



UNIVERSIDAD SIMÓN BOLIVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERIA DE LA COMPUTACIÓN

**LENGUAJE DE PROGRAMACIÓN ORIENTADO A OBJETOS PARA LA
COMPOSICIÓN TIPOGRÁFICA**

Por:

David Alejandro Lilue Borrero

PROYECTO DE GRADO

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

Sartenejas, Abril de 2016



UNIVERSIDAD SIMÓN BOLIVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERIA DE LA COMPUTACIÓN

**LENGUAJE DE PROGRAMACIÓN ORIENTADO A OBJETOS PARA LA
COMPOSICIÓN TIPOGRÁFICA**

Por:

David Alejandro Lilue Borrero

Realizado con la asesoría de:

Ricardo Monascal

PROYECTO DE GRADO

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

Sartenejas, Abril de 2016



UNIVERSIDAD SIMÓN BOLÍVAR
VICERRECTORADO ACADÉMICO
DECANATO DE ESTUDIOS PROFESIONALES
Coordinación de Ingeniería de la Computación



UNIVERSIDAD SIMÓN BOLÍVAR
VICERRECTORADO ACADÉMICO
DECANATO DE ESTUDIOS PROFESIONALES
Coordinación de Ingeniería de la Computación

ACTA DE EVALUACION DE PROYECTO DE GRADO

Código de la asignatura: EP5408

Fecha: 31-03-2016

Nombre del estudiante: David Alejandro Lilue Borrero

Carnet: 09-10444

Título del proyecto: Lenguaje de programación orientado a objetos para composición tipográfica.

Tutor: Prof. Ricardo Rafael Monascal Caso

Jurados: Marlene Goncalves Da Silva y Judith Cardinale

APROBADO

REPROBADO

Observaciones:

El jurado examinador, **por unanimidad**, considera el proyecto de grado merecedor de la mención especial SOBRESALIENTE:

SI

NO

En caso afirmativo, justifique su decisión en estricto cumplimiento del documento "Criterios para otorgar la mención especial en proyectos de grado y pasantías largas e intermedias" (ver al dorso):

Prof. Marlene Goncalves
Jurado
C.I 11.899.145



Prof. Judith Cardinale
Jurado
C.I 9602241

Prof. Ricardo Monascal
Tutor Académico
C.I 1797731

Nota: Colocar los sellos de los respectivos Departamentos Académicos. Para jurados externos usar el sello de la Coordinación Docente. Este documento debe entregarse sin enmiendas.

RESUMEN

La composición tipográfica se ha convertido en una necesidad al momento de realizar una publicación técnica y científica. Hoy en día existen herramientas que facilitan la elaboración de documentos de gran complejidad. Durante la década de los 70' se creó el lenguaje *troff*, inspirado por *runoff*, el primer programa para el formato de texto, desarrollado a mediados de los 60'. *troff* posee la cualidad de darle formato al texto, así como el manejo arbitrario de elementos tipográficos. Por otro lado, el lenguaje para la composición tipográfica comúnmente usado, en el ámbito académico, es *LATEX*, pero existen otros lenguajes de *markup* destinados a otros objetivos.

Un aspecto importante a considerar cuando se elabora un lenguaje de composición tipográfica es, a quién está destinado; debe tomarse en cuenta si está dirigido a una comunidad científica, la comunidad literaria o una comunidad orientada a la administración y la gerencia. También es posible que se desee una herramienta de propósito general, que se ajuste a las necesidades de cualquier comunidad. Lenguajes de *markup*, como *markdown* y *LATEX*, son usados en distintas áreas para un mismo objetivo, generar un documento; uno trata de simplificar y el otro trata de generar un documento enfocado en la belleza, respectivamente. En este trabajo se desea la posibilidad de generar documentos similares a los generados por *LATEX*, a través del diseño de un lenguaje que sea sencillo, que mantenga un nivel aceptable de flexibilidad. Dándole un punto de vista más estructural y abstracto, haciendo uso de la orientación a objetos.

Palabras clave: composición tipográfica, orientación a objetos, elementos tipográficos, formato de texto.

V

Para todos esos artistas que dejaron su obra,
sin su nombre y aun así trascendieron.

Agradezco a un perro de papel
que algunos dicen que está perdido,
más no quiere ser encontrado.

Índice general

Resumen	IV
Dedicatoria	V
Agradecimientos	VI
Índice General	X
Índice de Figuras	XII
Índice de Tablas	XIII
1. Introducción	1
2. Marco Teórico	3
2.1. Composición Tipográfica	3
2.2. Programación orientada a objetos	5
2.3. Lenguajes de Marcado	7
2.3.1. Lenguajes Ligeros de Marcado	8
2.4. Serialización de datos	8
2.5. Lenguajes de dominio específico	9
3. Marco Tecnológico	11
3.1. T _E X/L ^A T _E X	12
3.2. Ruby	13

3.3. Racc	14
3.4. Git	14
3.5. PDF	15
3.6. TeX Live	15
3.7. MiKTeX	15
3.8. Shell Script	16
4. Desarrollo	17
4.1. OhTeX como un lenguaje embebido	19
4.2. Herramientas de composición tipográfica	19
4.3. Entrada y Salida	20
4.4. Clases e Instancias	20
4.4.1. Instancias	20
4.5. Inicialización y Asignación	21
4.6. Delimitadores y Secuenciación	22
4.7. Ejecución	23
4.8. Implementación	23
4.8.1. Análisis Lexicográfico	24
4.8.2. Análisis Sintáctico	24
4.8.3. Librerías	25
5. OhTeX	26
5.1. Estructura de un archivo OhTeX	27
5.2. Ejecución	28
5.2.1. Línea de comandos	28
5.3. Paquetes e Inclusiones	29
5.3.1. Paquetes	29
5.3.2. Inclusiones	30
5.4. Lexicografía	30

5.4.1. Identificadores	30
5.4.2. Comentarios	31
5.4.3. Palabras reservadas	32
5.4.4. Expresiones	32
5.5. Variables y Literales	32
5.5.1. Variables globales	33
5.5.2. Variables locales	33
5.5.3. Variables de instancia	33
5.5.4. Variables de clase	34
5.5.5. Variables y constantes predefinidas	34
5.6. Formato de texto	34
5.6.1. Tamaño de fuente	35
5.7. Familia de fuente	35
5.7.1. Cursiva	36
5.7.2. Negrita	37
5.8. Literales	37
5.8.1. Números	37
5.8.2. Strings	38
5.8.3. Arreglos	38
5.8.4. Tablas de Hash	39
5.8.5. Rangos	40
5.8.6. Unidades de tamaño y formato de texto	40
5.8.7. Verbatim	41
5.9. Operadores	43
5.9.1. Asignación	43
5.9.2. Lógicos	44
5.10. Estructuras de Control	44
5.10.1. Condicionales	44

ÍNDICE GENERAL	x
5.10.2. Bucles	46
5.11. Instrucciones	47
5.12. Clases	48
5.12.1. Definición de clase	48
5.12.2. Atributos	48
5.12.3. Instanciación	49
5.12.4. Herencia	50
5.13. Estructuración del documento	51
6. Conclusiones y Recomendaciones	52
Glosario de términos	54
Apéndice A. Ejemplo	56
Bibliografía	58

Índice de Figuras

3.1. Estructura de un documento L ^A T _E X	13
5.1. Programa básico: O ^A T _E X	27
5.2. Uso de paquetes	29
5.3. Uso de paquetes (<i>inline</i>)	30
5.4. Nombres de identificadores	31
5.5. Comentarios de línea	31
5.6. Comentarios de bloque	32
5.7. Variables globales	33
5.8. Variables de instancia	33
5.9. Variables de Clase	34
5.10. Familias de fuente y fuentes	36
5.11. Numeros	37
5.12. String e interpolación	38
5.13. Arreglos	39
5.14. Hash (diccionario)	39
5.15. Rangos	40
5.16. Unidades de tamaño y tamaño de fuente	41
5.17. Verbatim Ruby	41
5.18. Verbatim L ^A T _E X	42
5.19. Asignación	43
5.20. Auto-asiganación	43
5.21. Estructura de control: Condicional <i>if</i>	45

5.22. Estructura de control: Condicional <i>if-elsif-else</i>	45
5.23. Estructura de control: Condicional <i>case-when</i>	45
5.24. Estructura de control: Bucle <i>while</i>	46
5.25. Estructura de control: Bucle <i>for</i>	46
5.26. Estructura de control: Bucle en <i>Hash</i>	47
5.27. Declaración de instrucción	47
5.28. Declaración de clase	48
5.29. Declaración de variables de instancia	49
5.30. Declaración de variables de clase	49
5.31. Instanciación	50
5.32. Inicialización	50
5.33. Herencia de clases	50
5.34. Seccionamiento de un documento	51

Índice de Tablas

4.1. Herramientas en distintos ambientes	23
5.1. Comando de consola	28
5.2. Palabras reservadas	32
5.3. Tamaño de fuente	35
5.4. Familias de fuente	36
5.5. Unidades de tamaño	40

Capítulo 1

Introducción

En este trabajo se expone el proceso evolutivo de la composición tipográfica, así como sus componentes. Además se muestra el desarrollo y especificación del lenguaje resultante de todo el proceso investigativo. Hoy en día, el lenguaje más usado para la generación de documentos de índole académica y de investigación, es L^AT_EX [1]; hasta el momento tiene 30 años siendo el mejor y posiblemente le queden varios. La intención de este trabajo no es crear un nuevo sistema de composición tipográfica, porque T_EX/L^AT_EX lo hacen muy bien; el propósito es demostrar un concepto: que es posible elevar el nivel de abstracción sin perder la parte esencial y flexible de T_EX/L^AT_EX.

