de importancia para el desarrollo y entendimiento de este documento, por lo cual se va a dar una serie de pautas a seguir para el correcto modelado de un problema lineal. Para facilitar su entendimiento, estas pautas serán explicadas mediante el siguiente ejemplo práctico:

[79] CABALLERO, José A.; GROSSMAN, Ignacio E. Una revisión del estado del arte en optimización. Revista Iberoamericana de Automática e Información Industrial. Vol 4, Núm 1, 2007. p. 6.

3.1.1 Ejemplo práctico para el modelado de un problema lineal.

El siguiente problema es tomado de la página 25 del libro TAHA en su 6ta edición.

Problema: La tienda de comestible BK vende dos tipos de bebidas: La marca sabor a cola A1 y la marca propia de la tienda, Bk de cola, más económica. El margen de utilidad en la bebida A1 es de 5 centavos de dólar por lata, mientras que la bebida de cola Bk suma una ganancia bruta de 7 centavos por lata. En promedio, la tienda no vende más de 500 latas de ambas bebidas de cola al día. Aun cuando A1 es una marca más conocida, los clientes tienden a comprar más latas de la marca Bk, porque es considerablemente más económica. Se calcula que las ventas de la marca Bk superan a las de la marca A1 en una razón 2:1 por lo menos. Sin embargo, BK vende, como mínimo, 100 latas de A1 al día. ¿Cuántas latas de cada marca debe tener en existencia la tienda diariamente para maximizar su utilidad?

Respuesta:

1. Lo primero que se debe identificar son las variables de decisión, ya que son estas las incógnitas que hacen referencia a los valores a optimizar en el problema.

   En la pregunta al final del enunciado, se pueden identificar fácilmente las variables de decisión, ya que se hace referencia a las dos marcas de bebidas de cola en lata. Por lo tanto las podemos expresar de la siguiente manera:

   x_1 = Latas de bebida A1 que debe tener la tienda en existencia diariamente.
   x_2 = Latas de bebida Bk que debe tener la tienda en existencia diariamente.

2. A continuación se debe identificar cuál es la función objetivo del problema, teniendo en cuenta las variables de decisión y los coeficientes de los costos de cada una de ellas.

   En este caso el objetivo es incrementar al máximo la utilidad por la venta de los dos tipos de bebidas. Los costos de utilidad son de 5 centavos por la lata A1 y 7 centavos por la lata Bk. A partir de estos datos podemos definir la función objetivo de la siguiente manera:

   $$max: z = 5x_1 + 7x_2$$

3. Seguido de esto, se procede a analizar detalladamente el enunciado del problema buscando las condiciones que permiten definir las restricciones. Se debe tener en cuenta que las variables de decisión siempre deben estar a la izquierda y las constantes a la derecha.

En este problema se pueden identificar 4 restricciones, que se enuncian a continuación:

- En promedio la tienda no vende más de 500 latas de ambas bebidas al día:

  Restricción 1: $x_1 + x_2 \leq 500$

- Los clientes tienden a comprar más latas de la marca Bk :

  $$x_2 \geq x_1$$

  y teniendo en cuenta el comentario de las variables de decisión, entonces queda así

  Restricción 2: $-x_1 + x_2 \geq 0$

- Las ventas de Bk superan a las ventas de A1 en una razón de 2:1 por lo menos

  $$x_2 \geq 2x_1$$

  y teniendo en cuenta el comentario de las variables de decisión, entonces queda así

  Restricción 3: $-2x_1 + x_2 \geq 0$

- Se venden como mínimo 100 latas de A1 al día:

  Restricción 4: $x_1 \geq 100$

4. Al final el modelo en su plenitud quedaría expresado de la siguiente manera:

   $\max: z = 5x_1 + 7x_2$

   Sujeto a:

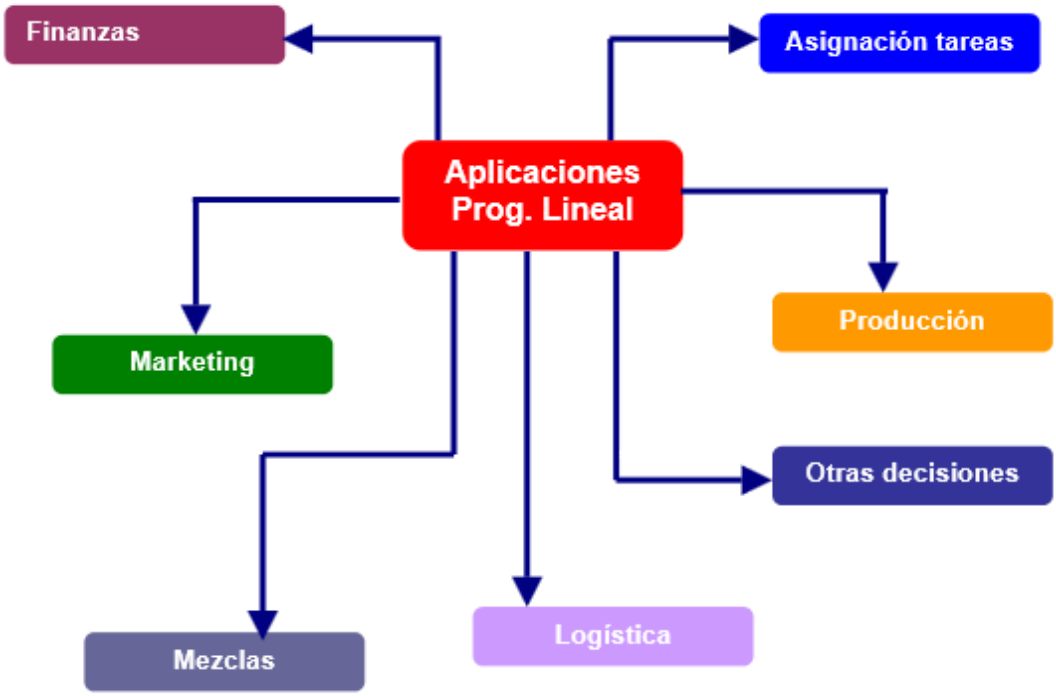
   $$x_1 + x_2 \leq 500$$
   $$-x_1 + x_2 \geq 0$$
   $$-2x_1 + x_2 \geq 0$$
   $$x_1 \geq 100$$
   $$x_1, x_2 \geq 0 \rightarrow \text{(La condición de no negatividad)}$$

De esta manera ya el problema queda modelado y listo para ser resuelto.

3.2 Aplicaciones

La PL surgió a partir de diferentes necesidades que ha manifestado la sociedad a lo largo de la historia. A partir de estas necesidades se desarrollaron los conceptos de la programación lineal que se conocen hoy en día. No se trata de simples aplicaciones matemáticas, sino de una gran cantidad de casos prácticos del mundo real que han provocado el auge real de esta técnica operacional. La programación lineal tiene muchas aplicaciones, especialmente en el ámbito de la ingeniería, las ciencias, la industria y los negocios. En términos generales, la PL se puede aplicar para cualquier actividad que involucre la toma de decisiones que cuenten con variables de decisión, una función objetivo lineal, y restricciones lineales.

Figura 26. Aplicaciones clásicas de la programación lineal en el mundo real.



Fuente. FAULIN, Javier; JUAN, Ángel A. Aplicaciones de la programación lineal.

La Figura 26 muestra algunas de las principales aplicaciones que tiene esta técnica en el mundo real. A parte de estas aplicaciones existen más, sin embargo, se tomarán éstas como ejemplo y serán explicadas en las siguientes secciones.

3.2.1 Finanzas

El manejo oportuno de las finanzas son vitales en cualquier entidad y la PL tiene una importante aplicación en este campo, ayudando a determinar la mejor forma posible de maximizar beneficios en una empresa. La selección de una cartera de valores es un ejemplo típico en el que es usada esta técnica. Acá los directivos de bancos, fondos de inversión, y compañías de seguros deben seleccionar una serie de inversiones concretas entre diferentes posibles en el mercado. Por consiguiente, el objetivo es maximizar los beneficios esperados de estas inversiones, teniendo en cuenta un conjunto de restricciones, tanto legales como las de la misma empresa.

Otro ejemplo de uso de la PL en las finanzas es de la asignación de créditos. Este problema parte de un monto inicial para repartir en diferentes créditos, y cada crédito tiene un rendimiento distinto. El objetivo es maximizar beneficios de otorgar los créditos de acuerdo al rendimiento que cada uno ofrece.

3.2.2 Producción

En el campo de la producción es muy común usar la PL para la combinación óptima de bienes. En estos casos las fábricas deben decidir sobre la cantidad más adecuada que deben producir de cada uno de sus productos para maximizar beneficios sin dejar de lado ciertos requisitos, como pueden ser los financieros, de capacidad de producción, de demanda, de calidad o de disponibilidad de materias primas.

Así mismo, la PL también se usa en esta área para la planificación de la producción de una fábrica. Este tema es muy difícil e importante en la plantas de producción de las grandes fábricas debido a la gran cantidad de tareas y productos que deben producir. Para esta planificación se deben considerar muchos factores: mano de obra, limitaciones de espacio, costos de inventario y almacenamiento, demanda, entre otros. Este problema es similar al de la combinación óptima de bienes, ya que el objetivo de ambos es el de maximizar beneficios o minimizar costos de producción y de almacenamiento.

3.2.3 Marketing

La PL es usada en el campo del Marketing y la publicidad para determinar cuál es la mejor combinación de medios usados para anunciar algún producto o servicio.

En la mayoría de los casos se arranca con un presupuesto fijo para publicidad y el objetivo consiste en distribuirlo entre los diferentes medios de comunicación (televisión, radio, revistas, periódicos, internet, redes sociales, vallas publicitarias, etc.) de forma que se alcance la mayor difusión posible entre los clientes. En algunas ocasiones las restricciones serán dadas por la disponibilidad de cada medio y por las políticas publicitarias de la propia empresa.

Otra aplicación importante en este campo se da en los estudios de mercado, por ejemplo en el diseño de encuestas. Suponiendo que se debe realizar una encuesta para determinar la opinión sobre algún tema, pero ésta debe ser lo más neutral posible y se debe seleccionar una población objetivo que sería el dominio del problema. Esta población no debe ser muy extensa, pero sí debe tratar de representar con el mayor porcentaje posible la opinión de la población total. Entonces se deben establecer unos requisitos que vendrían siendo las limitaciones de la población factible a la que se le hará la encuesta. Pero resulta que estas encuestas implican también un costo, ya que deben realizarse en persona y eso implica gastos en transporte por ejemplo. Así que para este problema se debe tratar de minimizar costos pero a su vez seleccionar la mejor población objetivo posible.

3.2.4 Logística

Un caso de aplicación típico de logística es el problema del transporte. Este problema se refiere al proceso de determinar el número de bienes o mercancías que se deben transportar desde diferentes lugares de origen hacia otros lugares de destino posibles. El objetivo en este caso es el de minimizar los costes de transporte. Las restricciones vienen dadas por la capacidad productiva de cada origen y las necesidades de cada destino. Para tal propósito existen algoritmos especiales que facilitan su resolución, por ejemplo el método de Vogel, el método de Paso Secuencial o el método de distribución modificada MODI.

3.2.5 Asignación de tareas

La asignación de tareas es vital en muchas empresas para optimizar los recursos de los que se dispone. En este campo de aplicación se tiene como objetivo asignar de la forma más eficiente posible un trabajo a cada empleado o máquina de la empresa. El objetivo es el de minimizar tiempos o costos, o bien el de maximizar la efectividad en las asignaciones. Para esto, existen técnicas adicionales al conocido método simplex que fueron diseñadas especialmente para este tipo de problemas de la PL y permiten mejorar el tiempo de convergencia, minimizando el número de iteraciones para dar con la solución óptima. Un ejemplo es el método húngaro.

Una característica particular de los problemas de asignación es que tanto los coeficientes tecnológicos como los términos independientes siempre toman el valor 1. Además todas las variables de decisión son binarias, tomando el valor de 1 si se lleva a cabo la asignación propuesta y 0 en caso contrario.

De igual forma, la planificación de horarios es otro ejemplo práctico de la PL en este campo. Se corresponde con la intención de solucionar efectivamente las necesidades de personal durante un periodo de tiempo concreto. En este campo la PL se vuelve muy útil especialmente cuando se dispone de cierta flexibilidad a la hora de asignar tareas a empleados polifuncionales. Un ejemplo típico de esto se puede ver en la planificación de horarios de las entidades bancarias.

3.2.6 Mezclas

Un ejemplo importante en el área de las mezclas es el problema de la dieta. Este problema fue una de las primeras aplicaciones que se le dio a la PL y comenzó a usarse en hospitales para determinar la dieta más económica para alimentar a los pacientes a partir de unas especificaciones nutritivas mínimas. Tal como expresa Faulin[80], actualmente se aplica en el ámbito agrícola buscando de igual manera la forma más óptima de combinar alimentos que supongan el mínimo coste, pero con la mayor cantidad de nutrientes posible.

3.3 Alternativas para el modelado de un problema de programación lineal

En la elaboración de soluciones para modelos de programación lineal, al igual que en todos los problemas de optimización en general, varias son las alternativas que existen, cada uno con sus propias características. Sin embargo, a la hora de elegir con cuál alternativa trabajar, es importante tener en cuenta la necesidad y la complejidad de los modelos a resolver. Estas son las diferentes alternativas:

3.3.1 Lenguajes de Modelado Algebraico (AML)

Los lenguajes de modelado algebraico son DSLs orientados a describir y resolver problemas matemáticos de alta complejidad y a gran escala, es decir, problemas con una gran cantidad de datos y variables. La principal ventaja de estos es su similitud con la sintaxis de la notación matemática para problemas de optimización,

[80] FAULIN, Javier; JUAN, Ángel A. Aplicaciones de la programación Lineal. p. 16.

facilitando la definición clara y legible de problemas en dicho dominio, permitiendo así mejorar la productividad.

3.3.2 Lenguaje de Propósito General (GPL)

Esta alternativa generalmente implica el desarrollo de librerías o paquetes especializados para este tipo de problemas. Es una buena alternativa cuando se requiere integrar el modelo a una aplicación con interfaces específicas para la entrada y salida de datos. Requieren de un tiempo de desarrollo prolongado y desafortunadamente es difícil hacerles mantenimiento o modificar el modelo. Se benefician de las facilidades que proporcionan los entornos de desarrollo (IDE) para estos lenguajes, pero carecen de recursos de depuración orientados a un problema de optimización con restricciones.