Con el uso de un sistema de objetos, se abstraerán los elementos de un documento a tipos de datos abstractos con atributos que le den formato al texto, ademas de ilustrar una estructuración más perceptible y fácil de manejar, sin la necesidad de marcas (*tags*); las cuales requieren tiempo para escribir y dificultan el aprendizaje del lenguaje. En cambio, se propone marcar contenido por medio de clases, atributos y seccionando el documento a través de jerarquía; en vez de elementos ordenados secuencialmente.

Es bien sabido que T_EX [2] es difícil de aprender, leer y escribir. Esto se debe a que su creador, *Donald Knuth*, enfocó el diseño del mismo a la generación de documentos académicos de alta calidad, dejando a un lado la expresividad y la facilidad de aprendizaje; aún cuando uno

de los objetivos del lenguaje es que sea utilizado por todos. Posteriormente, *Leslie Lamport* creó un macro paquete, \LaTeX , para facilitar el uso del lenguaje incorporando un marcado descriptivo similar a *HTML*.

El uso conjunto de la orientación a objetos y marcado de texto ha sido implementado en el lenguaje *Curl* [3] desarrollado por el *MIT*, aunque está dirigido a las aplicaciones *web*. Tomando en cuenta el nivel de abstracción, expresividad, facilidad de aprendizaje, manejo de elementos tipográficos se desarrolló la herramienta \O\TeX , un lenguaje de dominio específico que se apoya en *Ruby* [4] para incorporar objetos, estructuras de control, a cualidades de \LaTeX y lenguajes de marcado, como composición tipográfica, formato de texto, estructuración de documentos; sin perder considerablemente flexibilidad.

En primera instancia, este libro expone brevemente un marco histórico que presenta el origen de la composición tipográfica como herramienta usada para la creación de documentos. Además, se presenta un marco teórico donde se abarcan los conceptos relevantes para la compresión del libro. Posteriormente, se exponen las herramientas que apoyaron el desarrollo del lenguaje, así como una justificación de su escogencia. El siguiente capítulo presenta el proceso de desarrollo y decisiones tomados en cuanto al diseño del lenguaje, dejando el capítulo posterior para la especificación del mismo. Y finalmente quedarían las conclusiones, recomendaciones.

Capítulo 2

Marco Teórico

A lo largo de este capítulo se expondrán los distintos conceptos que fundamentan este trabajo, es decir, los conceptos, modelos y notaciones que se tomaron en consideración a lo largo del desarrollo, diseño y especificación del lenguaje producto de este proyecto. Dos conceptos relevantes en el desarrollo del lenguaje son la composición tipográfica y la programación orientada a objetos, dado que son los dos sistemas que se desean relacionar. Además, se enuncian los lenguajes de marcado, los cuales están fuertemente vinculados a la composición tipográfica, junto al formato del texto. Un aspecto importante a tomar en cuenta es la serialización de datos, que formalmente no causa un gran impacto en el trabajo pero está relacionado con los lenguajes de marcado y se incorporan ciertos conceptos y notaciones del mismo. Por último, se ilustra qué es un lenguaje de dominio específico, pues este trabajo intenta resolver un problema preciso y como resultado, se implementa un DSL.

2.1. Composición Tipográfica

La composición tipográfica se define como la organización de caracteres de imprenta (*types*) [5] o su equivalente en digital. Se componen letras, símbolos y glifos acorde a la ortografía de un lenguaje, junto a la gramática y al final, obtener la visualización de un texto.

Este término se maneja desde alrededores del año 1040 [6], empleando piezas móviles hechas de cerámica en China. Luego en 1377, se usan *types* metálicos inventados en Korea [7], durante la dinastía *Koryo* y usados más que todo para la impresión de documentos budistas, *Jikji* (el libro más viejo impreso con *types* metálicos) pero ya se había desarrollado esa tecnología en 1234. Posteriormente, llegó a occidente alrededor del 1440 y era usada principalmente para cartas [8].

Durante el proceso, cada *type* era una pieza metálica (o de cerámica) con un carácter en relieve sobre uno de los lados. Estos se colocaban a mano (uno a uno) para crear palabras, luego oraciones, párrafos, páginas y al final, se imprimía una página colocando tinta encima del conjunto de *types* y presionando el papel sobre el mismo. Nótese que todos los caracteres en los *types* debían estar invertidos para que en el papel se imprimiese bien orientado.

En el año 1884, *Ottmar Mergenthaler*; quien fundó *The Mergenthaler Linotype Company*, desarrolló una máquina llamada máquina linotipia (*Linotype Machine*) [9] destinada para la imprenta, facilitando la alineación de *types* de una manera rápida y sencilla. Para alrededores de 1914, ya tenía un competidor: *The Intertype Company* que implementó el mismo concepto, usando otros materiales en su máquina (acero y aluminio).

Para el siglo 19, se había globalizado e industrializado esta forma de impresión, más que todo por imprentas y editoriales. Pero esto fue hasta los 70' y 80', dado que fueron sustituidas por una nueva tecnología llamada foto-composición tipográfica (*Phototypesetting*), que computarizó este concepto; incorporaron la computadora de escritorio junto al concepto de composición tipográfica y la fotografía para crear documentos. Su mayor impacto, fue ofrecer la posibilidad usar estas máquinas en oficinas, en caso contrario a sus predecesoras.

El proceso era más simple, se imprimían caracteres en un negativo o un papel sensible a la luz para su posterior impresión. También se incorporó el uso de programas más complejos para la composición tipográfica electrónica como *troff* [10], que generaban la entrada de un fotocomponedor tipográfico *phototypesetter* (*Graphic System CAT*) [11].

A pesar de su innovación, se convirtió rápidamente en un método obsoleto. Gracias a que se hizo posible generar una salida en tiempo real de lo que se iba componiendo, a través de una pantalla que imprimía una imagen digital; para ese momento la tecnología usada para la visualización

de imágenes digitales era por medio de un tubo de rayos catódicos [12]. Y posteriormente, en 1985 se podía hacer uso de programas destinados a la publicación de escritorio (*Desktop Publishing /DTP*), combinándolo con *PostScript*.

De manera independiente, en el año 1978 aparece \TeX , un lenguaje para la composición tipográfica para que cualquiera, con un esfuerzo razonable, pueda generar un documento o libros. Una de las principales motivaciones era la portabilidad, asegurando que la herramienta funcionara en cualquier computador. Éste desplazó ampliamente a *troff*, que había mejorado al incorporar otras salidas que no fuesen un *phototypesetter*, como impresoras y *PostScript*, pero las capacidades del mismo fueron superadas. Más tarde, en el año 1985, aparece un macro paquete, \LaTeX para reducir la dificultad que presentaba aprender \TeX .

Cabe destacar que herramientas de *Desktop Publishing* es diferente a $\text{\TeX}/\text{\LaTeX}$ y *troff*, dado que estas últimas no brindan una interfaz gráfica con una filosofía *WYSIWYG* (*What-you-see-is-what-you-get*).

2.2. Programación orientada a objetos

La orientación a objetos es un paradigma basado en objetos, estructuras de datos que contienen atributos y bloques de código, conocidos como métodos [13]. Una característica distintiva dentro de este paradigma es que un método de un objeto puede acceder y/o modificar atributos del objeto mismo, es decir, referenciarse a si mismo usando la notación de *self* o *this*.

La idea de este concepto es elevar el nivel de abstracción al momento de representar problemas de la vida real en algoritmos y estructuras de datos. Esto permite al programador enfocarse en el problema, más que en su representación. Muchos lenguajes incorporan ciertas características de este paradigma a su diseño y otros un sistema de objetos puro, donde todo el lenguaje es un objeto y todo objeto es una instancia de una clase.