3.3.3 Hoja de cálculo con Solver Asociado

Se trata de la alternativa más simple de todas y la más asequible para especificar y resolver modelos pequeños de optimización lineal. La entrada y salida de datos se beneficia de todas las facilidades que presenta el entorno de la hoja, especialmente la posibilidad de representar gráficamente los resultados. El principal inconveniente proviene de la dispersión en que queda la especificación del modelo, al tener que expresarse a través de fórmulas asociadas a las diferentes celdas. Si el modelo tiene cierto tamaño, la implantación y mantenimiento se hace muy tediosa y compleja. Esta alternativa resulta válida a la hora de ensayar pequeños prototipos de modelos.

3.3.4 Entorno de cálculo numérico y/o simbólico

Existen muchos entornos de cálculo matemático como Matlab, Maple o Mathematica. Estos disponen de solver asociados que pueden ser usados desde el propio medio de programación del entorno. Un gran beneficio que poseen, al igual que las hojas de cálculo con solver asociados, es que existen muchas herramientas de visualización gráfica que facilitan la presentación de los resultados de diferentes maneras. El principal inconveniente es la ausencia de recursos específicos para la expresión de los modelos y la depuración de su funcionamiento.

4. ANÁLISIS

4.1 Introducción

Según Mernik, Heering y Sloane[81] en la fase de análisis del DSL se recopila el conocimiento del dominio y se identifica el dominio del problema, debido a que "el DSL simplemente actúa como un mecanismo para expresar cómo está configurado el modelo. Por ende el DSL es sólo una fachada delgada sobre el modelo"[82]. Para hacer esto se requiere la colaboración de expertos de dominio y/o la disponibilidad de documentos de los cuales se pueda obtener el conocimiento del dominio.

Para cumplir dicho propósito se usaron los lineamientos de *Diseño Guiado por el Dominio* (*DDD*[83]) postulados por Eric Evans[84] para definir apropiadamente el alcance del dominio[85] del lenguaje a implementar. Una vez se determina el dominio se procede a definir formalmente los requerimientos que satisfacen la necesidad detectada.

4.2 Diseño Guiado por el Dominio (DDD)

"El diseño guiado por el dominio es un enfoque de diseño de software que enlaza el modelado del dominio y el diseño del software, con el objetivo de crear un modelo del dominio que evolucione a través de sucesivas iteraciones del diseño"[86].

Es necesario por tanto, definir qué es un modelo. Como menciona Evans[87] un modelo es una simplificación, es una interpretación de la realidad que abstrae los aspectos relevantes para la solución del problema en cuestión e ignora detalles superfluos, por tanto, esa área temática a la que el usuario aplica el programa es el dominio del software. Para crear software que está valiosamente involucrado en actividades de los usuarios, un equipo de desarrollo debe poseer altos

---

[81] Marjan Mernik, Jan Heering and Anthony M. Sloane "When and How to Develop Domain-Specific Languages" ACM 2005, p. 8.

[82] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 17.

[83] Por su nombre en inglés Doman-Driven Design

[84] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003

[85] Todo lo que se podría hacer con el lenguaje a implementar.

[86] BLOGSPOT. Domain-Driven Design. [en línea]. 2015. <http://tratandodeentenderlo.blogspot.com/2013/08/domain-driven-design.html>

[87] Evans. Op. cit., p. 24.

conocimientos relacionados con esas actividades, los modelos son herramientas que sirven para recopilar toda esa información, así pues, un modelo es una forma simplificada, selectiva y conscientemente estructurada de conocimientos, también se puede decir que un modelo adecuado le da sentido a la información y la enfoca en un problema.

Sin embargo, "un modelo de dominio no es un diagrama en particular; es la idea que el diagrama pretende transmitir"[88], por tanto, un modelo de dominio no es sólo el conocimiento que posee un experto de ese dominio; es una abstracción rigurosamente organizada y selectiva de ese conocimiento, por ende, un diagrama puede representar y comunicar un modelo, así como también puede hacerlo un algoritmo.

Así pues, el diseño guiado por el dominio tiene las siguientes premisas[89]:

1. Para la mayoría de proyectos de software, el enfoque principal debe estar en la lógica de dominio y en el dominio, ya que poseer un modelo, el cual exprese fielmente los requerimientos del cliente, hace que todo el equipo de desarrollo entienda fácilmente el problema y por ende, detecte rápidamente la solución; mejorando así la productividad.

2. Diseños de dominio complejo deben basarse en un modelo, es decir que para modelar software que posea un gran rango de acción o dominio grande, es fundamental usar un modelo como guía de todo el proceso de desarrollo, pues de esta forma se segmenta adecuadamente el dominio.

Para llevar a cabo los postulados del DDD se debe usar alguna metodología de desarrollo ágil (Según Evans[90] la más usada es programación extrema) ya que su desarrollo es iterativo y los desarrolladores (los que saben cómo construir software) y expertos del dominio (los que conocen el dominio) deben trabajar en equipo.

Para que la comunicación entre desarrolladores y expertos del dominio sea exitosa se debe determinar un lenguaje ubicuo, el cual es un lenguaje común entre los desarrolladores y expertos de dominio para referirse al modelo[91]. Es necesario definir este lenguaje ubicuo pues "los sobrecostos por traducir todo, más el riesgo

---

[88] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003, p. 25.

[89] Ibid., p. 25.

[90] Ibid., p. 15.

[91] MARTIN FOWLER. Ubiquitous Language [Lenguaje ubicuo]. [en línea]. 2015. <http://martinfowler.com/bliki/UbiquitousLanguage.html>

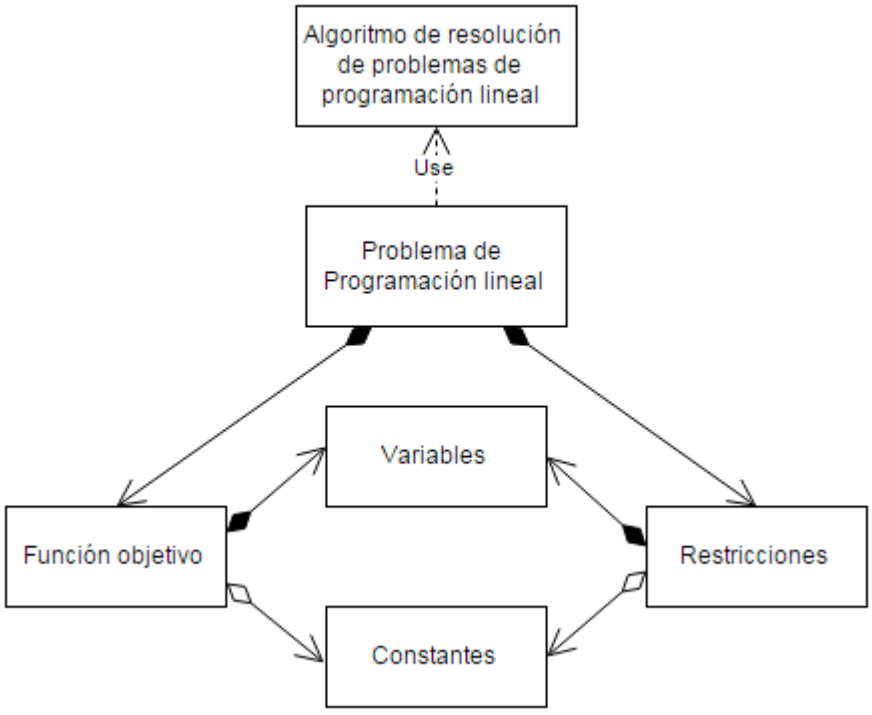
de malentendidos, es demasiado alto"[92], además, con un lenguaje ubicuo el modelo no es sólo un artefacto de diseño, sino que se convierte en parte integral de todo lo que los desarrolladores y expertos en el dominio hacen juntos.

4.3 Análisis del dominio

En el análisis que propone el DDD se debe definir y plasmar por medio de diagramas el modelo a implementar.

El modelo obtenido después del análisis del dominio de problemas de programación lineal hecho en el capítulo 3, es el siguiente:

Figura 27. Modelo de programación lineal



Fuente: Los autores.

[92] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003. p. 42.

Una vez determinado el modelo, se definen los patrones de elementos del dominio. A continuación se plantean estos patrones[93] y los elementos del modelo de la Figura 27 que corresponden a cada patrón.

- **Entidad**: Objetos que tienen una identidad distintiva que se ejecuta a través del tiempo y las diferentes representaciones. Para el dominio plasmado en la Figura 27, las constantes son la entidad pues una vez definidas, no pueden cambiar su valor.

- **Objeto de valor**: Objetos que no tienen identidad, describen alguna característica de algo. Para el dominio plasmado en la Figura 27, los elementos del dominio que cambian su valor y de esa forma describen las características que tendrá el problema de programación lineal, son las variables, la función objetivo y las restricciones.

- **Servicio**: Una operación independiente en el contexto de su dominio. Para el dominio plasmado en la Figura 27, el servicio es el algoritmo solucionador de problemas de programación lineal elegido para resolver el problema de programación lineal, pues es una operación que no hace parte del problema, sino que es utilizada una vez se haya definido por completo el problema.

4.4 Requisitos generales

4.4.1 Funciones del sistema

Las funciones del DSL son:

- **Almacenar datos**: Los datos proporcionados por el usuario deben ser almacenados en una estructura de datos. Estos datos son:
    - Variables del problema
    - Función objetivo a optimizar
    - Restricciones del problema

4.4.1 Características de los usuarios

- **Administrador:** Profesional, técnico o estudiante con conocimientos en desarrollo de software en el lenguaje de programación Ruby.

[93] MARTINFOWLER. Evans Classification. [en línea]. 2015. <http://martinfowler.com/bliki/EvansClassification.html>

- **Usuarios:** Profesionales, técnicos y estudiantes que hacen uso de la programación lineal.

4.5 Requisitos específicos

4.5.1 Características del sistema

Las restricciones que surgieron una vez se definió el modelo son las siguientes.

- **Ingresar datos**: Esta característica permite obtener los datos necesarios para plantear un problema de programación lineal.

Tabla 5. Restricción 1

1. Ingresar variables
Descripción:
El sistema debe permitir al usuario ingresar los nombres de las variables a usar.
Criterios de aceptación:
● El sistema debe permitir al usuario ingresar el número de variables que desee.

Fuente: Los autores.

Tabla 6. Restricción 2

2. Ingresar función objetivo
Descripción:
El sistema debe permitir al usuario ingresar la función que representa la función objetivo.
Criterios de aceptación:
● El sistema debe permitir al usuario ingresar la función objetivo.

Fuente: Los autores.

Tabla 7. Restricción 3

3. Ingresar restricciones
Descripción:
El sistema debe permitir al usuario ingresar las funciones que representan las restricciones del problema.
Criterios de aceptación:
● El sistema debe permitir al usuario ingresar el número de restricciones que desee.

Fuente: Los autores.

- **Verificar sintaxis**: Esta característica permite al sistema analizar la sintaxis digitada por el usuario y comprobar que sea aceptada por el DSL.

Tabla 8. Restricción 4

4 Comprobar campos obligatorios
Descripción:
El sistema debe validar si el usuario digitó los campos obligatorios para plantear un problema de programación lineal.
Criterios de aceptación:
● El sistema debe validar que se escribieron las variables con la sintaxis que el DSL reconoce. ● El sistema debe validar que se escribió la función objetivo con la sintaxis que el DSL reconoce. ● El sistema debe validar que se escribieron las restricciones con la sintaxis que el DSL reconoce.

Fuente: Los autores.

Tabla 9. Restricción 5

5 Comprobar sintaxis en variables
Descripción:
El sistema debe validar si el usuario digitó las variables, en el estándar válido.
Criterios de aceptación:

● El sistema debe validar que el usuario digitó las variables con el patrón: caracter alfabético (carácter alfabético | carácter numérico | _)*.

Fuente: Los autores.

Tabla 10. Restricción 6

6. Comprobar sintaxis en funciones
Descripción:
El sistema debe validar si el usuario digitó las funciones del problema de programación lineal, en el estándar válido.
Criterios de aceptación:
● El sistema debe validar que se escribió la función de la función objetivo con el estándar: (variable | constante variable) (+ (variable | constante variable))*. ● El sistema debe validar que se escribieron las restricciones con el estándar: -? (variable | constante variable) ((+ | -) (variable | constante variable))* (= | >= | <=) (constante | constante variable).

Fuente: Los autores.

Tabla 11. Restricción 7

7. Comprobar uso de variables
Descripción:
El sistema debe validar si el usuario digitó en las funciones las variables que planteó.
Criterios de aceptación:
● El sistema debe validar que en la función de la función objetivo se escribieron sólo y todas las variables que se plantearon. ● El sistema debe validar que en la función de cada restricción se escribieron sólo las variables que se plantearon.

Fuente: Los autores.

- **Generalidades de sintaxis**: Esta característica permite al usuario digitar los datos del problema de programación lineal de una manera cómoda.

Tabla 12. Restricción 8

8. Orden de escritura
Descripción:
El sistema debe permitir al usuario escribir los datos del problema de programación lineal en el orden que desee.
Criterios de aceptación:
El usuario tiene libertad para plantear el problema de programación lineal.

Fuente: Los autores.

Tabla 13. Restricción 9

9. Elección de forma de optimizar
Descripción:
El sistema debe permitir al usuario elegir de una manera simple la opción de maximizar o minimizar la función objetivo.
Criterios de aceptación:
Para minimizar o maximizar la función objetivo, el usuario debe seguir los mismos pasos.

Fuente: Los autores.

4.5.2 Restricciones del sistema

Una vez finalizada la etapa de análisis, las herramientas necesarias para diseñar e implementar el DSL son las siguientes.

- **Lenguajes de programación**: El lenguaje de programación a utilizar será Ruby.

- **Metodología de desarrollo**: Programación extrema.

4.5.3 Atributos del sistema

Las restricciones implícitas en el proceso de diseño e implementación son las siguientes.

- **Usabilidad**: El lenguaje debe tener alto nivel de usabilidad.

- **Mantenibilidad**: El código fuente del sistema deberá estar comentado y codificado con notación camel.

5. DISEÑO

5.1 Introducción

En este capítulo se hace uso del modelo y las restricciones generadas en la fase de análisis para definir la sintaxis y la gramática del DSL. Para lograr esto se definen primero los conceptos necesarios para determinar la sintaxis, una vez definida la sintaxis, se procede a especificar la gramática del lenguaje, detallando claramente los token, gramáticas regulares y lenguaje regular del DSL.

A continuación se exponen los conceptos básicos que se deben tener en cuenta para la comprensión del tema tratado en este capítulo.

5.1.1 Ruido sintáctico

El ruido sintáctico se refiere a símbolos en el lenguaje que afectan a la forma de la información, elementos que desde la perspectiva del programador son superfluos y que pueden hacer tediosa la tarea de codificar, por ejemplo, el punto y coma (;) que utilizan lenguajes como C y Java para especificar el fin de una sentencia.

5.1.2 Interfaz fluida

Es el patrón por medio del cual se logra que la sintaxis del DSL sea muy próxima al lenguaje natural. “La interfaz fluida es lo que diferencia a una API de un DSL”[94].

Figura 28. APA para crear objeto computador

```
Processor p = new Processor(2, 2500, Processor.Type.i386);
Disk d1 = new Disk(150, Disk.UNKNOWN_SPEED, null);
Disk d2 = new Disk(75, 7200, Disk.Interface.SATA);
return new Computer(p, d1, d2);
```

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p 67.

[94] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 68.

En la Figura 28 se muestra un ejemplo de un Api tradicional para crear un objeto computador; a continuación se reseñan los patrones de interfaz fluida propuestos por Martin Fowler y se muestra la implementación del mismo ejercicio planteado en la Figura 28 bajo cada patrón de interfaz fluida, de esa forma se puede apreciar que es más agradable a la vista e intuitiva la forma de especificar los ejercicios con interfaz fluida. Esto es por lo que abogan los DSLs, una interfaz fluida, intuitiva, agradable y con poco ruido sintáctico.

5.1.3 Patrones de interfaz fluida

Métodos encadenados[95]: En este patrón se invoca una secuencia de métodos los cuales devuelven su resultado al método que los invocó. "El encadenamiento de métodos permite componer fácilmente varias llamadas a métodos sin depender de muchas variables, lo que resulta en un código que parece fluir, sintiéndolo más como el lenguaje natural"[96].

Figura 29. Implementación bajo métodos encadenados de creación objeto computador

```
computer()
  .processor()
    .cores(2)
    .speed(2500)
    .i386()
  .disk()
    .size(150)
  .disk()
    .size(75)
    .speed(7200)
    .sata()
  .end();
```

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p 67.

[95] Por su nombre en inglés Method Chaining

[96] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 68.

Secuencia de funciones[97]: Según Fowler[98], mediante este patrón las secuencias se pueden leer con tanta claridad como con el patrón de Métodos encadenados. Está orientado a ser implementado en lenguajes no orientados a objetos, ya que se manipulan funciones y no métodos.

Figura 30. Implementación bajo secuencia de funciones de creación objeto computador

```
computer();
  processor();
    cores(2);
    speed(2500);
    i386();
  disk();
    size(150);
  disk();
    size(75);
    speed(7200);
    sata();
```

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 68.

Funciones anidadas[99]: Según Fowler[100], este patrón combina funciones haciendo llamados a los argumentos de funciones de nivel superior. El resultado es un agrupamiento de invocaciones a funciones. Usando este patrón se puede ver claramente la estructura jerárquica del dominio, con los patrones Métodos encadenados y secuencia de funciones se puede apreciar esta jerarquía solamente usando estándares de sangría.

[97] Por su nombre en inglés Function Sequence

[98] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 68.

[99] Por su nombre en inglés Nested Function

[100] Fowler and Parson. Op. cit., p. 70.

Figura 31. Implementación bajo funciones anidadas de creación objeto computador

```
computer(
  processor(
    cores(2),
    speed(2500),
    i386
  ),
  disk(
    size(150)
  ),
  disk(
    size(75),
    speed(7200),
    SATA
  )
);
```

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 70.

Lista Literal[101]: Según Fowler[102], este patrón captura una lista de elementos, que pueden ser de diferentes tipos o del mismo, sin tamaño fijo. Las expresiones se escriben como una lista de instrucciones.

Figura 32. Implementación bajo lista literal de creación objeto computador

```
computer(
  processor (...),
  disk(...),
  disk(...)
);
```

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 74.

[101] Por su nombre en inglés Literal List

[102] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 73.

Mapa Literal[103]: Tal como expresa Fowler[104], mediante este patrón se parametrizan las instrucciones a ejecutar mediante mapas, también llamados Hash o diccionario.

Figura 33. Implementación bajo mapa literal de creación objeto computador

```
computer(processor(:cores => 2, :type => :i386),
         disk(:size => 150),
         disk(:size => 75, :speed => 7200, :interface => :sata))
```

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 74.

5.2 Elección de la interfaz fluida

Para elegir cuál patrón de interfaz fluida usar, se siguieron las pautas recomendadas por Martin Fowler[105] , las cuales sugieren que se elija el patrón de interfaz fluida considerando la lógica que tendrá la gramática. Así pues, en la siguiente figura se aprecian las estructuras que pueden tener las gramáticas y sus respectivos BNF[106], además se sugieren los patrones de interfaz fluida con los cuales se debería implementar cada tipo de gramática.

Tabla 14. Tabla comparativa de patrones de interfaz fluida.

Estructura	BNF	Considere
Lista obligatoria	parent ::= first second third	Función anidada
Lista opcional	parent ::= first maybeSecond? maybeThird?	Métodos encadenados, mapa literal
Contenedor homogéneo	parent ::= child*	Lista literal, Secuencia de funciones
Contenedor heteronéneo	parent ::= (this | that | theOther)*	Métodos encadenados
Conjuntos	n/a	Mapa literal

Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 75.

[103] Por su nombre en inglés Literal Map

[104] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 70.

[105] Ibid., p. 75.

[106] Notación Backus-Naur

De acuerdo a la Tabla 14 y teniendo en cuenta lo estipulado en la fase de análisis, la gramática planteada a implementar no es una lista obligatoria pues el orden de los elementos (después de la primera línea) no es estricto, así que, las peticiones se pueden hacer en cualquier orden. Por ende, si es una lista opcional pues la primera línea siempre es la misma, mientras que las demás pueden variar. Tampoco es un contenedor homogéneo pues los elementos de las sentencias no son del mismo tipo, de igual manera, tampoco es un contenedor heterogéneo, ya que las sentencias no se pueden repetir. De acuerdo al tipo de gramática en los cuales se encuentra la gramática del DSL planteado, se decidió hacer uso del patrón de interfaz fluida Métodos encadenados. A pesar de que existía la opción de usar el patrón de Mapa literal pues la gramática a implementar es una lista opcional, se decidió usar el patrón Métodos encadenados puesto que, a consideración de los autores, genera menos ruido sintáctico el patrón Métodos encadenados que el patrón Mapa literal.

Cabe recordar que según Fowler[107] la ventaja de estos patrones de interfaz fluida es que estimulan la reflexión sobre las técnicas que se pueden utilizar y sobre cómo leer el DSL, por otro lado, la desventaja es que pueden convertirse en continuas elecciones bajo preferencia personal.

5.3 Definición de la sintaxis

Para diseñar la sintaxis del DSL, primero se plantea la sintaxis ideal para el usuario (sin ninguna restricción). Para el dominio de la programación lineal esta tarea es sencilla pues se conoce la sintaxis que generalmente utilizan los expertos del dominio para plantear sus problemas, dicha sintaxis corresponde a la estructura de la Figura 34.

Figura 34. Planteamiento de un problema de programación lineal en el lenguaje natural

$$\text{Maximize: } z = 15x + 10y$$
$$\text{Subject to:}$$
$$1/3x + 1/2y = 100$$
$$1/3x + 1/6y = 80$$

Fuente: Los autores.

[107] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 68.

Siguiendo la metodología del patrón de interfaz fluida Métodos encadenados, conociendo el azúcar sintáctico de Ruby, también, teniendo en cuenta el modelo estipulado en la fase de análisis y abogando porque la sintaxis fuese similar a la sintaxis de la Figura 34 evitando ruido sintáctico, se planteó la sintaxis mostrada en la Figura 35 después de en un proceso iterativo de diseño.

Figura 35. Planteamiento de un problema de programación lineal en la sintaxis del DSL

```
LinearProgramming
  .variables('x', 'y')
  .max('15x+10y')
  .subjectTo(
   '1/3x + 1/2y = 100',
   '1/3x + 1/6y = 80')
```

Fuente: Los autores.

Como se aprecia en la Figura 35, la primera línea invoca la clase desde la cual se van a llamar los métodos, ésta clase toma el nombre de LinearProgramming debido a que se usará la técnica de optimización llamada programación lineal.

Puesto que se pretende diseñar una sintaxis intuitiva para el usuario, las siguientes líneas pueden ser nombradas de diferentes formas, por consiguiente en la segunda línea se puede escribir variables o var, e inmediatamente se definen los tokens que serán reconocidos en la función objetivo y en las restricciones. Así mismo, en la tercera línea se escribe max, maximize o maximizar si se requiere maximizar, de lo contrario se escribe min, minimize o minimizar si requiere minimizar. Finalmente en la cuarta línea se debe escribir el identificador subjecTo, sujetoA, restrictions o restricciones y entre paréntesis se escriben las funciones que representan a las restricciones del problema, las restricciones se pueden separar por un espacio, sin embargo, por legibilidad se recomienda separarlas con un salto de línea.

Los identificadores (variables, max o min, subjectTo) se pueden escribir en cualquier orden, es decir que la única línea que no puede cambiar de orden es la primera, debido a que ésta es la que identifica la clase.

En comparación con la sintaxis definida en la Figura 34, la sintaxis definida para el DSL presenta ruido sintáctico al usar los puntos para separar los métodos, en el uso de paréntesis, comillas sencillas o comillas dobles, además difieren las dos sintaxis

en que en la de la Figura 34 no se estipulan las variables, sin embargo son necesarias escribirlas en la sintaxis de la Figura 35 pues de esta forma se evitan ambigüedades en las funciones.

Sin embargo, Fowler expresa que no se debe tratar de hacer que el DSL se lea como el lenguaje natural. En efecto, ha habido varios intentos de hacerlo en GPL's, con Applescript como el ejemplo más obvio, suscitando el problema de que se genera una gran cantidad de azúcar sintáctica que complica la comprensión de la semántica. Por otra parte, se debe tener en cuenta que un DSL es un lenguaje de programación, así que usarlo se debe sentir como programando. Así pues, en DSL debe acarrear un mínimo de ruido sintáctico.

5.4 Definición de la gramática

La gramática se determina para definir formalmente el lenguaje y para limitar su expresividad, pues de acuerdo a lo que en este apartado se define, posteriormente se implementan los analizadores del código que comprueban que se cumplan todas las reglas gramaticales.

Así pues se procede a definir los tokens que reconoce el lenguaje, posteriormente se definen las expresiones regulares del lenguaje, ya que con ellas se construye por último el lenguaje regular.

5.4.1 Tokens

Los tokens que deben ser tenidos en cuenta para la definición del lenguaje son los siguientes:

Palabras reservadas: Las palabras reservadas pueden ser escritas solamente una vez, ellas son:

- LinearProgramming
- variables
- var
- min
- minimize
- minimizar
- max
- maximize

- maximizar
- subjectTo
- sujetoA
- restrictions
- restricciones

Delimitadores: Los símbolos utilizados como separadores de las distintas construcciones del lenguaje son:

- **Paréntesis**: Agrupan a las variables, función objetivo y restricciones.

- **Comas**: Separan los elementos de la lista de variables y de la lista de restricciones.

- **Puntos**: Indican el final de una sentencia, también sirve para separar la parte fraccionaria de la parte entera de un número en las funciones.

- **Comillas simples y comillas dobles**: Encierran a cada variable, a la función objetivo y a cada restricción.

Caracteres alfabéticos: Cualquier letra del abecedario, omitiendo la ñ. Esto se puede representar con la siguiente expresión regular:

[a,..., z, A, …, Z] (sin incluir [ñ, Ñ]).

Caracteres alfanuméricos: Cualquier letras del abecedario, número arábigo o símbolo "_". Esto se representa con la siguiente expresión regular:

[a,..., z, A, …, Z] (sin incluir [ñ, Ñ]) y [0, 1,..., 9] y [_, .].

Constantes: Los caracteres numéricos [0.0, 0.1, 1, 1.0, ..., 9].

Operaciones matemáticas: +, -, *, /.

Símbolos de comparación: =, <=, >=.

Comentarios: Los comentarios se pueden escribir de dos formas:
- Los comentarios de una línea inician con el carácter #.
- Los comentarios multilínea inician con la sentencia =begin y termina con la sentencia =end.

5.5.2. Gramáticas regulares

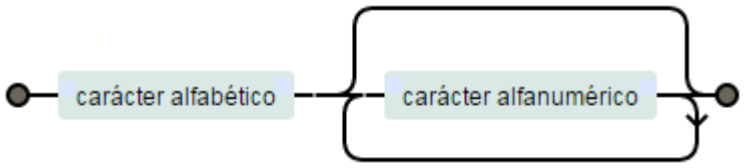
En cada uno de los diagramas de expresiones regulares que representan a las gramáticas regulares, las etiquetas con comillas dobles (“ ”) denotan que la palabra se debe escribir cómo aparece, sin embargo, cuando la etiqueta consta de una o dos letras y no tiene comillas dobles, debe escribirse también como aparece. Cuando una etiqueta tiene más de dos letras es porque hace alusión a un tipo de Token o a otra expresión regular.

Variables: El primer símbolo debe ser un carácter alfabético. Después de ese primer carácter se pueden poner caracteres alfanuméricos. Las mayúsculas y las minúsculas se consideran diferentes.

Ejemplos válidos: x, x1, X_1, xy, xY1, x.y
Ejemplos no válidos: 1, 1x, x y, ñ1, á1

La expresión regular que representa a las variables es la plasmada en la Figura 36.

Figura 36. Expresión regular de variables



Fuente: Los autores.

De igual forma su notación Backus-Naur es:

caracter alfabético (carácter alfanumérico)*

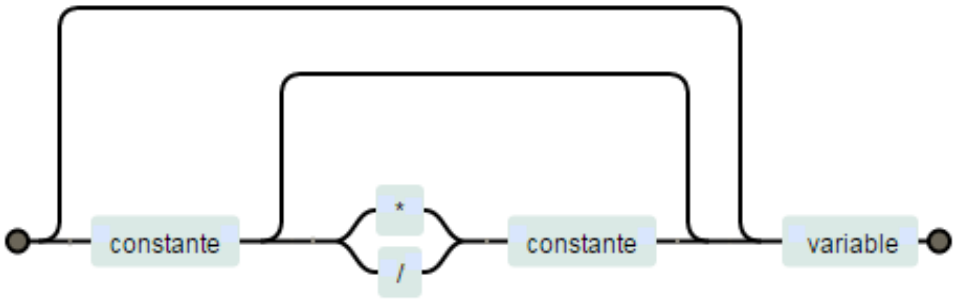
Expresiones: Las expresiones deben cumplir alguno de los siguientes patrones:

- Variable
- Constante seguida de variable
- Constante dividida (/) o multiplicada (*) por otra constante, seguida de variable.