En este paradigma hay dos conceptos esenciales, las clases; que es la definición de un conjunto de datos y comportamientos, y los objetos; que son las instancias de las clases. Los mismos

pueden tener distintos miembros, ya sean métodos, atributos, campos, constructores, propiedades. Dependiendo del lenguaje, estos términos son usados de diferente manera. En algunos casos, los atributos se refieren de igual manera a los campos y fundamentalmente son variables asociadas a una clase o instancia. Los constructores y propiedades pudiesen ser métodos de una clase o instancia. A pesar de esto, existen conceptos básicos y esenciales en cuanto a la información que contienen las clases e instancias:

- Variables de clase. Pertenece a la clase, es decir, que solo existe una, compartida por todas las posibles instancias.
- Variables de instancia. Pertenece individualmente a cada objeto (instancia) de una clase con un valor particular para cada uno.
- Métodos de clase. Pertenece a la clase y por tanto sólo puede utilizar variables de clase y parámetros que sean pasados al método.
- Métodos de instancia. Pertenece individualmente a cada objeto (instancia) de una clase y sólo tiene acceso a las variables desde donde es llamado, así como a los parámetros que se le pasan. Además de variables y métodos de alguna clase.

Un concepto intrínseco en la programación orientada a objetos y de gran importancia en cuanto a la abstracción es el encapsulamiento. Este principio trata sobre ocultar información. Esto es útil porque evita que un programador, usando dicho objeto, se preocupe del funcionamiento del mismo y al momento de refactorizar código [14].

Objetos pueden contener otros objetos de distinta clases dentro de sus atributos y crear una relación entre instancias. Por otro lado, la relación más fuerte entre objeto es la herencia. En muchos casos, cuando un lenguaje tiene clases, tiene herencia. Si se interpretaran las clases como conjuntos, una subclase sería como un subconjunto (que se aumenta potencialmente, con funcionalidades adicionales).

La herencia de clases en los lenguajes orientados a objetos puede ser simple o múltiple, y esa decisión queda a discreción de su creador. La herencia múltiple implica algunos problemas de

alcance, como el problema del diamante [15]. Cada lenguaje que implementa este tipo de herencia, lo resuelve a su propia manera. En el caso de la herencia simple, no ocurre este problema pero el lenguaje “perdería” poder y usan otras estrategias para compensarlo; haciendo uso de mixins, interfaces, clases abstractas, etc.

Los lenguajes que incorporan este concepto a su diseño, lenguajes como Java y C++ usan conceptos de clases, herencia, atributos, métodos, instancias, interfaces, etc; siguen siendo imperativos y otros como Scala, son declarativos. Así, que no van en función del paradigma de programación.

2.3. Lenguajes de Marcado

Estos lenguajes ofrecen un sistema que permite explicar y realizar anotaciones en un documento, sin que se mezcle con el contenido. Para ello, la mayoría de los lenguajes usan marcas (o *tags*). Uno de los más conocido es *HTML*, que es mundialmente usado y es un ejemplo que sigue los estándares generalizados de los lenguajes de marcado, en su mayor parte.

Dentro de estos lenguajes existen distintas formas de marcar el texto y categorías [16]:

- Marcado por presentación: Es usado por sistemas para el procesamiento de texto, siguiendo el paradigma *WYSIWYG*, donde todo el proceso de marcado está oculto para el usuario, es decir, se hace directamente en la máquina.
- Marcado procedural: El marcado está integrado con el texto, mientras provee instrucciones para el procesamiento del texto en el lenguaje. Todo es visible y manipulable por el usuario. Este tipo de marcado ofrece procedimientos, métodos o macros que permiten manipular el texto.
- Marcado descriptivo: Ofrece la posibilidad de indicar cómo debería ser procesado el texto, en vez de hacerlo, separando la estructura del texto, y sea independiente de su interpretación al momento de procesarlo.

Es importante notar que la mayoría de los lenguajes de marcado no se encasillan en alguna categoría específicamente. Por ejemplo, a `TEX`, siendo un lenguaje de marcado procedural, se le dio un enfoque más descriptivo con `LATEX`.

El primer lenguaje que usó este concepto de separar la presentación del texto de su contenido, fue *Scribe*. Ahora, en el caso del marcado descriptivo, la presentación queda por parte de otro lenguaje, programa, etc. *HTML* se usa comúnmente junto a *CSS*, que es una hoja de estilo y describe la presentación de un documento.

2.3.1. Lenguajes Ligeros de Marcado

En los últimos años han surgido lenguajes de marcado más sencillos, fáciles de entender y utilizar en cualquier ambiente. Algunos no usan *tags* explícitamente y se enfocan en el contenido, de forma que el mismo pueda ser leído con facilidad sin pasar por un procesador de texto y obtener una presentación comprensible por un humano. Uno de los lenguajes de marcado más usados es *Markdown* [17], con el cual es fácil de escribir en cualquier editor de texto. Pero una desventaja notable de este tipo de lenguajes, es que no brindan muchas posibilidades para generar documentos, ya que se enfocan principalmente en el contenido.

2.4. Serialización de datos

La serialización es el proceso de traducir estructuras y objetos en un formato específico para que pueda ser transmitido por algún medio y que pueda ser reconstruido independientemente del ambiente o entorno (deserialización).

Es comúnmente usado para transferir datos por redes, entre distintas herramientas, guardar información, etc. Es una manera de guardar información en un formato estándar y que puede ser interpretado dependiendo de quién lea los datos.

En el caso de tipos abstractos de datos que posean información privada o protegida (representación encapsulada), la serialización podría romper el encapsulamiento y revelar su estructura interna.

Existen diversos lenguajes para la transmisión de datos y su serialización, entre los cuales se destacan tres:

- *XML*. Es un lenguaje de marcado para transmitir información entre lenguajes, buscando que la misma sea humanamente leible.
- *JSON*. Es una versión más simple, que almacena la información en objetos, usando una notación de pares; atributo-valor.
- *YAML*. Posee más cualidades, por lo que lo hace más poderoso. Está diseñado para que sea más amigable al momento de leer la información.

2.5. Lenguajes de dominio específico

Los lenguajes de dominio específico son lenguajes de programación diseñados para abarcar una clase de problemas específica. Existen muchos lenguajes que son usados comúnmente, como *awk*, *make*, *HTML*, inclusive *troff*. En contraste a los lenguajes de propósito general, que son aplicables en cualquier área, en el diseño de un lenguaje de dominio específico se debe tomar una decisión en cuanto a si será interno o externo [18]. Los *DSL* interno forman parte de un lenguaje anfitrión para apoyar sus funcionalidades y diseñan un modelo de uso; pudiendo llamarse lenguajes embebidos o ser simplemente una librería de un *GPL*. Los *DSL* externo tienen su propia sintaxis y poseen un *parser* completo, pero mantienen la idea de enfocarse en un problema específico.

Por otro lado, al momento de diseñar un *DSL* existen patrones [19]:

- Una herramienta independiente. Ej. *make*, *grep*.

- Un sistema de macros en el lenguaje anfitrión. Ej. L^AT_EX.
- Un lenguaje embebido o librería. Ej. *OpenGL*.
- Un lenguaje para una funcionalidad de un *GPL*, que pueden ser operaciones en el mismo. Ej. Expresiones regulares.

Este tipo de lenguajes incorporan mayor nivel de abstracción y expresividad en cuanto al problema que abarca. Por otro lado, el costo de aprender un nuevo lenguaje en relación a sus capacidades podría llegar a ser una desventaja. Por otro lado, una implementación inocente, podría comprometer el desempeño pero la eficiencia es considerada una desventaja dependiendo de los requerimientos del lenguaje.

Capítulo 3

Marco Tecnológico

En este capítulo, se describirán las distintas herramientas que se usaron durante el desarrollo del lenguaje. Se consideran los requerimientos del lenguaje, el cuál integra L^AT_EX, y se expone cómo ha logrado ser la herramienta de composición tipográfica más usada en el ámbito académico. La idea del lenguaje es que sea utilizado en distintas plataformas, por lo que se incorporaron herramientas como *TeX Live* y *MikTeX*, para ambientes Linux y Windows, respectivamente.

Tomando en cuenta el paradigma de programación orientado a objetos, previamente definido, el lenguaje incluye a *Ruby* para el manejo de objetos, ya que el lenguaje diseñado está influenciado por éste. Además, la implementación de este paradigma es compleja y requiere un trabajo extenso, por lo que se usó a *Ruby* para solventar esto; cabe destacar que la idea es ilustrar un concepto.

En la implementación de un lenguaje, es implícito el uso de una herramienta generadora de un *parser* y tratando de mantener cierta consistencia, se utilizó *Racc*, el cual está hecho en *Ruby*. En el desarrollo de un software es fundamental el uso de herramientas para el control de versiones, y para ello se usó *Git*, dado que es ampliamente usado y recomendado. Siguiendo la filosofía de que sea una herramienta multiplataforma, se crearon una serie de *scripts* que pudiesen ser ejecutados en una línea de comandos, ya sea en Linux o Windows.

3.1. **TEX/LATEX**

TEX es un sistema destinado a la composición tipográfica, diseñado por *D. Knuth* [20] [21], y ha sido categorizado como la mejor herramienta tipográfica (digital) hasta el momento. La misma es usada comúnmente en el ámbito académico para la generación de libros, artículos y publicaciones, principalmente en el área de matemáticas, ciencias de la computación, ingeniería, física, estadística y, en general, cuando se requiera crear un documento de investigación.