Ejemplos válidos: x, 2, 2x, 2.2x, 2/3x, 2*3x.
Ejemplos no válidos: x2, x/2, 2/x.

La expresión regular que representa a las expresiones es la plasmada en la Figura 37.

Figura 37. Expresión regular de expresiones de una función



Fuente: Los autores.

De igual forma su notación Backus-Naur es:

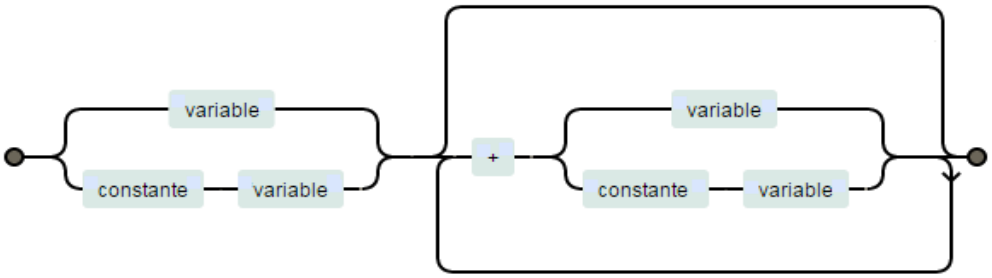
(constante ((* | /) constante)?)? variable

Funciones de función objetivo: Debe tener una o más expresiones, si la función tiene más de una expresión, cada expresión debe separarse con una suma (+). Por otra parte, la función debe contener como mínimo una variable.

Ejemplos válidos: x, 2x, x + y, x - 3x, 2.2x * 3.
Ejemplos no válidos: 3, 2 + 3, +x, x^2, x = 2.

La expresión regular que representa a la función de la función objetivo es la plasmada en la Figura 38.

Figura 38. Expresión regular de funciones de la función objetivo



Fuente: Los autores.

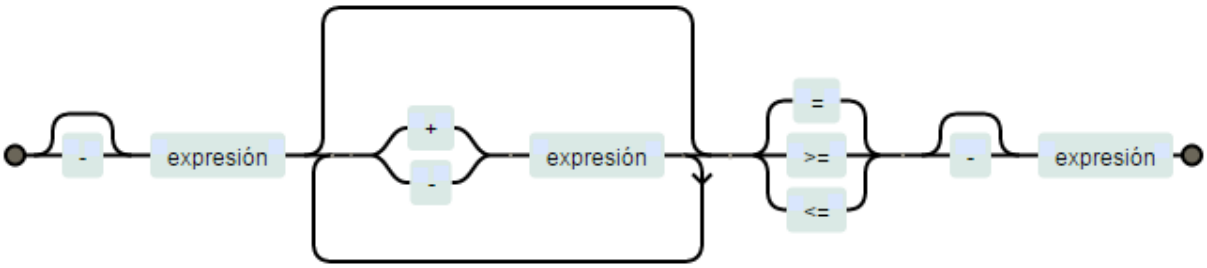
De igual forma su notación Backus-Naur es:

expresión (variable | constante variable) (+ (variable | constante variable))*

Funciones de restricciones: Este tipo de función inicia con una expresión que puede ser negativa, así mismo, puede contener una sumatoria de funciones hasta que se escribe un signo de comparación (=, >= o <=) y finaliza con una expresión simple.

Ejemplos válidos: x = 2, 3x <= 2y, -x + 3y + 8 >= 4, 1/2x - 2*3y = 14.
Ejemplos no válidos: 2x, 2x >= 4x + 3, 2x < 4.

Figura 39. Expresión regular de funciones de restricciones



Fuente: Los autores.

De igual forma su notación Backus-Naur es:

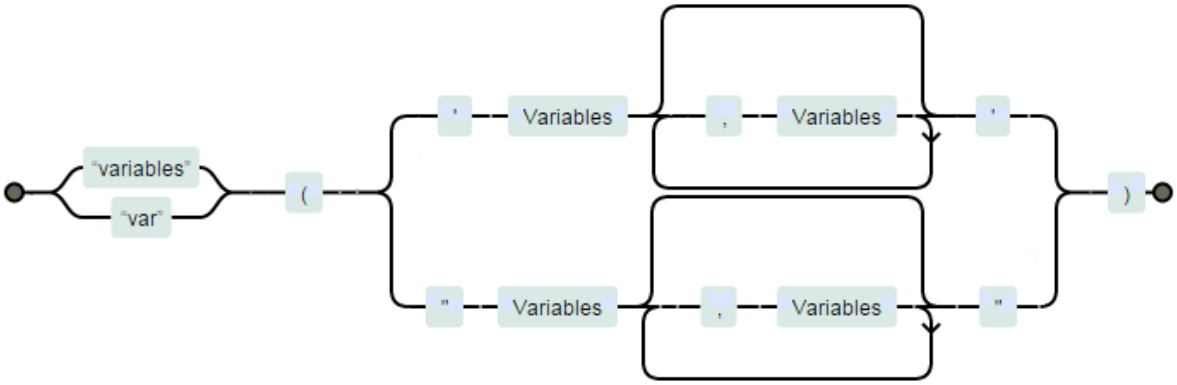
-? expresión ((+ | -) expresión)* (= | >= | <=) -? expresión

Métodos: Los parámetros de los métodos deben ir antecedidos de alguna de las palabras reservadas: variables, var, min, minimize, minimizar, max, maximize, maximizar, subjectTo, sujetoA, restrictions, restricciones y estar encerrados entre paréntesis, de igual forma, cada parámetro debe estar encerrado entre comillas simples (‘ ’) o entre comillas dobles (“ ”), si se escribe más de un parámetro, deben estar separados por una coma (,). Si la palabra reservada antecedida al parámetro es variables o var, los parámetros serán de tipo Variables, por otro lado, si la palabra reservada antecedida al parámetro es min, minimize, minimizar, max, maximize o maximizar, solamente se puede escribir un parámetro de tipo Función de función objetivo, mientras que si la palabra reservada antecedida al parámetro es subjectTo, sujetoA, restrictions o restricciones los parámetros serán de tipo Función de restricciones.

Ejemplos válidos: variables(“x”), variables(‘x’, ‘y’), variables(‘x’, “y”), variables(“x”, “y”), max(‘x+y’), min(“x+y”), subjectTo('1y/3x + 1/2y = 100', '1/3x + 1/6y = 80').

Ejemplos no válidos: variables “x”, variables(x), variables(‘x’ ‘y’), min(x+y).

Figura 40. Expresión regular método variables

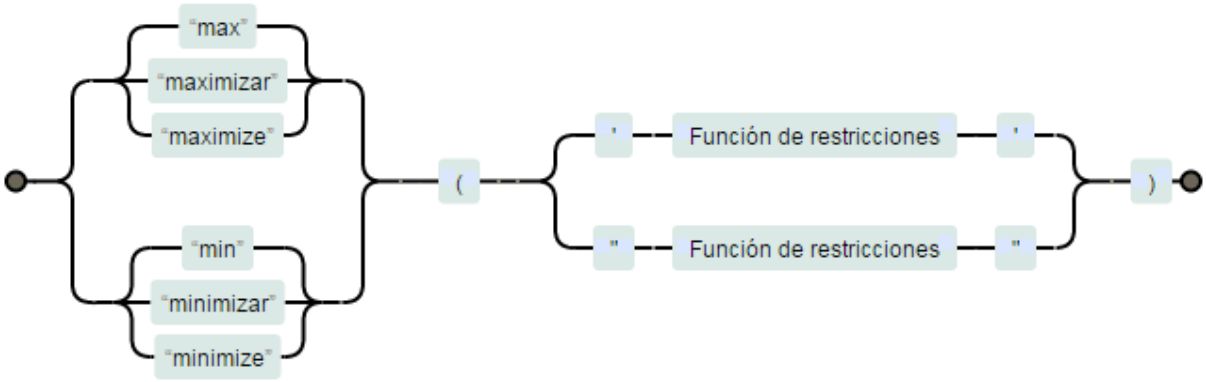


Fuente: Los autores.

De igual forma su notación Backus-Naur es:

(‘variables’ | ‘var’)\((' Variables(,Variables)* ' | "Variables(,Variables) * ") \)

Figura 41. Expresión regular método función objetivo

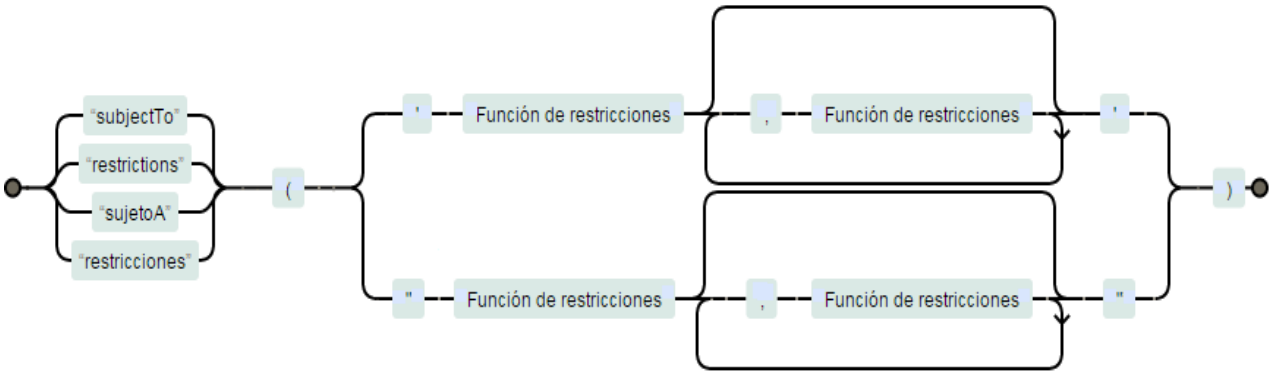


Fuente: Los autores.

De igual forma su notación Backus-Naur es:

((‘max’ | ‘maximizar’ | ‘maximize’) | (‘min’ | ‘minimizar’ | ‘minimize’)) \(('Función de restricciones' | " Función de restricciones ") \)

Figura 42. Expresión regular método restricción



Fuente: Los autores.

De igual forma su notación Backus-Naur es:

(‘subjectTo’ | ‘restrictions’ | ‘sujetoA’ | ‘restricciones’) \(('Función de restricciones (,Función de restricciones)* ') | ("Función de restricciones(,Función de restricciones)* ") \)

Cambio de método: Para marcar el inicio de un método, se debe escribir punto (.). Este punto puede ser escrito seguidamente al método anterior.

Ejemplo: variables(‘x’, ’y’).min(‘x+y')

También se puede escribir el punto dejando cualquier cantidad de espacios en blanco después del método anterior.

Ejemplo: variables(‘x’, ’y’) .min(‘x+y')

Como última opción el punto puede ser escrito haciendo un salto de línea después del método anterior.

Ejemplo: variables(‘x’, ’y’)
.min(‘x+y')

El método también puede ser escrito dejando cualquier cantidad de espacios en blanco después del punto.

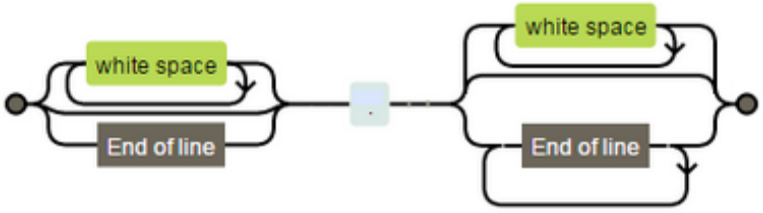
Ejemplo: variables(‘x’, ’y’). min(‘x+y')

Además también se puede escribir el método después de hacer cualquier cantidad de saltos de línea después del punto.

Ejemplo: variables(‘x’, ’y’).

min(‘x+y')

Figura 43. Expresión regular cambio de método



Fuente: Los autores.

De igual forma su notación Backus-Naur es:

(\s+ || $) . (\s+ || ($)+)

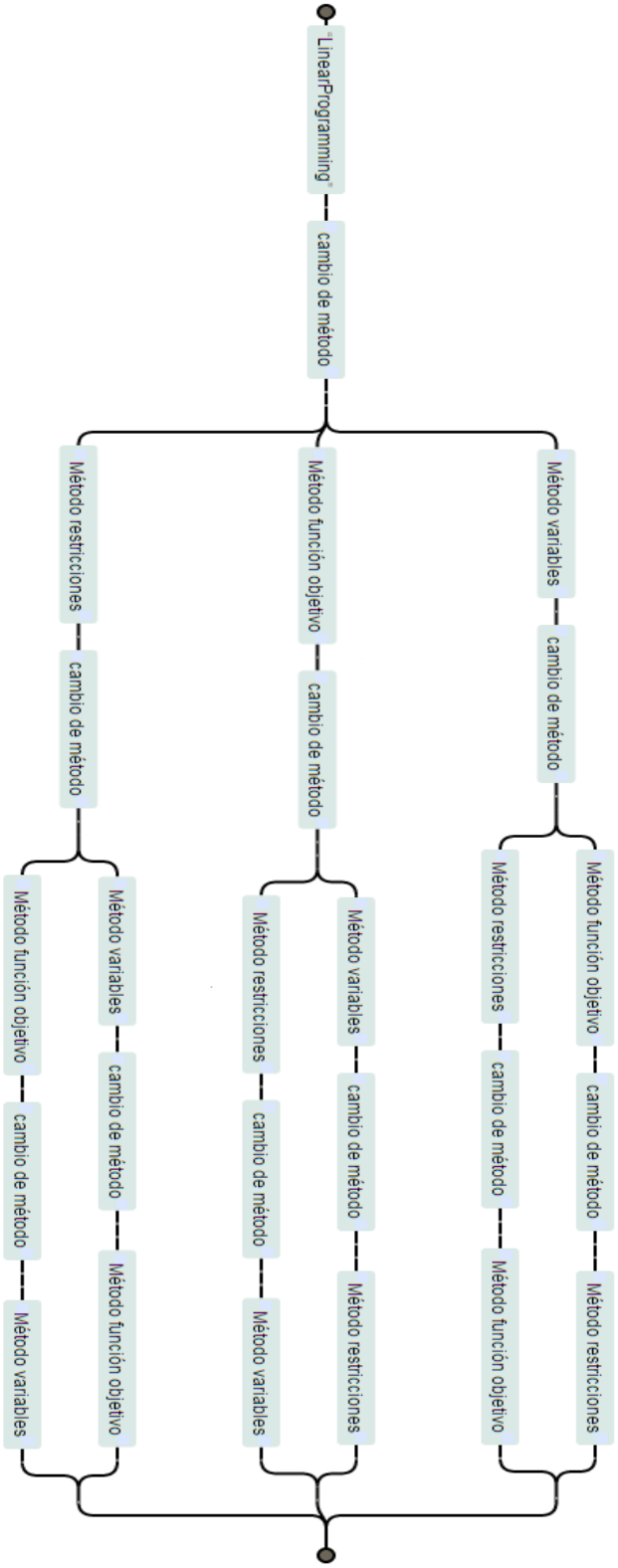
Lenguaje regular: Tal como expresa Kleene en su teorema “Un lenguaje es regular si y sólo si es aceptado por un autómata finito”[108]. Es por ello que para demostrar que el DSL implementado es un lenguaje regular, a continuación se muestra el gráfico que representa su autómata finito.

La notación Backus-Naur es:

> LinearProgramming (cambio de método) (Método variables (cambio de método) (Método función objetivo (cambio de método) (Método restricciones) | Método restricciones (cambio de método) (Método función objetivo)) | Método función objetivo (cambio de método) (Método variables (cambio de método) (Método restricciones) | Método restricciones (cambio de método) (Método variables)) | Método restricciones (cambio de método) (Método variables (cambio de método) (Método función objetivo) | Método función objetivo (cambio de método) (Método variables)))

[108] Dean Kelley. Teoría de autómatas y lenguajes formales. Prentice Hall, 1995. p. 81.

Figura 44. Autómata finito del DSL



Fuente: Los autores.

6. IMPLEMENTACIÓN

6.1. Introducción

En esta fase el DSL se transforma en código escrito en el lenguaje anfitrión elegido (en este caso Ruby), utilizando las técnicas vistas en el capítulo 2 para lograr implementar el DSL con la menor cantidad de ruido sintáctico posible y también para realizar las validaciones estipuladas tanto en la fase de análisis como en la fase de diseño.

Para abordar el capítulo primero se realizan definiciones de componentes de la arquitectura diseñada por Martin Fowler para implementar DSLs internos, posteriormente se implementan dichos componentes y por último se implementan las validaciones del lenguaje.

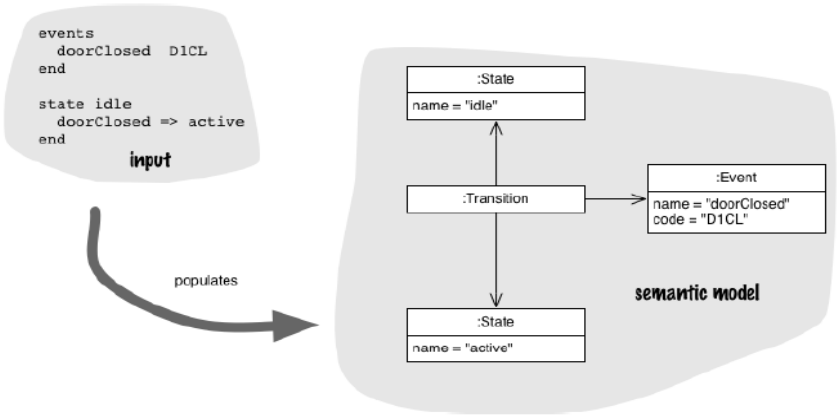
A continuación se exponen los conceptos básicos que se deben tener en cuenta para la comprensión del tema tratado en este capítulo.

6.1.1. Modelo semántico

"Cuando la gente habla de un lenguaje de programación, a menudo se oye hablar de la sintaxis y la semántica. La sintaxis capta las expresiones legales del programa, todas las expresiones soportadas por la gramática. La semántica de un programa es lo que él significa, es decir, lo que hace cuando se ejecuta. En el caso de los DSLs, el modelo define la semántica. "[109] Por ende, el modelo semántico es la abstracción a un lenguaje de programación del modelo planteado en la fase de análisis, se codifican las entidades y los servicios para que el usuario pueda darle valor a los objetos de valor, previo uso del constructor de expresiones. De esta forma usando un lenguaje de programación orientado a objetos, el modelo semántico es una clase en la cual las entidades y servicios pueden ser métodos o variables.

[109] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 16.

Figura 45. Especificación del modelo semántico

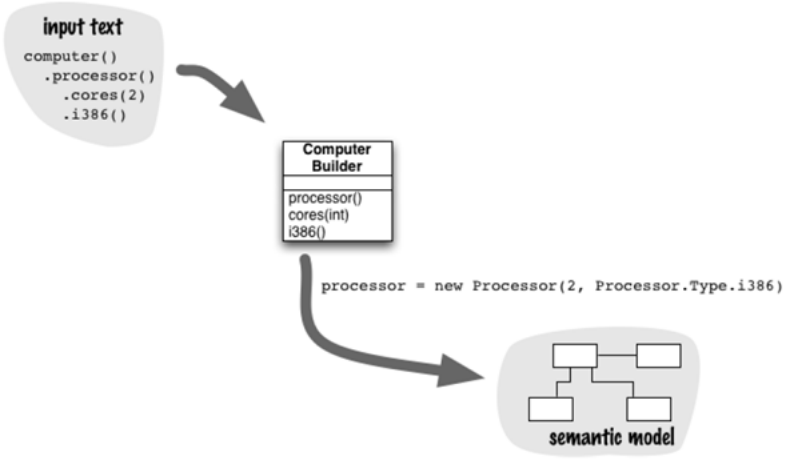


Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 159.

6.1.2. Constructor de expresiones

La interfaz fluida debe pasar por una capa de análisis llamada constructor de expresiones, la cual es una estructura de datos que traduce lo definido en una interfaz fluida a la forma que el lenguaje anfitrión la puede interpretar y darle tratamiento a los datos; el constructor de expresiones es la interfaz entre el modelo y los datos.

Figura 46. Especificación del constructor de expresiones



Fuente: Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010. p. 343.

En la figura 46 se aprecia cómo el constructor de expresiones es la capa intermedia entre el texto que digita el usuario y el modelo semántico, es por ello que en los lenguajes de programación internos, el constructor de expresiones cumple las labores de análisis que se realizan en un compilador.

Fowler[110] recomienda que se traten como elementos diferentes al modelo semántico y al constructor de expresiones, principalmente para que existe una separación de responsabilidades, además, un modelo semántico claro permite probar la semántica y el análisis del DSL por separado, se puede probar la semántica poblando el modelo semántico directamente y ejecutando las pruebas unitarias, de igual forma, se puede probar el analizador observando si éste pobla el modelo semántico con los elementos correctos. Por otra parte si se tiene más de un analizador, se puede probar si producen salidas semánticamente equivalentes comparando cómo poblan el modelo semántico, esto hace que sea fácil soportar múltiples DSL y evolucionar el DSL por separado del modelo semántico.

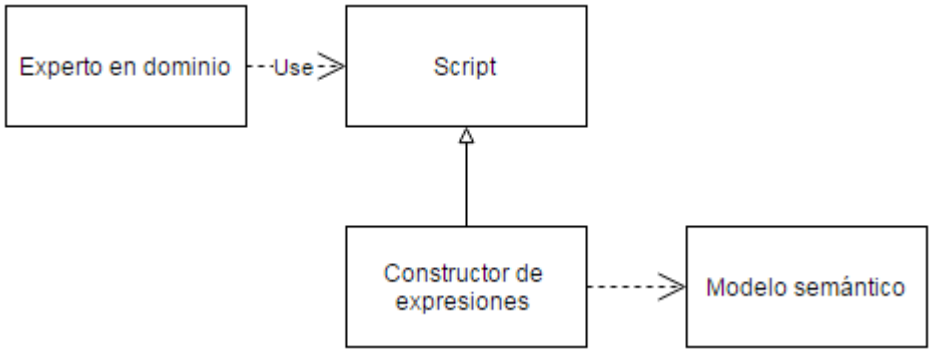
6.3. Arquitectura del DSL

Es necesario hacer que la interfaz poblada por el usuario (Script) sea compatible con la interfaz del modelo semántico, es decir, crear las estructuras necesarias para traducir las peticiones del usuario a la forma que el modelo semántico reconoce; es por ello que se hace uso del patrón de diseño adaptador, ya que éste “hace compatibles dos clases que no lo son, para que trabajen cooperativamente”[111]. La clase adaptador que une a las dos interfaces mencionadas es el constructor de expresiones.

[110] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010, p. 344.

[111] Erich Gamma, Richard Helm, Ralph Johnson, Jhon Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995, p. 157.

Figura 47. Patrón de diseño Adapter



Fuente: Los autores.

En la Figura 47 se plasma la colaboración entre los componentes del sistema, cabe aclarar que lo único que se implementa es el constructor de expresiones y el modelo semántico, el Script es simplemente la interfaz por la cual se invocan los métodos del constructor de expresiones.

6.4. Implementación del modelo semántico

Los elementos necesarios para plantear un modelo de optimización son la función objetivo y las restricciones; se implementó la función objetivo como una cadena; las restricciones como una tabla hash en la que la llave es la letra r seguida del número que corresponde a la posición en la secuencia en la que fue escrita y el valor es la cadena que representa a la función[112]. El modelo semántico se compone también de una tabla hash para especificar las variables, los elementos de la tabla tendrán por clave el nombre que se le asigne a la variable y el valor será el número que tome dicha variable una vez se use un solver. De esta forma el modelo semántico corresponde a la Figura 48.

[112] Para las restricciones planteadas en la Figura 37, la tabla Hash es {"r1"=>"1/3x + 1/2y = 100", "r2"=>"1/3x + 1/6y = 80"}

Figura 48. Implementación modelo semántico

```
class Modelo
  attr_accessor :variables, :funcionObjetivo, :restricciones

  def initialize()
    @variables = Hash.new()
    @funcionObjetivo = ''
    @restricciones = Hash.new()
    self
  end
end
```

Fuente: Los autores.

6.5. Implementación del constructor de expresiones

Las funciones se implementan en el constructor de expresiones como funciones de clase para evitar crear una instancia de la clase y con ello evitar ruido sintáctico[113]. Además para asegurar que los métodos se pueden invocar en cualquier orden, se inicializa el objeto instancia del modelo semántico desde el primer método que se invoque.

Cada método invocado desde el Script lo que hace es poblar o darle valor al atributo correspondiente de la clase del modelo semántico como se aprecia en la Figura 49.

[113] Si las funciones no son instancias de clase, habría que crear un objeto de la clase, por ende la primera línea sería de la forma: nuevoObjeto = LinearProgramming.new, lo cual es ruidoso para un usuario que no esté acostumbrado a usar lenguajes de programación.

Figura 49. Implementación constructor de expresiones

```
class LinearProgramming
 attr_accessor :modelo

 def self.initialize()
  if @modelo == nil
    @modelo = Modelo.new()
  end
  self
 end

 def self.variables (*variables)
  initialize
  variables.each do |variable|
    @modelo.variables["#{variable}"]=nil
  end
  self
 end

 def self.max (function)
  initialize
    @modelo.funcionObjetivo = function
  self
 end

 def self.min (function)
  initialize
    @modelo.funcionObjetivo = function
  self
 end

 def self.subjectTo(*functions)
  initialize
  i = 0
  functions.each do |function|
    i = i+1
    @modelo.restricciones["r#{i}"]=function
  end
  self
 end
end
```

Fuente: Los autores.

También desde esta clase se realizan las validaciones de la sintaxis del DSL, dichas validaciones se exponen en la siguiente sección.

Tabla 15. Flujo de instrucciones en el DSL

Script	Constructor de expresiones	Modelo semántico
LinearProgramming	class LinearProgramming	class Modelo
.variables('x', 'y')	def self.variables(*variables) initialize variables.each do |variable| @modelo.variables["#{variable}"]=nil end self end	variables
.min('15x+10y')	def self.max (function) initialize @modelo.funcionObjetivo = function self end	funcionObjetivo
	def self.min (function) initialize @modelo.funcionObjetivo = function self end	
.subjectTo('1/3x + 1/2y = 100', '1/3x + 1/6y = 80')	def self.subjectTo(*functions) initialize i = 0 functions.each do |function| i = i+1 @modelo.restricciones["r#{i}"]=function end self end	restricciones

Fuente: Los autores.

En la Tabla 15 se observa la interacción entre los módulos de la arquitectura, la secuencia es escribir primero la palabra LinearProgramming, la cual llama a la clase LinearProgramming sin instanciarla. Seguidamente se puede escribir cualquiera de los identificadores .variables, .min, .max o .subjectTo (los nombres de los métodos en la clase LinearProgramming), con sus respectivos valores de parámetros. Al escribir alguno de estos identificadores se invoca al método de clase correspondiente en la clase LinearProgramming, en dichos métodos se modifica el atributo correspondiente de la clase Modelo.

6.6. Implementación de verificaciones

Para implementar analizadores de sintaxis, Ruby cuenta con excelentes herramientas como técnicas de metaprogramación y expresiones regulares. A

continuación se exponen las restricciones en la sintaxis del lenguaje y se explican las técnicas usadas para su resolución.

6.6.1. Restricciones:

Teniendo en cuenta que otros tipos de restricciones léxicas son detectadas por el intérprete de Ruby, las restricciones propias del DSL son las siguientes:

1. Solamente se pueden escribir como métodos las palabras reservadas.

2. Solo se pueden declarar una vez las variables, la función objetivo y las restricciones.

3. Se les debe dar valor a las variables, la función objetivo y las restricciones.

4. Cada variable debe iniciar con un carácter alfabético y contener sólo caracteres alfanuméricos y guiones bajos (_).

5. Todas las variables declaradas, tienen que estar en la función objetivo.

6. La función objetivo, debe obedecer a la expresión regular:

   (variable | constante variable) ((+ | -) (variable | constante variable))*

7. Las restricciones, deben obedecer a la expresión regular:

   (variable | constante variable) ((+ | -) (variable | constante variable))* (= | >= | <=) (constante | constante variable).

8. Variable no puede ir en el divisor.

9. Números reales.

10. El método variable también se puede llamar var, el método max también se puede llamar maximize o maximizar, el método min también se puede llamar minimize o minimizar y el método subjectTo puede ser llamado bajo el nombre de sujetoA, restrictions o restricciones.