En el momento en que surgió este lenguaje, en 1979, fue la principal competencia de *troff*, el primer sistema de composición tipográfica. Dado que incorporaba aspectos que para el momento se hacían con muchos errores o no se hacían, lo desplazó rápidamente. Además de generar una composición de calidad, usa algoritmos para el espaciado de ecuaciones, justificación de texto y separación de palabras (*hyphenation*), que en su momento fueron características innovadoras y de gran interés; aún hoy lo siguen siendo [22].

TEX trabaja en conjunto con otro lenguaje, *Metafont*, el cual genera los glifos que usa.

El sistema funciona con una serie de instrucciones (*commands*) [20] precedidos por una barra invertida (*backslash*) y agrupaciones entre llaves. La base del lenguaje comprende de alrededor de 300 instrucciones, llamadas *primitives*, que no son usadas por usuarios dado que son instrucciones de bajo nivel. Por lo tanto, se crearon un conjunto de instrucciones basadas en éstas para facilitar el uso del lenguaje.

A pesar de ello, **TEX** es considerado difícil de aprender, contradiciendo el hecho de que es un sistema para que cualquiera persona genere documentos. Por ello se han creado macro paquetes como **LATEX** [1] y **ConTExt** [23]. Además de ello, se han implementado interfaces gráficas que se apoyan en **TEX** y hacen el lenguaje más amigable para el usuario en general.

LATEX es un lenguaje de marcado e incorpora a su lenguaje anfitrión (**TEX**) una serie de instrucciones, ambientes y formatos predefinidos para ayudar a su más fácil entendimiento y reducir la curva de aprendizaje. Usando la misma filosofía de un lenguaje de marcado descriptivo,

separa el contenido de la presentación, enfocándose más en la estructura de un documento. Puede generar distintas salidas, que pueden ser un archivo *DVI*, *PS* o *PDF*, dependiendo de qué se hará con el documento generado.

A continuación, en la figura 3.1, se muestra la estructura básica de un archivo *LATEX*, suponiendo que se usarán dos ambientes, *ambiente1* y *ambiente2*.

Figura 3.1: Estructura de un documento *LATEX*

```
\documentclass[opciones]{clase_del_documento}
% Comandos del preambulo, si se necesitan

\begin{document}
% Texto del documento e instrucciones

\begin{ambiente1}
% Instrucciones y texto del ambiente 1
\end{ambiente1}

% Más texto del documento e instrucciones

\begin{ambiente2}
% Instrucciones y texto del ambiente 2
\end{ambiente2}

\end{document}
```

Es importante notar que *ambiente1* y *ambiente2*, en la figura 3.1, son ejemplos y que se pueden usar más ambientes, incluso anidados, en un documento *LATEX*.

3.2. *Ruby*

Ruby es un lenguaje orientado a objetos, enfocado a la pureza conceptual, dinámico, reflexivo, y de propósito general. Surgió en el año 1995 en Japón, diseñado por *Yukihiro Matsumoto*.

Está influenciado por Perl, Eiffel, Ada, Lisp y principalmente Smalltalk por su sistema de objetos, donde todo elemento dentro del lenguaje es un objeto; inclusive las clases y elementos que otros lenguajes ven como literales (enteros, booleanos, caracteres, etc.).

Es un lenguaje que brinda una sintaxis (similar a Perl), flexible y concisa, por lo tanto el código es limpio, enfocándose en lo que se quiere expresar y brinda la facilidad de crear lenguajes de dominio específico [18]. También, tiene la filosofía de que todo es una expresión y todo es ejecutado de manera imperativa. Por otro lado, tiene capacidad de introspección y reflexividad de objetos, facilitando el uso de meta-programación. Usa el modelo de referencia para las variables, y tiene recolección de basura. Además, incorpora características funcionales, gracias a su flexibilidad.

3.3. Racc

Racc es un generador de *parsers* LALR(1), escrito en *Ruby* y que genera un programa *Ruby*. Esta herramienta toma como entrada una gramática libre de contexto, en notación *Backus-Naur* y crea un *parser* LALR¹, el cual reconoce el lenguaje definido en la gramática. Este tipo de *parser* fue inventado por *Frank DeRemer*, partiendo de los *parsers shift-reduce LR(k) (left-right)* definidos por *Donald Knuth* [24], que eran considerablemente más extensos en cuanto a la memoria, en su época, para su implementación; haciéndolos imprácticos, por lo que fue una alternativa eficiente en cuanto a memoria. El primero y más popular es *Yacc*, y hoy en día muchos de los *parsers* están inspirados en él, como Racc.

3.4. Git

Es una herramienta para el control de versiones de un proyecto, usado en el desarrollo de *software*. Ofrece la posibilidad de crear una bitácora no líneal durante el desarrollo de un proyecto, usada para la colaboración de grupos grandes al mismo tiempo, a través de un sistema de unión

¹Los *parsers* LALR, se refieren a LALR(1)

(*merge*) y derivación (*branch*) del proyecto en un servidor privado o público (como GitHub o Bitbucket). Puede ser usado de forma remota a través de un servidor, o localmente.

3.5. PDF

Es un acrónimo para *Portable Document Format* (“formato de documento portable”). Almacena toda la información de un documento, incluyendo el texto, fuente, imágenes y cualquier elemento que se necesite mostrar. Este formato usa en parte al lenguaje *PostScript* para generar el diseño del documento, además de presentar simplicidad y mejoras al mismo.

3.6. TeX Live

TeX Live es un *software* para adquirir el sistema tipográfico \TeX , junto a los archivos binarios para su uso en ambientes Unix, como GNU/Linux. Incluye gran parte de los programas relacionados con \TeX , macro paquetes y soporte. Además, posee un gestor de paquetes, que mantiene actualizado todos los componentes de \TeX . Ofrece la posibilidad de escoger entre distintos esquemas de instalación, pudiendo seleccionar qué paquetes se desean instalar. La idea de usar este herramienta es apoyar y facilitar el proceso de traducción incorporando un *software* existente.

3.7. MiKTeX

Esta herramienta es muy similar a Tex Live, pero está desarrollada para funcionar sólo en ambiente *Windows*. Una de las funcionalidades más relevantes, que no está presente en TeX Live, es la instalación automática de paquetes no instalados (cuando no es encontrado por \TeX).

3.8. Shell Script

Es un programa diseñado para ejecutarse en un *Shell* de Unix. Sus capacidades más usadas son la manipulación de archivos, ejecución de programas e imprimir texto en la línea de comandos. Es un lenguaje de programación de *scripting*. En caso de que se cree un archivo con un conjunto de instrucciones, irá ejecutando cada una de ellas mientras lee el archivo de arriba a abajo. En el caso de *Windows* existe algo similar a *shellscrip*: los archivos *Batch*, que consisten en un conjunto de instrucciones guardadas en un archivo y son ejecutadas desde la línea de comandos.

Capítulo 4

Desarrollo

En principio se ideó la posibilidad de que el lenguaje fuese no más que la librería de un lenguaje orientado a objetos que tradujese a L^AT_EX, pero esto limitaría las características y posibilidades al momento de generar documentos. Es evidente que el lenguaje limita el poder de L^AT_EX, pero se quiere evitar en lo posible perder su generalidad y tener una interfaz lo suficientemente flexible para abarcar las cualidades necesarias para la generación de documentos.

Se pensó también en implementar una herramienta que tradujese a L^AT_EX directamente, evitando intermediarios, pero se descartó esta posibilidad. Aunque es totalmente factible, y un desligamiento de otro lenguaje representaría una mejora considerable en el rendimiento, sería un trabajo extenso para la demostración de un concepto. La idea es comprobar que es posible reducir la dificultad de aprendizaje sin perder la flexibilidad de manejar documentos, además de los elementos tipográficos, como objetos referenciables, elevando el nivel de abstracción; y no simplemente como elementos ordenados secuencialmente enmarcados.

Existen herramientas que tienen distintos enfoques al momento de la composición tipográfica, y siguen distintas filosofías al momento de procesar texto. Dos de tales enfoques son *WYSIWYG* (“what you see is what you get”) y *WYSIWYM* (“what you see is what you mean”). *LyX* y *TeX/L^AT_EX* usan estas filosofías respectivamente. En este caso no están desligados uno del otro, dado que *LyX* ofrece una interfaz donde muestra lo que sería el documento generado, pero se sigue escribiendo

\LaTeX , siendo simplemente una fachada para interactuar directamente con el contenido sin llegar a la semántica para generar dicho elemento tipográfico.

En este caso, no tiene sentido que nuestro lenguaje haga lo mismo que *LyX*, porque éste lo hace bastante bien y no sería innovador crear otra herramienta que aplique el mismo concepto. Por ello, se decidió incrementar la expresividad de la semántica al momento de estructurar un documento siguiendo la filosofía *WYSIWYM*, facilitando la comprensión de lo que se escribe, abstrayendo los elementos de un documento a objetos y encapsulando el texto con su formato.

La primera decisión tomada, a nivel de diseño, fue que la sintaxis del lenguaje no debe hacer uso de marcas (*tags*) para enmarcar un comportamiento, ambiente o formato de texto. Éste puede ir intrínseco en cada objeto del documento, con ayuda de una estructuración adecuada, símbolos y jerarquía.