6.6.2. Verificaciones

Tabla 16. Verificación 1

Restricción verificada: 1, 10.
Se hizo uso del método de metaprogramación llamado method_missing. Según[114] cuando se llama un método, Ruby entra en la clase del constructor de expresiones (Linear Programming) y busca entre sus métodos. Si no puede encontrar el método en esa clase, busca en las clases de orden superior, luego en la clase Object y finalmente en el Kernel, haciendo de esta forma reflexión o metaprogramación reflexiva en el código. [115]Si no encuentra el método en ninguna de estas clases, se llama y ejecuta el método method_missing en la clase receptora inicial (LinearProgramming) y si el método invocado es maximizar o maximize, se llama al método max, asi mismo, si el método invocado es minimizar o minimize, se llama al método min, además si el método invocado es var, se llama al método variables, por otro lado, si el método invocado es restrictions, sujetoA o restricciones, se llama al método subjectTo, finalmente si el método invocado no coincide con ninguno de estos identificadores, se le envia al usuario el mensaje: “Palabra … incorrecta, rectifique porfavor.”
def self.method_missing(method, *parameters) case when method.to_s == 'maximizar' || method.to_s == 'maximize' then max(parameters[0]) when method.to_s == 'minimizar' || method.to_s == 'minimize' then min(parameters[0]) when method.to_s == 'var' then variables(parameters) when method.to_s == 'restrictions' || method.to_s == 'sujetoA' || method.to_s == 'restricciones' then subjectTo(parameters) else puts "Palabra '#{method}' incorrecta, rectifique por favor" end self end

Fuente: Los autores.

[114] Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010. p. 72.

[115] En Ruby los métodos alias y alias_method sirven para darle sinónimos a otros métodos, sin embargo, como los métodos declarados son métodos de clase, no se pueden usar los métodos alias. Es por ello que se tuvieron que implementar los sinónimos de los métodos de esta forma alternativa.

Tabla 17. Verificación 2

Restricción verificada: 2.
Para verificar si al objeto del método llamado ya se le asignó un valor, se establece una condición, la cual verifica si el objeto está vacío, dicha condición es lo primero que realiza el método.

def self.variables (*variables)
  if @modelo.variables.length != 0
  puts "Las variables ya fueron asignadas"
  else
  variables.each do |variable|
  @modelo.variables["#{variable}"]=nil
  end
  end
  self
end

def self.fo(function, opc)
  if @modelo.funcionObjetivo.length != 0
  puts "La función objetivo ya fue asignada"
  else
  @modelo.funcionObjetivo = function
  end
  self
  self
end
 	
def self.max function = nil
  fo(function, 'max')
end
 	
def self.min function = nil
  fo(function, 'min')
end

def self.subjectTo(*functions)
  if @modelo.restricciones.length != 0
  puts "Las restricciones ya fueron planteadas"
  else
  i = 0
  functions.each do |function|
  i = i+1

```
      @modelo.restricciones["r#{i}"]=function
    end
  end
  self
end
```

Fuente: Los autores.

Tabla 18. Verificación 3

Restricción verificada: 3.
Para verificar si el objeto no está vació, se establece una condición, la cual verifica si el objeto está vacío.

`if @modelo.variables.length == 0`
`  puts "Digite las variables del problema"`
`end`

`if @modelo.funcionObjetivo.length == 0`
`  puts "Digite función objetivo"`
`end`

`if @modelo.restricciones.length == 0`
`  puts "Digite las restricciones del problema"`
`end`

Fuente: Los autores.

Tabla 19. Verificación 4

Restricción verificada: 4.
Se hace uso de la expresión regular (carácter alfabético | carácter numérico | _)* la cual se compara con cada llave de la lista de variables[116] para verificar que se cumpla la restricción.

`expresionVariable = /[a-z|A-Z]([a-z|A-Z]|[0-9]|_)*/`
 	
`@modelo.variables.keys.each do |var|`
`  resultadoExpresionVariable = expresionVariable.match(var)`
`  if resultadoExpresionVariable.to_s != var`
`  puts "Variable #{var} debe empezar con una letra y solo puede contener letras y numeros"`

[116] Las variables están almacenadas en una Hash table, es por ello que es necesario escribir la sentencia @modelo.variables.keys para que retorne la lista de las llaves de las variables, ya que en ese campo se almacena el nombre de la variable.

```
        variablesCorrectas = false
    end
end
```

Fuente: Los autores.

Tabla 20. Verificación 5

Restricción verificada: 5.
Se recorre la lista de los nombres de las variables y cada elemento se utiliza en una expresión regular, la cual verifica que ese nombre de variable se encuentre en la función objetivo.

`@modelo.variables.keys.each do |var|`
`  if /#{var}/.match(@modelo.funcionObjetivo) == nil`
`  puts "Variable #{var} no se encuentra en la función objetivo"`
`  end`
`end`

Fuente: Los autores.

Tabla 21. Verificación 6

Restricción verificada: 6, 8, 9.
Para validar las restricciones 6, 8 y 9 en la función objetivo, se hace uso de la siguiente expresión regular: (variable | (constante+(.constante+)?) variable) (+ (variable | (constante+(.constante+)?) variable))* En la cual por medio de la expresión “constante+(.constante+)?” se verifica la restricción 9, ya que cuando se desee agregar parte fraccionaria a un número, solo se escribe un punto, seguido de los números que representen la cantidad fraccionaria. Mediante la expresión regular planteada se satisface la restricción 8 ya que nunca va a estar una variable como denominador. La restricción 6 se valida por medio de toda la expresión regular. Si la búsqueda de la expresión regular en la función objetivo es satisfactoria, retorna un objeto de tipo MatchData el cual se asigna al atributo resultado, sin embargo, es posible que la búsqueda sea satisfactoria parcialmente. Siempre que la búsqueda sea satisfactoria totalmente, en el atributo resultado queda almacenada la misma

cadena que tiene la función objetivo. Así pues, se comparan la función objetivo y el resultado para determinar si se cumplen las restricciones o no.

```
expresionFuncionObjetivo = /\s*(\d+(\.\d+)?#{@modelo.variables.keys}|\d+(\.\d+)?|#{@modelo.variables.keys})\s*(((\+|-|\*)\s*(\d+(\.\d+)?#{@modelo.variables.keys}|\d+(\.\d+)?|#{@modelo.variables.keys})\s*)|(\/\s*(\d+(\.\d+)?#{@modelo.variables.keys}|\d+(\.\d+)?)\s*))*/

resultado = expresionFuncionObjetivo.match(@modelo.funcionObjetivo)
   if resultado.to_s != @modelo.funcionObjetivo
      puts "Funcion a optimizar esta escrita incorrectamente, por favor revise (recuerde que las variables solo pueden ir al final de las expresiones, ejemplo: 2x+3y)"
   end
end
```

Fuente: Los autores.

Tabla 22. Verificación 7

Restricción verificada: 7, 8, 9.
Para validar las restricciones 7, 8 y 9 en las funciones de las restricciones, se hace uso de la siguiente expresión regular: -? (variable | (constante+(.constante+)?) variable) ((+ | -) (variable | (constante+(.constante+)?)variable))* (= | >= | <=) ((constante+(.constante+)?) | (constante+(.constante+)?)variable) En la cual por medio de la expresión “constante+(.constante+)?” se verifica la restricción 9, ya que cuando se desee agregar parte fraccionaria a un número, solo se escribe un punto, seguido de los números que representen la cantidad fraccionaria. Mediante la expresión regular planteada se satisface la restricción 8 ya que nunca va a estar una variable como denominador. La restricción 7 se valida por medio de toda la expresión regular. Se recorre la lista de los valores de las restricciones[117] y se verifica si cada uno satisface a la expresión regular. Si la búsqueda de la expresión regular en los valores de las restricciones es satisfactoria, retorna un objeto de tipo MatchData el

[117] Las restricciones están almacenadas en una Hash table, es por ello que es necesario escribir la sentencia @modelo.restricciones.values para que retorne la lista de los valores de las restricciones, ya que en ese campo se almacena la función de la restricción.

cual se asigna al atributo resultado, sin embargo, es posible que la búsqueda sea satisfactoria parcialmente. Siempre que la búsqueda sea satisfactoria totalmente, en el atributo resultado queda almacenada la misma cadena que tiene el valor de la restricción. Así pues, se comparan el valor de la restricción y el resultado para determinar si se cumplen las restricciones o no.

```
expresionFuncionRestriccion = /\s*-?\s*(\d+(\.\d+)?\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?\s*)?#{@modelo.variables.keys}(\s*(\+|-)\s*((\d+(\.\d+)?)\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?\s*)?#{@modelo.variables.keys})*\s*(=|>=|<=)\s*-?\s*(\d+(\.\d+)?\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?\s*)?#{@modelo.variables.keys}?/

@modelo.restricciones.values.each do |restriction|
   resultado = expresionFuncionRestriccion.match(restriction.to_s)
   if resultado.to_s != restriction.to_s
      puts "Restricción #{restriction} está escrita incorrectamente, por favor revise (recuerde que las variables solo pueden ir al final de las funciones, ejemplo: 2x+3y >= 3x)"
   end
end
```

Fuente: Los autores.

7. PRUEBAS

A continuación se presentan los resultados obtenidos por el DSL interno propuesto en la sección 1.3. Inicialmente se describe el entorno de pruebas en el que se validará el lenguaje, especificando las variables que se comprobarán y detallando la manera como se obtendrá dicha información. Posteriormente se describe la planificación y ejecución de los casos de prueba para finalmente, exponer los resultados obtenidos en las pruebas ejecutadas.

7.1. Descripción del entorno de pruebas

Para validar el DSL desarrollado se procederá a la planificación y ejecución de diferentes pruebas que medirán la sintaxis y el correcto modelado de un problema lineal. Se usará un formato construido por los desarrolladores del proyecto de grado, basándose en formatos usados por ellos mismos en sus experiencias laborales, con el objetivo de facilitar la comprensión y claridad de los casos de prueba estipulados. Estos formatos serán completados por los desarrolladores del DSL y probados por ellos mismos y por otros estudiantes de la Universidad Tecnológica de Pereira.

A continuación se puede detallar la tabla con el formato mencionado anteriormente y la explicación de cada uno de los ítems que lo conforman:

Tabla 23. Formato para los casos de prueba

# caso de prueba: Es un consecutivo, mediante este ítem se identifica el caso de prueba de manera única.	**Fecha de creación**: Fecha en que se planificó el caso de prueba
Nombre caso de prueba: Nombre que describe en pocas palabras en qué consiste el caso de prueba	
Datos de entrada: Los datos que van a ser evaluados en la ejecución del caso de prueba	
Descripción: Descripción más detallada del caso de prueba	
Criterios de evaluación: Parámetros que determinan si el resultado del caso de prueba es satisfactorio o no.	

Fuente: Los autores.

Así mismo, para analizar los resultados de manera clara se usará el siguiente formato expresado en la Tabla 24, el cual también fue elaborado por los desarrolladores del proyecto de grado:

Tabla 24. Formato para los resultados de los casis de prueba

# caso de prueba: Identificador del caso de prueba que fue evaluado	**Fecha de ejecución**: Fecha en que se ejecutó el caso de prueba
Tester: Persona que ejecutó la prueba	
Desarrollo de la prueba: Descripción de cómo fue el procedimiento para desarrollar el caso de prueba	
Salida esperada: Expresa la salida esperada una vez haya completado todo el caso de prueba	
Salida obtenida: Expresa la salida real obtenida una vez completó todo el caso de prueba	
Resultados: Indica el resultado final de analizar las salida esperada y la obtenida, generalmente con un Correcto/Fallido	
Observaciones: Comentarios adicionales sobre algún suceso ocurrido durante la ejecución del caso de prueba, o también sugerencias del mismo.	

Fuente: Los autores.

Las pruebas serán diseñadas, planificadas y ejecutadas en los equipos descritos en la sección de recursos físicos 1.9.2 y otros equipos de características similares de estudiantes de la Universidad Tecnológica de Pereira.

7.2. Desarrollo de los casos de prueba

Tabla 25. Desarrollo caso de prueba 1

# caso de prueba: 1	**Fecha de creación**: 02/05/2015
Nombre caso de prueba: Definición de variables	
Datos de entrada: Las variables del problema	
Descripción: Este caso de prueba tiene como propósito comprobar que las variables de decisión definidas por el usuario cumplen con un estándar de nomenclatura definido previamente.	
Criterios de evaluación: Se considera correcta una variable de decisión definida por el usuario, siempre que cumpla con la siguiente expresión regular: [a-z|A-Z]([a-z|A-Z]|[0-9]|_)* Esta expresión regular indica que toda variable debe comenzar con una letra, seguida de una combinación de letras y dígitos y el carácter '_' únicamente.	

Fuente: Los autores.

Tabla 26. Desarrollo caso de prueba 2

# caso de prueba: 2	**Fecha de creación**: 02/05/2015
Nombre caso de prueba: Variables presentes en la función objetivo	
Datos de entrada: La función objetivo y las variables de decisión	
Descripción: Este caso de prueba tiene como propósito verificar que todas las variables de decisión estén presentes en la función objetivo, y que esta a su vez no contenga ninguna variable no declarada en las variables de decisión.	
Criterios de evaluación: Se considera correcta una función objetivo en la que todas las variables se encuentren también en las variables de decisión, que no haya ni más ni menos variables declaradas en las variables de decisión ni en la función objetivo. Esto quiere decir que haya igual cantidad de variables de decisión que variables de la función objetivo	

Fuente: Los autores.

Tabla 27. Desarrollo caso de prueba 3

# caso de prueba: 3	**Fecha de creación**: 02/05/2015
Nombre caso de prueba: La función objetivo es lineal	
Datos de entrada: La función objetivo	
Descripción: Este caso de prueba tiene como propósito verificar que la función objetivo es lineal.	
Criterios de evaluación: Se considera correcta una función objetivo que cumpla con la definición formal de una ecuación lineal: $\sum_{i=1}^{n} k_i x_i$ Donde: k_i: Costos de utilidad x_i: Variables de decisión	

Fuente: Los autores.