Gran parte del lenguaje es inspirado en *Ruby*, y esto se debe a la filosofía que posee el mismo como lenguaje orientado a objetos. Se necesitaba que el lenguaje tuviese un sistema de objetos que fuese puro, es decir, donde todo fuese un objeto y enfocarlo a lo que sería la estructuración de un documento.

Por otra parte, se quería reducir la dificultad de aprendizaje, al momento de entender (leer) y generar (escribir) un documento. Para ello, se tomó en consideración un concepto en computación para el empaquetamiento de información, llamado *serialización*, implementado por varias herramientas, entre los que destaca *JSON*, un lenguaje ampliamente usado. Además de eso, los objetos en el lenguaje deben contener información sobre el estilo o formato (*style*) del mismo. Por este motivo se decidió tomar en consideración a *CSS*.

Juntando estos dos conceptos para encapsular en un objeto la información, texto o contenido junto a su *style*, el seccionamiento del documento es dado por la composición de distintos objetos referenciados.

4.1. O^HT_EX como un lenguaje embebido

Una de la decisiones más relevantes de O^HT_EX es ser usado a través de otro lenguaje, que en este caso es *Ruby*. Aunque es posible implementarlo en otro lenguaje, esta decisión está fundamentada en el hecho de que el modelo de las variables, alcance y las asociaciones son las mismas que en *Ruby*. Pero estas características serán usadas para un propósito específico y así, se reduce el nivel de trabajo en cuanto a la implementación del lenguaje.

A pesar de que el lenguaje posea su propia sintaxis, cada expresión guardaría un símil prácticamente directo a una expresión en *Ruby*. Por lo tanto, O^HT_EX sería un lenguaje de dominio específico embebido (externo) en *Ruby*.

4.2. Herramientas de composición tipográfica

Existen muchos lenguajes para la composición tipográfica, con distintos propósitos, características, flexibilidades, expresividad, áreas de uso, etc. O^HT_EX tiene un propósito académico, es decir, generar documentos académicos y de investigación. Por ello es necesario usar una herramienta que sea diseñada para ese fin. Por lo tanto, L^AT_EX es la opción más adecuada, y además, se estaría elevando el nivel de abstracción de un lenguaje poderoso pero difícil de entender. Evidentemente, esto genera una perdida de flexibilidad en cuanto a las posibilidades de generar un elemento tipográfico, pero se abarcaría gran parte de los más usados y relevantes para la creación de un documento completo.

La decisión de usar T_EX/L^AT_EX también fue tomada debido a que al momento de su diseño, su creador no se enfocó en reducir la dificultad de aprendizaje, sólo en la libertad de manejar elementos tipográficos a discreción y generar documentos centrados en la estética. Sin embargo, todo ese poder no debe implicar una gran complejidad en lo que sería entender el lenguaje.

4.3. Entrada y Salida

En cuanto a la incorporación de instrucciones para el manejo de entrada y salida (*I/O*), éstas no existirían en *O^HT_EX*. La única salida posible del lenguaje será generar un documento haciendo uso de un operador que se especificará en el próximo capítulo. Ésto se debe a que no limita ni brinda poder para el propósito final del lenguaje.

4.4. Clases e Instancias

Tomando en cuenta a *L^AT_EX* y *Ruby*, donde es posible definir una clase de documento (*documentclass*) y una clase (*class*) respectivamente, en *O^HT_EX* también se pueden definir clases que tienen una estructura similar a las de *Ruby*, pero tienen un significado semántico similar a las clases en *L^AT_EX*. En la mayoría de los casos, se usarán las clases para definir una nueva subclase de la clase abstracta *Document*, donde se definirá un formato específico de un documento en particular, que no pueda usar las clases predefinidas. En cualquier otro caso, toda clase que se defina sin un parente, será subclase de la clase mayor *Object*, siguiendo la filosofía de *Ruby*.

En toda clase se pueden definir atributos, que serán siempre públicos, al igual que en todas las clases predefinidas en *O^HT_EX*. Además, se pueden definir instrucciones usando la sintaxis adecuada.

4.4.1. Instancias

Con respecto a la instanciación de una clase, en la mayoría de los lenguajes orientados a objetos se puede notar que existe una palabra reservada o método *new* para la construcción de instancias. En otros casos, la llamada a un método con el mismo nombre de la clase es suficiente. En el caso de *O^HT_EX* se decidió utilizar el símbolo doble dos-puntos (:) para indicar la clase que se está instanciando, junto al identificador que referencia a dicha instancia y a los argumentos que

serán pasados al constructor. Dichos argumentos serán pasados por nombre, siguiendo el concepto de serialización de datos.

4.5. Inicialización y Asignación

Al momento de construir una instancia de clase, los parámetros para inicializarla son pasados de una manera particular, sólo para la construcción. Esto con la intención de aumentar la expresividad al momento de crear un objeto, tanto para quien lo escribe como para quien lo lee. Por ello, se decidió que fuese una inicialización por nombre, seguida de su valor.

Tomando en cuenta los símbolos más usados en los lenguajes para expresar una asignación, sustitución o asociación de valor a un identificador, el símbolo que representaría esta “asignación” o “inicialización” en \LaTeX podía ser uno de los siguientes:

Opción a: (=)

Opción b: (:=)

Opción c: (:)

Opción d: (<-)

Para que al momento de leer la expresión, ésta tuviese un significado completo y concreto, debía considerarse lo siguiente:

- El símbolo se escribiría seguido. Por lo tanto, debía poseer la menor cantidad de caracteres posible, es decir, uno.
- El símbolo se referiría a la asociación de un atributo con un valor.
- El símbolo poseería una semántica acorde a lo que se deseaba expresar.

En primer lugar se descartó al símbolo `{ := }`, debido a lo tedioso de su escritura y a que representa una sustitución textual, lo cual no va acorde a la semántica esperada.

También el símbolo `{ <- }` fue descartado, principalmente porque su significado no se relaciona con lo que busca decir su expresión (que sacará “algo” del lado derecho y lo colocará en el lado izquierdo).

Entre los símbolos restantes, `{ = }` y `{ : }` se optó al final por `{ : }`, tomando en cuenta las siguientes razones:

- Desde un punto de vista lingüístico, representa una pausa para hacer un llamado de atención en lo que sigue y que siempre está relacionado con el texto precedente.
- Cuando se tiene la expresión `«font : 10pt»` por ejemplo, el lado izquierdo habla del lado derecho y eso es lo que quiere expresar.
- Es usado por CSS (lenguaje de *markup*) y JSON (lenguaje de serialización de datos).

Luego, se decidió usar el símbolo `(=)` para la asignación de objetos o valores a una variable (o atributo), a pesar de que el símbolo `(:)` representa la asociación de un atributo con su valor al momento de su construcción.

4.6. Delimitadores y Secuenciación

Para la delimitación de un alcance, donde puede haber una definición de clase, un constructor, una instrucción, o una estructura de control, se contó con tres opciones: indentación, la palabra reservada `“end”` y llaves `({})`. Se escogió esta última por su asociación directa con CSS, es decir, con el formato y estilo del texto.

Por otra parte, se decidió que la secuenciación de instrucciones, expresiones y objetos

sería por medio del salto de línea. Sin embargo, puede usarse el punto-y-coma para el mismo fin, permitiendo así dar la posibilidad de escribir expresiones en una sola línea (*inline*), o en bloque.

4.7. Ejecución

O^HT_EX hace uso de dos *software*, TeX Live y MikTeX para ambientes Linux y ambientes Windows, respectivamente. Integrado a la herramienta con estas aplicaciones se aprovechan la capacidades de las mismas en el manejo de paquetes y uso de L^AT_EX. El uso conjunto de O^HT_EX, Ruby, TeX Live y MikTeX se hace a través de una serie de instrucciones almacenadas y ejecutadas en un archivo *shell script* y un *batch file*, para tener la posibilidad de trabajar en distintos ambientes.

Tabla 4.1: Herramientas en distintos ambientes

Sistema Operativo	Herramientas
Linux	ShellScript O ^H T _E X Ruby L ^A T _E X (TeXLive) PDF
Windows	Batch File O ^H T _E X Ruby L ^A T _E X (MikTeX) PDF

4.8. Implementación

O^HT_EX posee una sintaxis similar a Ruby y existe una correspondencia entre ambos lenguajes. En primer lugar, el proceso de traducción genera un archivo intermedio Ruby y este es el resultado de un análisis lexicográfico y sintáctico de un programa O^HT_EX. Posteriormente existen un conjunto de clases y métodos que son usados por este archivo intermedio para la generación de

archivo \LaTeX . Dicho documento \LaTeX sería el segundo archivo intermedio, el cuál es la entrada del compilador $\{pdflatex\}$ para la generación del documento final en formato PDF .

El proceso de traducción es:

$$\text{\O\kern-1.5pt\TeX} \longrightarrow \text{\textit{Ruby}} \longrightarrow \text{\LaTeX} \longrightarrow \text{PDF}$$

4.8.1. Análisis Lexicográfico

En cuanto al análisis léxico del lenguaje, no se hace uso de herramientas externas a *Ruby*, es implementado desde cero haciendo uso de expresiones regulares; las cuales son inherentes en el lenguaje. En primer lugar se define un *hash* que contiene cada nombre de los *tokens* reconocidos en \O\kern-1.5pt\TeX y cada uno de estos nombres es una llave (*key*) que está vinculado a su expresión regular correspondiente. A través de este *hash*, se va a crear una clase a tiempo de ejecución para representar a cada *token*, haciendo uso de meta-programación, gracias a la capacidad de introspección y reflexividad de *Ruby*. Luego de haber generado todas las clases, se lee el archivo de entrada, en el caso de lograr reconocer un *token*, se creará una instancia de la clase a la que pertenece y se guarda en una lista que posteriormente será la entrada del analizador sintáctico. En caso de que existan *tokens* irreconocibles, se imprimirán cada uno de ellos con un mensaje de error.

4.8.2. Análisis Sintáctico

Para realizar el análisis sintáctico de \O\kern-1.5pt\TeX se uso *Racc*, una herramienta que generar un *parser* a través de una gramática libre de contexto. Luego de definir una gramática libre contexto que no posea conflictos, la herramienta genera un archivo que posee una clase llamada *Parser*. Esta clase se instancia compuesto con un objeto que reconozca *tokens*, es decir, el analizador léxico previamente descrito. En el momento que consiga un error sintáctico, inmediatamente imprimirá un mensaje de error dando la ubicación del carácter que no es reconocido por la gramática definida.

Cuando el análisis sintáctico tiene éxito, el *parser* da como resultado un árbol sintáctico abstracto. Este árbol es una composición jerárquica de instancias de distintas clases que representan elementos presentes en la especificación del lenguajes. Todas las clases que representan algún elemento en \LaTeX pertenecen al árbol sintáctico abstracto y son creadas a tiempo de ejecución, usando meta-programación; sólo debe especificarse como se llama la clase y cuales son sus atributos. Por otro lado, cada clase debe abrirse¹ nuevamente para definir un método que realice la traducción a *Ruby*. Cada clase del árbol sintáctico abstracto tiene una correspondencia con un expresión en *Ruby*, la cuál está definida en dicho método.

Tomando en consideración la posibilidad de extender e incorporar elementos o estructuras al lenguaje. Se debe definir como es reconocido sintácticamente dicho elemento en la gramática del *parser*, agregar la clase que lo representa y maneja en el árbol sintáctico abstracto, y por ultimo definir como será la correspondencia con *Ruby*.

4.8.3. Librerías

Ya generado un archivo *Ruby*, éste depende de clases y métodos previamente definidos, que manejen los elementos tipográficos que se hayan considerado hasta el momento en \LaTeX . Estas clases y métodos crean una correspondencia entre *Ruby* y \LaTeX , por lo tanto cada uno de ellos deben ser definidos para generar su elemento tipográfico \LaTeX correspondiente. Esto brinda la posibilidad de abarcar estructuras en \LaTeX , a medida que se requieran y necesiten; expandiendo progresivamente la librería.

¹En *Ruby* está permitido abrir un clase previamente definida para modificarla. Extenderla o sobre-escribirla

Capítulo 5

OffTEX

OffTEX es un lenguaje de programación de dominio específico, imperativo, destinado a la composición tipográfica y orientado a objetos. En el diseño del lenguaje se tienen elementos comúnmente usados en los lenguajes de programación, así como estructuras de datos, estructuras de control, funciones, procedimientos, etc.

OffTEX es un lenguaje de dominio específico (*DSL*) externo, es decir, que tiene su propia sintaxis, embebido en *Ruby*. Esta decisión fue tomada porque *Ruby* es un lenguaje orientado a objetos y así, todo el trabajo que conlleva el manejo de objetos en el lenguaje, el alcance dinámico y el modelo de referencia son realizados por *Ruby*. Además, el mismo crea un ambiente propicio para creación de *DSLs*, haciéndolo el mejor candidato para la implementación de OffTEX.

Ahora, considerando la composición tipográfica, se tomó en cuenta a LATEX como herramienta para la generación y estructuración del texto, exportado en *PDF*, que sería el producto final, es decir, un documento. Por lo tanto, OffTEX es un traductor (compilador) de *Ruby* a LATEX, demostrando que un documento puede ser abstraído a un nivel más alto y mantener una filosofía de generar documentos de gran calidad estética, de propósito académico y al mismo tiempo poseer una sintaxis compresible.

5.1. Estructura de un archivo O^HTEX

Un archivo de O^HTEX puede tener definiciones, instancias, estructuras de control, generar documentos, etc; o ninguna de ellas. No es necesario definir un método *main* para que sea un programa válido.

La estructura de un programa O^HTEX se muestra en la figura 5.1.

Figura 5.1: Programa básico: O^HTEX

```
use <paquete>

<documento> :: <clase_documento> {
    <atributo> : <valor>
}

<instancia1> :: <clase> {
    ...
}

<instancia2> :: <clase>

if <condicion> {
    ...
} else {
    ...
}

<documento> << <instancia1>

<documento> >> '<nombre_archivo>'
```

Durante toda la ejecución del programa, en el archivo pueden crearse *n* objetos, pero eso no le indica al lenguaje que esos *n* objetos estarán presentes en un documento o no, si es que se decide exportar alguno.

Por lo tanto, un programa en O^HTEX puede contener n objetos sin generar una salida pero ya se habría realizado un análisis lexicográfico, sintáctico y contextual del mismo.

5.2. Ejecución

La herramienta comienza leyendo de arriba a abajo, interpretando, creando objetos, interactuando con los mismos y al final puede o no generar uno o más documentos. Crear un objeto no implica que el mismo estará en un documento.

5.2.1. Línea de comandos

La ejecución de la herramienta, se hace a través de la línea de comando y sus opciones se muestran en la tabla 5.1.

Tabla 5.1: Comando de consola

ohtex	[OPTION] ... FILE	
	- -tex-command=CMD	especify the TeX command to compile
	- -ruby-command=CMD	especify the Ruby command to use
	- -output-directory=DIR	use a existing DIR as the directory to write files in
	-a, - -all	generate all the TeX output files on each case (aux, dvi, log, out, ps, toc).
	- -aux	generate the aux output file of TeX
	- -dvi	generate the dvi output file of TeX
	- -log	generate the log output file of TeX
	- -out	generate the out output file of TeX
	- -ps	generate the ps output file of TeX
	- -toc	generate the toc output file of TeX
	- -help	prints this help
	- -version	output version information and exit

5.3. Paquetes e Inclusiones

El uso de paquetes en L^AT_EX es parte esencial en la creación de un documento. Facilitan la creación de elementos tipográficos, agrupando un conjunto de instrucciones que tienen un objetivo compartido. Por otro lado, en algunas ocasiones un documento puede ser extenso o un objeto tipográfico puede ser usado más de una vez en distintos documentos. Por ello, se tiene la capacidad de incluir objetos declarados en otro archivo, pudiendo dividir un documento de una forma más ordenada y estructural.

5.3.1. Paquetes

La palabra clave `use` es usada para indicarle al lenguaje cuáles de los paquetes de L^AT_EX se desean usar en cualquiera de los documentos que se vayan a generar. Ciertos paquetes poseen *syntactic sugar* en O^HT_EX y otros no. En este último caso, igualmente se puede usar el paquete pero debe usarse con un *verbatim* de L^AT_EX. Algunos de los paquetes exigen cero (opcionales) o más parámetros; en cualquiera de los casos, la sintaxis para el pase de parámetros es por nombre-valor, de manera secuencial, sin orden; se muestra en la figura 5.2.

Figura 5.2: Uso de paquetes

```
use babel { lang : english }
use microtype { potrusion : true; expasion : true }
use geometry {
    paper      : letter
    left       : 1in
    right      : 1in
    top        : 1.5in
    headheight : 1in
    headsep    : 0.3in
}
```

Por otro lado, se pueden especificar varios paquetes en una misma línea separados por espacios en blanco, eso sólo en el caso de que ninguno de los paquetes necesita parámetros, de la

manera que se presenta en la figura 5.3.

Figura 5.3: Uso de paquetes (*inline*)

```
use amsmath amsfonts amsthm
```

Esto resulta cómodo y flexible al momento de escribir, por otro lado existen paquetes que vienen en conjunto ó tienen un propósito similar. Evitando escribir repetidamente la palabra reservada `use`.

5.3.2. Inclusiones

Es posible usar un archivo O^HT_EX que posea ciertos objetos de interés desde otro archivo. Eso sería en el caso de que se decida agrupar un conjunto de instrucciones, clases, instancias; similares, con un mismo fin o arbitrarios, y que pueda usar en distintos archivos sin la necesidad de escribirlos nuevamente.