Tabla 28. Desarrollo caso de prueba 4

# caso de prueba: 4	**Fecha de creación**: 04/05/2015
Nombre caso de prueba: Las restricciones son lineales	
Datos de entrada: Restricciones	
Descripción: Este caso de prueba tiene como propósito verificar que las restricciones son lineales.	
Criterios de evaluación: Se considera correcta una restricción que cumpla con la definición formal de una restricción lineal, en cualquiera de sus diferentes alternativas: $\sum_{i=1}^{n} a_i x_i \leq b_i, \quad \sum_{i=1}^{n} a_i x_i \geq b_i \quad o \quad \sum_{i=1}^{n} a_i x_i = b_i$ Donde: a_i= Es el coeficiente i-ésimo técnico conocido que acompaña la variable de decisión x_i. $x_i =$ Es la variable de decisión i-ésima que se desea optimizar. b_i = El valor límite de la restricción, el cual debe ser respetado para que los valores de las variables de decisión sean factibles.	

Fuente: Los autores.

Tabla 29. Desarrollo caso de prueba 5

# caso de prueba: 5	**Fecha de creación**: 04/05/2015
Nombre caso de prueba: Validar variables de decisión en las restricciones	
Datos de entrada: Restricciones y variables de decisión	
Descripción: Este caso de prueba tiene como propósito verificar que todas las variables presentes en las restricciones estén declaradas en las variables de decisión.	
Criterios de evaluación: Se considera correcta una restricción en la cual todas las variables de decisión presentes en la desigualdad, estén declaradas en las variables decisión.	

Fuente: Los autores.

Tabla 30. Desarrollo caso de prueba 6

# caso de prueba: 6	**Fecha de creación**: 04/05/2015
Nombre caso de prueba: El modelo está completo	
Datos de entrada: Variables de decisión, función objetivo y restricciones	
Descripción: Este caso de prueba tiene como propósito verificar que se instancia cada uno de los elementos necesarios para crear un modelo de programación lineal.	
Criterios de evaluación: Se considera exitoso este caso de prueba si cuando un usuario deja de instanciar alguno de los 3 elementos (Variables de decisión, restricciones y función objetivo) aparece un error y obliga al usuario a que rectifique el modelo.	

Fuente: Los autores.

Tabla 31. Desarrollo caso de prueba 7

# caso de prueba: 7	**Fecha de creación**: 04/05/2015
Nombre caso de prueba: Comprobación de sintaxis	
Datos de entrada: Variables de decisión, función objetivo y restricciones	

Descripción: Este caso de prueba tiene como propósito verificar que el lenguaje hace un análisis correcto de la sintaxis, informando cuando se presenta algún error oportunamente.
Criterios de evaluación: Se considera exitoso este caso de prueba si cuando un usuario realiza algún error sintáctico el lenguaje detecta este error y lo muestra por consola.

Fuente: Los autores.

Tabla 32. Desarrollo caso de prueba 8

# caso de prueba: 8	**Fecha de creación**: 04/05/2015
Nombre caso de prueba: No se repiten instancias	
Datos de entrada: Variables de decisión, función objetivo y restricciones	
Descripción: Este caso de prueba tiene como propósito verificar que el usuario no instancie más de una vez alguno de los 3 elementos básicos del modelo.	
Criterios de evaluación: Se considera exitoso este caso de prueba si el DSL detecta cuando un usuario instancia más de una vez un elemento (Variable, Restricciones o Función objetivo)	

Fuente: Los autores.

7.3. Resultados de los casos de prueba

A continuación se exponen los resultados obtenidos en la ejecución de los casos de prueba descritos en la sección 7.2.

Tabla 33. Resultado caso de prueba 1

# caso de prueba: 1	**Fecha de ejecución**: 16/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: La prueba se realizó de 3 formas diferentes. En la primera, se ingresaron 2 variables de decisión (‘x1-3’, ‘2y’) que como se puede apreciar están mal definidas. En la segunda se ingresaron 2 variables de decisión (‘x_1’,’x-2’), la primera bien definida y la segunda no. En el último caso se ingresaron 2 variables de decisión (‘mi_variable_1’,’miVariable2’), todas bien definidas.	
Salida esperada: Mensaje en la consola del error en la definición de las variables de decisión para cada una de las variables mal definidas. El mensaje debe ser el siguiente: “Variable … debe empezar con una letra y solo puede contener letras, números y el símbolo _”	
Salida obtenida: La salida obtenida fue la esperada para cada una de las 3 formas que se desarrolló de este caso de prueba. En la primera forma la salida fue la siguiente: “Variable x1-3 debe empezar con una letra y solo puede contener letras, números y el símbolo _” En la segunda forma la salida fue la siguiente: “Variable 2y debe empezar con una letra y solo puede contener letras, números y el símbolo _” En cambio la tercera salida no tuvo ningún inconveniente y no mostró ningún mensaje de error.	
Resultados: Teniendo en cuenta la salida obtenida en este caso de prueba, se puede determinar que fue exitoso.	
Observaciones: No hay ninguna observación.	

Fuente: Los autores.

Tabla 34. Resultado caso de prueba 2

# caso de prueba: 2	**Fecha de ejecución**: 16/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: Este caso de prueba se realizó de 3 formas diferentes: En la primera se ingresaron las siguientes variables: x1, x2, y. La función objetivo fue la siguiente: 34x1 + 20x2 En la segunda se ingresaron las siguientes variables: x1, x2. La función objetivo fue la siguiente: 2/3x1 + 5x2 + x3 En la tercera se ingresaron las siguientes variables: x1, x2. La función objetivo fue la siguiente: x1 + 5x2	
Salida esperada: Se espera que para los primeros dos casos informe de que las variables de decisión instanciadas y las escritas en la función objetivo no coinciden. El mensaje para el caso 1 sería el siguiente: “Variable y no se encuentra en la función objetivo.” Para el segundo caso el mensaje sería el siguiente: “La función objetivo no cumple con la definición matemática de linealidad o o alguna de las variables no esta declarada. Ejemplo: 3x + 2y”	
Salida obtenida: La salida obtenida para ambos casos fue la esperada.	
Resultados: Los resultados finales indican que este caso de prueba fue exitoso.	
Observaciones: No hay observaciones.	

Fuente: Los autores.

Tabla 35. Resultado caso de prueba 3

# caso de prueba: 3	**Fecha de ejecución**: 16/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba:	

Teniendo las variables x1, x2 y x3 se procedió a realizar este caso de prueba de 2 maneras diferentes: La primera corresponde a la siguiente función objetivo: 3x1 + 4x25x3 La segunda corresponde a la siguiente función objetivo: 23x2 + x2 + 19x3
Salida esperada: Se espera que salga un error en la primera función objetivo declarada únicamente, con el siguiente mensaje: “La función objetivo no cumple con la definición matemática de linealidad o o alguna de las variables no esta declarada. Ejemplo: 3x + 2y”
Salida obtenida: La salida obtenida en los dos casos fue la esperada.
Resultados: Este caso de prueba fue exitoso.
Observaciones: No hay observaciones.

Fuente: Los autores.

Tabla 36. Resultado caso de prueba 4

# **caso de prueba**: 4	**Fecha de ejecución**: 16/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: Se procedió a realizar este caso de prueba ingresando las siguientes restricciones: 1. 23x/3y >= 4 2. x*y + 3.5z <= 29 3. y - 4z >= 16x	
Salida esperada: Se espera que salga un mensaje de error para todas las restricciones. El mensaje debe ser el siguiente: “La restricción 1no cumple con la definición matemática de linealidad o alguna de las variables no esta declarada . Ejemplo: 3x + 2y <= 5”	
Salida obtenida: La salida obtenida en todas las restricciones fue la esperada.	
Resultados: Este caso de prueba fue exitoso.	

Observaciones: No hay observaciones.

Fuente: Los autores.

Tabla 37. Resultado caso de prueba 5

# caso de prueba: 5	**Fecha de ejecución**: 16/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: Para este caso de prueba se ingresaron las variables de decisión (x1, y) y las siguientes restricciones: 1. x1 - 2x2 <= 15 2. x2 <= 10 3. -2y + 4x1 >= 2	
Salida esperada: Se espera que salga un mensaje de error en las primeras dos restricciones. El mensaje es el siguiente: “La restriccion 1 no cumple con la definicion matematica de linealidad o alguna de las variables no esta declarada. Ejemplo: 3x + 2y <= 5 ”	
Salida obtenida: La salida obtenida al realizar este caso de prueba fue la esperada, el DSL informó sobre un error en las restricciones 1 y 2.	
Resultados: Los resultados finales para este caso de prueba son satisfactorios	
Observaciones: No hay observaciones.	

Fuente: Los autores.

Tabla 38. Resultado caso de prueba 6

# caso de prueba: 6	**Fecha de ejecución**: 17/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: Se procedió a realizar este caso de prueba de 4 formas diferentes: 1. Se declararon las variables y la función objetivo, pero no las restricciones 2. Se declararon las variables y las restricciones, pero no la función objetivo	

3. Se declararon la función objetivo y las restricciones, pero no las variables 4. Se declararon todos los elementos
Salida esperada: Se espera que la salida para el primero caso sea la siguientes: “Digite las restricciones del problema” Para el segundo caso sea la siguiente: “Digite la funcion objetivo del problema” Para el tercer caso sea la siguiente: “Digite las variables del problema” y para el cuarto caso no haya mensaje de error
Salida obtenida: La salida fue la esperada inicialmente
Resultados: Este caso de prueba se considera que fue exitoso, pues cumplió con el propósito inicial.
Observaciones: No hay observaciones.

Fuente: Los autores.

Tabla 39. Resultado caso de prueba 7

# caso de prueba: 7	**Fecha de ejecución**: 17/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: Se procedió a ingresar 3 problemas diferentes: Problema 1. Falta el caracter ‘ después de instanciar la variable x2 LinearProgramming .variables('x1','x2,'y') .maximizar('34x1 + 20x2') .subjectTo('x + 1/2y >= 100x', '1/3.6x + 1/6y = 80', '4y >= 2') .showModel() .solveModel() Problema 2. Falta el caracter ‘c’ para el método subjectTo.	

```
LinearProgramming
   .variables('x1','x2','y')
   .maximizar('34x1 + 20x2')
   .subjetTo(
     'x + 1/2y >= 100x',
     '1/3.6x + 1/6y = 80',
     '4y >= 2')
   .showModel()
   .solveModel()


Problema 3. El modelo está correcto
LinearProgramming
   .variables('x1','x2','y')
   .maximizar('34x1 + 20x2')
   .subjectTo(
     'x + 1/2y >= 100x',
     '1/3.6x + 1/6y = 80',
     '4y >= 2')
   .showModel()
   .solveModel()
```

Salida esperada:

Se espera que para la salida 1, el compilador del lenguaje anfitrión detecte el error sintáctico. Para la salida 2 se espera el siguiente mensaje:

“Palabra ‘subjetTo’ incorrecta, rectifique por favor”

Y para la salida 3 no es espera ninguna salida de error, puesto que el modelo está correcto.

Salida obtenida:

Para cada uno de los 3 casos la salida fue la esperada.

Resultados:

Teniendo en cuenta que la salida fue la que se esperaba, se considera este caso de prueba exitoso.

Observaciones:

No hay observaciones

Fuente: Los autores.

Tabla 40. Resultado caso de prueba 8

# caso de prueba: 8	**Fecha de ejecución**: 17/05/2015
Tester: Brahyam Pineda Cardona	
Desarrollo de la prueba: Para este caso de prueba, se realizó de 3 formas: 1. Ingresando dos veces las variables de decisión 2. Ingresando dos veces las restricciones 3. Ingresando dos veces la función objetivo	
Salida esperada: Se espera que en cada caso, en la salida se muestren los siguientes mensajes: “Las variables ya fueron asignadas” “Las restricciones ya fueron asignadas” “Las función objetivo ya fueron asignada”	
Salida obtenida: La salida obtenida en cada uno de los casos fue la esperada.	
Resultados: Los resultados para este caso de prueba indican que fue exitoso.	
Observaciones: No hay observaciones	

Fuente: Los autores.

8. CONCLUSIONES

En este proyecto se implementó un DSL interno que permite modelar problemas de programación lineal cuyo funcionamiento puede ser resumido de la siguiente manera:

- Consta de 3 elementos básicos que son las variables de decisión, la función objetivo y las restricciones.
- Estos elementos deben cumplir con la propiedad de linealidad para que sean validados e interpretados correctamente.
- El orden en que se declaran los elementos del modelo es indiferente, siempre que estén los 3 presentes.
- La sintaxis es muy cercana a la notación matemática de la programación lineal.

A partir de este resumen podemos deducir las siguientes conclusiones sobre el desarrollo del proyecto:

- La sintaxis del lenguaje, a pesar de ser bastante clara y manejar un vocabulario muy similar al original del dominio que representa, no puede evitar tener un mínimo de ruido sintáctico, debido a que está construido bajo un lenguaje anfitrión y se limita a utilizar el azúcar sintáctico que éste posee.
- La construcción de DSLs está en auge, especialmente en los últimos años gracias al cada vez más conocido concepto de DSL interno y todas las facilidades que este ofrece.
- El uso del lenguaje de programación Ruby ofrece gran variedad de herramientas que permiten desarrollar DSLs fácilmente.
- La productividad de las empresas aumenta, automatizando procesos con un DSL intuitivo que haga más sencillo el trabajo del experto.
- La programación lineal es una técnica ampliamente usada por las empresas y entidades de diferentes sectores de la sociedad, es por ello que el DSL implementado es necesario, pues este sector requiere herramientas cada vez más usables.
- La metodología creada para desarrollar DSLs define claramente cada aspecto necesario en su construcción y explica el funcionamiento de las herramientas requeridas.

- El desarrollo orientado al dominio mejora la eficiencia en el proceso de desarrollo pues se limita muy bien el alcance del software, eliminando elementos superfluos que pueden hacer más largo, costoso y tedioso el proceso de desarrollo.

- Antes de diseñar la sintaxis del DSL es necesario conocer muy bien el lenguaje anfitrión en el cual se va a desarrollar, pues conociendo las maniobras que se pueden hacer, se expande el campo de posibilidades para crear la sintaxis.

- Es recomendable definir claramente mediante autómatas finitos la gramática del DSL previamente a implementar las validaciones, pues de esta forma la implementación es más rápida, ya que se tiene claridad del alcance que posee la sintaxis.

- El uso de DSLs puede cerrar la brecha en la comunicación entre humanos y máquinas, creando lenguajes cada vez más parecidos al lenguaje natural.

8.1. Trabajo futuro

La elaboración de este proyecto establece la base para desarrollar trabajos que permiten enriquecer las características del lenguaje y también crear su entorno de desarrollo.