Para ello, se usa la palabra reservada `include`, seguido de un *string*, con comillas simples, del nombre del archivo. Y así, todo objeto definido en dicho archivo estará en el alcance actual. Cabe destacar que toda inclusión debe ser realizada al principio de cada archivo, si no será considerado un error.

5.4. Lexicografía

5.4.1. Identificadores

Cada identificador es un nombre que se usa para referir a una variable, método, comando, o paquete.

De igual manera a como se permite en *Ruby* y la mayoría de los lenguajes, los identificadores constan de caracteres alfanuméricos (a-zA-Z0-9) y guiones bajos (_), pero sin que comience por número y que comience siempre por minúscula. También está permitido que los identificadores de los métodos puedan terminar en interrogación (?), exclamación (!) o igual (=), sin restricciones en la longitud del identificador. Por ejemplo, la figura 5.4.

Figura 5.4: Nombres de identificadores

```
foo  
bar  
baz  
this_is_my_4er_id
```

5.4.2. Comentarios

Los comentarios de línea serán a partir de dos puntos seguidos (..) hasta el salto de línea (sin incluirlo). En caso de que desee incluir el salto de línea en el comentario se deben usar tres puntos seguidos (...). Por ejemplo, en la figura 5.5.

Figura 5.5: Comentarios de línea

```
.. esta linea no hace nada  
  
a = 5 .. "a" es una variable local inicializada con 5  
  
a = 2 + ... esto es mas util con expresiones largas  
3
```

Los comentarios de bloque son a partir de dos puntos y una llave que abre (..{}) hasta una llave que cierra seguida de dos puntos (}..). Por ejemplo, en la figura 5.6.

Figura 5.6: Comentarios de bloque

```

...{

    Todo lo que se encuentre dentro de este bloque sera
    ignorado. El uso de comentarios de bloques anidados
    esta permitido para facilidad del programador,
    ...{

        asi que es posible hacerlo.

    }.

}..

```

5.4.3. Palabras reservadas

Las palabras reservadas en O^HT_EX se presentan en la tabla 5.2.

Tabla 5.2: Palabras reservadas

and	break	case	class	else
elsif	false	for	cmd	if
nil	or	return	self	super
true	use	when	while	in

5.4.4. Expresiones

Al igual que en *Ruby* (dado que O^HT_EX es un lenguaje embebido), todo es una expresión. La secuenciación es dada por los saltos de línea y el punto-y-coma (“;”).

5.5. Variables y Literales

O^HT_EX, al ser un lenguaje embebido en *Ruby*, mantiene un símil en ciertos aspectos de su sintaxis, así como su alcance y declaración de variables. De alguna manera es predecible

y eso ayuda a reducir el tiempo de aprendizaje y brinda la posibilidad de generar programas intuitivamente, enfocándose esencialmente en la objetivo final: generar un documento con una excelente composición tipográfica de manera estructural.

5.5.1. Variables globales

Toda variable que comience con dolar (\$) tiene alcance global, dicha variable puede ser accedida en cualquier momento de la ejecución desde cualquier alcance. En este caso no existe alguna restricción en el uso de mayúsculas o minúsculas. Por ejemplo, la figura 5.7:

Figura 5.7: Variables globales

```
$foo
```

5.5.2. Variables locales

Una variable local es aquella que comienza con un letra en minúsculas (a-z), underscore (_) o una llamada a un método. Además dicha variable será asequible en el alcance donde fue declarada, exclusivamente.

5.5.3. Variables de instancia

Las variables de instancia son definidas usando la palabra reservada attr y son referidas por medio de un arroba (@). Como se muestra en la figura 5.8.

Figura 5.8: Variables de instancia

```
@bar
```

5.5.4. Variables de clase

Las variables de clase son definidas igualmente como se hace en *Ruby*, usando doble arroba (@@). Pueden ser accedidas desde cualquier instancia. Por ejemplo, la figura 5.9.

Figura 5.9: Variables de Clase

```
@@baz
```

5.5.5. Variables y constantes predefinidas

Las pseudo variables disponibles en O^HT_EX son:

- `self`. Siendo un lenguaje orientado a objetos, existe una variable que refiera al objeto actual.
- `super`. Siendo un lenguaje orientado a objetos, existe la posibilidad de referir a los miembros de la clase padre dentro de un objeto.
- `nil`. Representa a nulo o a nada.
- `true`. Representa al valor booleano “cierto”.
- `false`. Representa al valor booleano “falso”.

5.6. Formato de texto

En esta sección se describen varios aspectos para darle un formato específico al texto. Se podrá ver que existe una similitud con *Markdown* y eso es para brindar facilidad cuando se genera un programa.

5.6.1. Tamaño de fuente

En O^TE_X, al igual que en L^AT_EX, se permiten tres tamaños de fuente (por puntos [pt]) para el documento, pero puede formatearse una sección o texto en particular con las siguientes instrucciones:

Tabla 5.3: Tamaño de fuente

Comando	10pt	11pt	12pt
\tiny	5	6	6
\scriptsize	7	8	8
\footnotesize	8	9	10
\small	9	10	10.95
\normalsize	10	10.95	12
\large	12	12	14.4
\Large	14.4	14.4	17.28
\LARGE	17.28	17.28	20.74
\huge	20.74	20.74	24.88
\Huge	24.88	24.88	24.88

Estas instrucciones son aceptados dentro de una cadena de caracteres, usando la misma sintaxis de L^AT_EX, como si se estuviese escribiendo directamente. Esto es porque se considera que son útiles y expresivos. Cada punto sigue el estándar norte-americano de *1pt* equivalente a *0.35136mm*.

5.7. Familia de fuente

En cuanto a la familia de la fuente, existen tres y cada una con ciertas fuentes que pueden usarse. Aquí se muestra una tabla con las opciones que vienen por defecto en L^AT_EX, por ende, en O^TE_X:

En el caso de O^TE_X, para especificar otra fuente, distintas a la que viene por defecto en cada familia, se usa la sintaxis de la figura 5.10.

Tabla 5.4: Familias de fuente

<i>Fuentes Serif</i>	
cmr	Computer Modern Roman (default)
lmr	Latin Modern Roman
pbk	Bookman
bch	Charter
pnc	New Century Schoolbook
ppl	Palatino
ptm	Times
<i>Fuentes Sans Serif</i>	
cmss	Computer Modern Sans Serif (default)
lmss	Latin Modern Sans Serif
pag	Avant Garde
phv	Helvetica
<i>Fuentes Typewriter</i>	
cmtt	Computer Modern Typewriter (default)
lmtt	Latin Modern
pcr	Courier

Figura 5.10: Familias de fuente y fuentes

```
doc :: Article {
    font-family : rm .. puede ser 'sf' o 'tt',
    font-typeface : ptm
}
```

Si la fuente que se escogió no pertenece a la familia, se dará un mensaje de error. En caso de que no se especifique la familia pero sí la fuente, la misma será inferida y si solo se especifica la familia, se usará la fuente por defecto.

5.7.1. Cursiva

Para cambiar el formato de algún texto en específico, el mismo debe estar entre dos asteriscos (*), sin espacio al principio y sin saltos de línea, o simplemente se usa la instrucción de LATEX \textit o \emph, dependiendo del caso.

5.7.2. Negrita

Similar a la sección anterior, se debe agrupar el texto, pero dentro de cuatro asteriscos (*), dos de cada lado del texto y siguiendo las mismas restricciones, pudiendo usar la instrucción \textbf{.

5.8. Literales

Los literales son una notación para representar un valor en bajo nivel, y en este lenguaje, como en la mayoría, existen literales para números, booleanos, *string* y estructuras como arreglos, tablas de *hash* (diccionarios) y rangos.

5.8.1. Números

Los números en O^HT_EX son representados en arábigo, y pueden interpretarse como enteros o flotantes. Se considera que la precisión de los números puede mantenerse sencilla, dado que no es relevante, como lo sería en un lenguaje destinado al cálculo numérico o métodos numéricos que requieren manejar la precisión de punto flotante. Por ejemplo, la figura 5.11.

Figura 5.11: Numeros

```
i = 9 .. Entero
f = 4.2 .. Flotante
```

5.8.2. Strings

Un *string* comienza y termina con una comilla simple ('), pudiendo estar sujeto a la incorporación de una expresión, agrupando la misma entre llaves precedidas de un {\$. Por ejemplo, la figura 5.12.

Figura 5.12: String e interpolación

```
s = 'Hola Mundo'  
i = 'Tengo ${3 + 6} manzanas'
```

El uso de comillas dobles no es permitido, y todo carácter que es escrito dentro de un *string* se considerará de forma literal, es decir, que no existe notación con *backslash*, excepto para '\'' y '\\"'.

5.8.3. Arreglos

Los arreglos en O^HT_EX son “heterogéneos”, dado que todo es un objeto, en todo momento los arreglos son de objetos. Para la definición de un nuevo arreglo se puede usar la notación de doble dos-puntos o con elementos separados por coma simple entre corchetes. Como se muestra en la figura 5.13.