El primer trabajo a realizar, es crear las estructuras necesarias para que el lenguaje se pueda comunicar con diferentes solvers que resuelvan el modelo de programación lineal especificado. Dichos solvers pueden ser implementados como módulos del lenguaje o también se pueden crear conexiones con otros ya existentes mediante interfaces. Así mismo, se pretende desarrollar una interfaz gráfica que le permita al usuario visualizar los resultados arrojados por el solver mediante diferentes tipos de representaciones: diagramas, tablas o figuras. Siguiendo esa perspectiva, se plantea también construir un editor de texto que reconozca la sintaxis del DSL interno, brinde ayudas sintácticas, autocompletado de texto y corrección de errores.

Por otro lado, se ampliará el dominio del lenguaje hacia otras ramas de la optimización, como la programación no lineal, la programación cuadrática y la programación entera. Es decir, evolucionar el lenguaje, con el objetivo de que su dominio sea la optimización, siempre pensando en brindar altos niveles de usabilidad e intuitividad al experto del dominio.

9. BIBLIOGRAFÍA

[1] Martin Fowler and Rebecca Parson. Domain-Specific Languages. Addison-Wesley Professional, 2010.

[2] Debasish Ghosh. DSLs in action. Manning Publications Co, 2011.

[3] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[4] David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. O'Reilly, 2008.

[5] Robert Fourer. Algebraic Modeling Languages for Optimization. 1990.

[6] C.A.C. Kuip. Algebraic Languages for Mathematical Programming. European Journal of Operational Research 67 (1993) 25 -51.

[7] Xueyu Chen, Krishnaraj S. Rao, Jufang Yu and Ralph W. Pike. Comparison of gams, ampl, and minos for optimization Louisiana State University - Baton Rouge, AL 70803

[8] Hermann Schichl. THEORETICAL CONCEPTS AND DESIGN OF MODELING LANGUAGES FOR MATHEMATICAL OPTIMIZATION. Institut f¨ur Mathematik der Universit¨at Wien - Strudlhofgasse 4, A-1090 Wien ,2004.

[9] Robert Fourer, David M. Gay and Brian Kernighan. AMPL A Modeling Language for Mathematical Programming. 2003

[10] Michael R. Bussieck & Alex Meeraus. GENERAL ALGEBRAIC MODELING SYSTEM (GAMS).

[11] Francisco Chediak. Investigación De Operaciones. 2004.

[12] Calle Lejdfors. Techniques for implementing embedded domain specific languages in dynamic languages. 2006.

[13] Jeremy Gibbons and Nicolas. WuFolding Domain-Specific Languages: Deep and Shallow Embeddings. Department of Computer Science, University of Oxford.

[14] Diomidis Spinellis. Notable design patterns for domain-specific languages. Department of Information and Communication Systems, University of the Aegean, GR-83 200 Karlovasi, Greece, 14 February 2000.

[15] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., 1977. A Pattern Language. Oxford University Press, Oxford.

[16] Marjan Mernik, Jan Heering and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. ACM, 2005.

[17] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain analysis and reuse environment. Annals of Software Engineering, 5:125–141, 1998.

[18] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures. ACM SIGSOFT Software Engineering Notes, 20(5):27–37, 1995.

[19] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[20] M. Simos and J. Anthony. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In Proceedings of the Fifth International Conference on Software Reuse, pages 94–102. IEEE Computer Society, 1998.

[21] Ankica Barišić, Pedro Monteiro, Vasco Amaral, Miguel Goulão and Miguel Monteiro. Patterns for Evaluating Usability of Domain-Specific Languages.

[22] Sebastian Günther. Agile DSL-Engineering with Patterns in Ruby. Faculty of Computer Science, University of Magdeburg. 2009.

[23] Erich Gamma, Richard Helm, Ralph Johnson, Jhon Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[24] Dean Kelley. Teoría de autómatas y lenguajes formales. Prentice Hall, 1995.

[25] Sebastian Günther. Development of Internal Domain-Specific Languages: Design Principles and Design Patterns, 2011.

[26] Sebastian Günther, Maximilian Haupt and Matthias Splieth. Agile Engineering of Internal Domain-Specific Languages with Dynamic Programming Languages. School of Computer Science University of Magdeburg Germany.

[27] Paolo Perrotta. Metaprogramming Ruby. The Pragmatic Bookshelf, 2010.

[28] Craig Larman. UML Y PATRONES: Introducción al análisis y diseño orientado a objetos. Pearson, 1999.

[29] Martin Fowler y Kendall Scott. UML gota a gota. Pearson, 1999.

[30] KOLMAN, Bernard; BECK, Robert E. Elementary Linear Programming with Applications. Orlando, Florida. Elsevier, 2014.

[31] WILEY, Jhon & Sons. Encyclopedia of Statistical Sciences. 2006.

[32] FERGUSON, Thomas S. Linear Programming, a concise introduction.

[33] CABALLERO, José A.; GROSSMAN, Ignacio E. Una revisión del estado del arte en optimización. Revista Iberoamericana de Automática e Información Industrial. Vol 4, Núm 1, 2007.

[34] FAULIN, Javier; JUAN, Ángel A. Aplicaciones de la programación Lineal.

10. ANEXOS

Código fuente:

- Modelo semántico

```
class Modelo
  attr_accessor :variables, :funcionObjetivo, :restricciones

  def initialize()
    @variables = Hash.new()
    @funcionObjetivo = ''
    @restricciones = Hash.new()
    self
  end

  # En este método se hará la conexión con el solver encargado
  # de solucionar el problema  de programación lineal.
  def resolverModelo()
  end
end
```

- Constructor de expresiones

```
class LinearProgramming
  attr_accessor :modelo

  def self.initialize()
    if @modelo == nil
      @modelo = Modelo.new()
    end
    self
  end

  def self.variables *variables
    # Validación invocación de método repetida
    initialize
    if @modelo.variables.length != 0
      puts "Las variables ya fueron asignadas"
    else
    # Si este método es llamado desde el method_missing,
    # el parámetro variables llega cómo un vector de un vector de Strings
    if variables[0].class.equal?(Array)
      variables[0].each do |variable|
      @modelo.variables["#{variable}"]=nil
  end
  #Si este método es llamado desde el Script, el parámetro variables llega como
un vector
  #de Strings
  else
    variables.each do |variable|
      @modelo.variables["#{variable}"]=nil
    end
  end
  end
  self
 end

 #Función común para max y min, es necesario tener separadas las funciones,
porque
 #cuando se implemente el solver, se le debe dar tratamiento diferente a las
funciones si es
 #de maximización a una de minimización
 def self.fo(function, opc)
    initialize
    if @modelo.funcionObjetivo.length != 0
      puts "La funcion objetivo ya fue asignada"
```

```
    else
    @modelo.funcionObjetivo = function
    end
  self
  self
end

def self.max function = nil
  fo(function, 'max')
end

def self.min function = nil
  fo(function, 'min')
end

def self.subjectTo(*functions)
  initialize
  if @modelo.restricciones.length != 0
    puts "Las restricciones ya fueron planteadas"
  else
    i = 0
    #Si este método es llamado desde el method_missing, el parámetro functions
llega
    #como un vector de un vector de Strings
    if functions[0].class.equal?(Array)
       functions[0].each do |function|
         i = i+1
         @modelo.restricciones["r#{i}"]=function
       end
      #Si este método es llamado desde el Script, el parámetro functions llega como
un
      #vector de Strings
    else
      functions.each do |function|
        i = i+1
        @modelo.restricciones["r#{i}"]=function
      end
    end
  end
  self
end

 #Mediante este método se valida mediante un mensaje que el usuario no escriba
una
```

```
 #función incorrecta, también sirve para asignarle alias a los métodos, ya que
como se
 #trabajan métodos de clase, no es posible utilizar el método alias.
 def self.method_missing(method, *parameters)
   case
     when method.to_s == 'maximizar' || method.to_s == 'maximize' then
max(parameters[0])
     when method.to_s == 'minimizar' || method.to_s == 'minimize' then
min(parameters[0])
     when method.to_s == 'var' then variables(parameters)
     when method.to_s == 'restrictions' || method.to_s == 'sujetoA' || method.to_s ==
'restricciones' then subjectTo(parameters)
     else puts "Palabra '#{method}' incorrecta, rectifique por favor"
   end
   self
 end

 def self.solveModel()

   #VALIDACIONES DE VARIABLES

   #Validacion variables no vacias
   if @modelo.variables.length == 0
     puts "Digite las variables del problema"
   else
     #Validacion variables escritas correctamente
     variablesCorrectas = true

     #Las variables deben seguir el patrón: (caracter alfabetico)(caracter alfabetico |
caracter
     #alfanumérico | _)*
     expresionVariable = /[a-z|A-Z]([a-z|A-Z]|[0-9]|_)*/
     @modelo.variables.keys.each do |var|
     resultadoExpresionVariable = expresionVariable.match(var)
     if resultadoExpresionVariable.to_s != var
       puts "Variable #{var} debe empezar con una letra y solo puede contener
letras, numeros y el simbolo _"
       variablesCorrectas = false
     end
     end
       if variablesCorrectas

         #VALIDACIONES DE FUNCION OBJETIVO

         #Validacion funcion objetivo no vacia
```

```
        if @modelo.funcionObjetivo.length == 0
          puts "Digite funcion objetivo"
        else
          #Validacion funcion de funcion objetivo escrita correctamenta

          #La función objetivo debe seguir el patrón: -?(constante((*|/)constante)?
variable)
          #((+|-) constante ((*|/)constante)? variable)*
          expresionFuncionObjetivo =
/\s*(\d+(\.\d+)?\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?)?(#{@modelo.variables.keys})+(\s*\+\
s*((\d+(\.\d+)?)\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?)?(#{@modelo.variables.keys})+)*/

          resultadoExpresionFuncionObjetivo =
expresionFuncionObjetivo.match(@modelo.funcionObjetivo)
          if resultadoExpresionFuncionObjetivo.to_s != @modelo.funcionObjetivo
            puts "La funcion objetivo no cumple con la definicion matematica de
linealidad o alguna de las variables no esta declarada. Ejemplo: 3x + 2y"
          else

            #Valida que no halla una variable inmediatamente después de otra, ya que
            #mediante la anterior expresión regular no se puede saber esto,
pues
            #cuando valida las variables: (#{@modelo.variables.keys})+ es necesario
ponerle
            #cerradura positiva porque sino solo reconoce el primer carácter de la
variable que
            #encuentre(esto da problema cuando la variable es de tamaño >1), con la
cerradura
            #reconoce toda la variable y también puede reconocer todas las variables
que se
            #pongan inmediatamente después

            caracteresAlfabeticos = /[a-z|A-Z]/
            variablesNoSeguidasEnFuncionObjetivo = true
            @modelo.variables.keys.each do |var|
              #Mediante la siguiente expresión se identifica la posición donde se
encuentra el
              #primer carácter de la variable en la funcion objetivo
              posicionDePrimerCaracterDeVariableEnFO = /#{var}+/ =~
@modelo.funcionObjetivo
              esCaracterAlfabetico =
caracteresAlfabeticos.match(@modelo.funcionObjetivo[posicionDePrimerCaracter
DeVariableEnFO+var.length].to_s)
              if esCaracterAlfabetico
```

```
            puts "La funcion objetivo no cumple con la definicion matematica de
linealidad o alguna de las variables no esta declarada. Ejemplo: 3x + 2y"
            variablesNoSeguidasEnFuncionObjetivo = false
          else
          end
        end

        if variablesNoSeguidasEnFuncionObjetivo
        #Validacion de que todas las variables estén en la funcion objetivo
        variablesEnFuncionObjetivo = true
        @modelo.variables.keys.each do |var|
          if /#{var}+/.match(@modelo.funcionObjetivo) == nil
            puts "Variable #{var} no se encuentra en la funcion objetivo"
            variablesEnFuncionObjetivo = false
          end
        end

        if variablesEnFuncionObjetivo
        #VALIDACIONES DE RESTRICCIONES

        #Validacion restricciones no vacias
        if @modelo.restricciones.length == 0
          puts "Digite las restricciones del problema"
        else
          #Validacion restricciones escritas correctamente
          restriccionesEscritasCorrectamente = true

          #Debe seguir el patrón: -?(constante((*|/)constante)? variable) ((+|-)
constante
          #((*|/)constante)? variable)* (=|>=|<=) -? constante((*|/)constante)?
variable
          expresionFuncionRestriccion = /\s*-
?\s*(\d+(\.\d+)?\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?\s*)?(#{@modelo.variables.keys})+(\s
*(\+|-
)\s*((\d+(\.\d+)?)\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?\s*)?(#{@modelo.variables.keys})+)*
\s*(=|>=|<=)\s*-
?\s*(\d+(\.\d+)?\s*((\*|\/)\s*\d+(\.\d+)?)?\s*(\*)?\s*)?(#{@modelo.variables.keys}+)?/

          @modelo.restricciones.values.each do |restriction|
            resultadoExpresionRestriccion =
expresionFuncionRestriccion.match(restriction.to_s)
            if resultadoExpresionRestriccion.to_s != restriction.to_s
              puts "La restriccion #{restriction} no cumple con la definicion
matematica de linealidad. Ejemplo: 3x + 2y <= 5 o alguna de las variables no esta
declarada."
```

```
                restriccionesEscritasCorrectamente = false
                else

              if restriccionesEscritasCorrectamente
              #Valida que no halla una variable inmediatamente después de otra, es
necesario
              #hacer esta validación porque la expresión regular:
expresionFuncionRestriccion
              #no puede reconocer cuando acaba una variable y empieza la otra
              @modelo.variables.keys.each do |var|
                posicionDePrimerCaracterDeVariableEnRestriccion = /#{var}+/ =~
restriction
                #Se valida que la variable esté dentro de la restricción
                if posicionDePrimerCaracterDeVariableEnRestriccion
                #Se valida que inmediatamente después de la variable no halla una
letra
                esCaracterAlfabetico =
caracteresAlfabeticos.match(restriction[posicionDePrimerCaracterDeVariableEnRe
striccion+var.length].to_s)
                #Se hay una letra inmediatamente después de la variable debe salir el
mensaje
                #de error
                if esCaracterAlfabetico
                  puts "La restriccion #{restriction} no cumple con la definicion
matematica de linealidad. Ejemplo: 3x + 2y <= 5 o alguna de las variables no esta
declarada."
                  restriccionesEscritasCorrectamente = false
                end
                end
              end
              end
              end
            end
            if restriccionesEscritasCorrectamente
              #SE EJECUTA EL SOLVER CUANDO NO HAY ERRORES
SINTÁCTICOS
              @modelo.resolverModelo()
            end
          end
          end
          end
          end
        end
      end
    end
```

```
    self
  end
end
```