Figura 5.13: Arreglos

```
array1 :: Array

array2 = [3,6,9]

.. acceso por indice
array2[0] .. 3
.. acceso por indice negativo
array2[-1] .. 9
.. sub arreglo usando rangos
array2[0..1] .. [3,6]
```

5.8.4. Tablas de Hash

Conocidos también como diccionarios, son estructuras que almacenan pares de objetos, una llave y su valor. Los valores son accedidos por su llave, que debe ser única. Como se muestra en la figura 5.14.

Figura 5.14: Hash (diccionario)

```
hash1 :: Hash

hash2 = {
    'foo' : 9
    'bar' : 27
    'baz' : 42
}

hash3 = { 'foo' : 9; 'bar' : 27; 'baz' : 42 }

.. acceso por llave
hash2['bar'] .. 27
```

5.8.5. Rangos

Son la representación de un subconjunto de valores posibles de los enteros (sólo se pueden usar para esta clase), incluyendo los extremos. Por ejemplo, la figura 5.15.

Figura 5.15: Rangos

```
1..9
-5..5

9..0 .. sintacticamente es valido, pero genera un rango
.. vacio
```

5.8.6. Unidades de tamaño y formato de texto

Cuando se habla del formato de un texto, existen medidas a lo largo de todo el documento, tamaño de letra, espacio entre líneas o párrafos y esto se logra con unidades de medida. Como se muestra en la tabla 5.5.

Tabla 5.5: Unidades de tamaño

pt	un punto
mm	un milímetro
cm	un centímetro
in	una pulgada
ex	depende de la fuente actual
em	depende de la fuente actual

Es posible usar estas unidades en atributos que lo permitan, y deben ser precedidas por un número (entero o flotante). Por ejemplo, la figura 5.16.

Figura 5.16: Unidades de tamaño y tamaño de fuente

```
doc.font = 11pt

text.resize 1.5em
```

5.8.7. Verbatim

En ciertas ocasiones se desea hacer uso explícito de una expresión que sea literal en *Ruby* ó *LATEX*. Cabe destacar que el uso de código literal en *Ruby* está sujeto a que se decidió implementar el lenguaje en el mismo y su objetivo es flexibilizar el manejo de objeto y estructuras en el momento que *OHTEX* no pueda o sea conveniente escribir el código en *Ruby*.

En cuanto a una expresión literal de *LATEX*, será posible introducirla en un documento, en cualquier momento de la ejecución. *OHTEX* abarca aspectos y funcionalidades comunes en *LATEX*, pero ciertamente existen elementos tipográficos específicos que hasta el momento no son soportados. Por esto, mientras no exista una sintaxis en el lenguaje para ello, es posible incorporarlo a mano en el documento.

Cualquier código que sea escrito dentro de las llaves podrá usar los objetos de archivo definidos en *OHTEX* u otro verbatim de *Ruby*.

Ruby, por ejemplo en la figura 5.17.

Figura 5.17: Verbatim *Ruby*

```
&ruby {
  ...
  código ruby
}
```

LATEX, por ejemplo en la figura 5.18:

En el caso de L^AT_EX es más simple, solamente se debe incorporar el código entre comillas simples al documento, haciendo uso del operador *insert* (<<). Haciendo distinción entre el preámbulo y los elementos dentro del documento.

Figura 5.18: Verbatim L^AT_EX

```
...
doc1 :: Article {
...
}

.. código ohtex

doc1.preamble << '
\newcommand{cmd}[1]{Un Macro - #1}
,

doc1.document << '
\noindent
párrafo...
,
...
'
```

Podría usarse el operador (<<) para introducir código directamente al documento sin especificarlo y si no se quisiera especificar explícitamente que se agregara algo al preámbulo, se puede usar el operador (<<<) que sólo opera con clases derivadas de Document. Los errores que puedan ser generados por esta incorporación de código a mano, quedan por parte del programador.

5.9. Operadores

5.9.1. Asignación

Ya sea un identificador o un objeto, el símbolo de asignación es el igual (=) por ejemplo, la figura 5.19:

Figura 5.19: Asignación

```
var = 'Hola Mundo'  
int = 27
```

Es posible realizar una auto-asignación, con todo objeto que tenga las instrucciones, (+), (-), (*), (**) o (/), dependiendo de la operación que se desea realizar. Como se muestra en la figura 5.20.

Figura 5.20: Auto-asiganación

```
foo = 9  
foo += 18  
foo -= 6  
foo *= 2  
foo **= 3  
foo /= 1764
```

Cabe destacar que los operadores aritméticos (+), (-), (*), (**) o (/) no son operadores en O^TE_X, sino instrucciones.

5.9.2. Lógicos

Al igual que los operadores aritméticos, los operadores lógicos ($<$), ($>$), (\leq), (\geq), ($=$), (\neq) y ($!$) son instrucciones, dependiendo de cada objeto.

Pero en O^HT_EX existen los operadores `and` y `or`, que corresponden a la conjunción y disyunción respectivamente.

5.10. Estructuras de Control

O^HT_EX, como en muchos lenguajes, posee estructuras de control esenciales para la interacción y manejo de objetos, así como condicionales y bucles, siempre manteniendo una equivalencia con *Ruby*, el lenguaje anfitrión.

5.10.1. Condicionales

Estas expresiones están constituidas por una expresión booleana y un bloque, con un conjunto de instrucciones. En el caso del condicional *case*, se tienen una serie de condiciones seguidos por una secuencia de instrucciones. En cierto casos, este tipo de condicional brinda comodidad pero podría traducirse a un *if-elsif-else*.

Condicional *if*. Figura 5.21Figura 5.21: Estructura de control: Condicional *if*

```
i = 9

if i < 10 {
    i *= i
}
```

Condicional *if-elsif-else*. Figura 5.22Figura 5.22: Estructura de control: Condicional *if-elsif-else*

```
if <condicion> {
    ...
} elsif <condicion> {
    ...
} else {
    ...
}
```

Condicional *case-when*. Figura 5.23Figura 5.23: Estructura de control: Condicional *case-when*

```
i = 27
case i {
    when 25      : i = 'Igual a 25'
    when 1..20   :
        i = 'Dentro de rango'
        i += ' (1..20)'
    when 0       : i = 'Igual a 0'
    else         : .. no pasa nada
}
```

5.10.2. Bucles

Los bucles son una secuencia de instrucciones que se repiten hasta cumplirse una condición. Pudiendo iterar sobre estructuras de datos de una manera cómoda.

while. Figura 5.24

Figura 5.24: Estructura de control: Bucle *while*

```
while <condicion> {  
    ...  
}
```

for. Figura 5.25

La declaración de un *for*, se usa para iterar sobre estructuras de una manera más cómoda, sin preocuparse por aumentar un contador.

Figura 5.25: Estructura de control: Bucle *for*

```
for <variable_local> in <rango/arreglo/hash> {  
    ...  
}
```

En el caso especial de una tabla de hash, se usan dos variables, una que itera sobre las llaves y otro que itera sobre los valores, respectivos uno con el otro. Si se quisiera iterar solamente sobre las llaves o los valores, debe decirse explícitamente. Como se muestra en la figura 5.26.

Si en algún momento se quiere detener algún tipo de bucle, se debe usar la palabra reservada *break*.

Figura 5.26: Estructura de control: Bucle en *Hash*

```

for llave in my_hash.keys {
    ...
}

for valor in my_hash.values {
    ...
}

```

5.11. Instrucciones

En O^HT_EX, existe el concepto de híbrido entre los métodos y funciones con las instrucciones de L^AT_EX; donde hay un identificador, parámetros o aridad, un alcance y posiblemente un valor de retorno o no. Se hace uso de la palabra reservada cmd para indicar una nueva declaración o sobre-escritura de un comando. Como se muestra en la figura 5.27:

Figura 5.27: Declaración de instrucción

```

cmd foo(a, b) { .. los parentesis son opcionales
    return a + b
}

cmd foo 2 { .. la definición anterior es sobre-escrita
    return #1 + #2
}

```

Las instrucciones pueden representar algo complejo como un método o algo tan simple como a un macro, al final pueden ser representados de igual manera pero tienen una finalidad distinta.

5.12. Clases

5.12.1. Definición de clase

Para definir una clase nueva, se debe comenzar con la palabra reservada `class`, seguida del nombre de la nueva clase a crear (el cual debe comenzar con mayúscula). Por ejemplo, la figura 5.28.

Figura 5.28: Declaración de clase

```
class <nombre_de_la_clase> {  
    ...  
}
```

5.12.2. Atributos

Todo atributo que se defina en una clase será pública (variables de instancia) y no existe la posibilidad de que sean privados o protegidos. Referenciar a un atributo desde un comando se hará similar a *Ruby*, poniendo de prefijo un arroba (@) o se puede omitir si no es opacado por un parámetro con el mismo nombre. Es recomendable usar el arroba para saber de que se habla de una variable de instancia.

Para definir los atributos de una instancia se usa la palabra reservada `attr`, seguido de uno o más identificadores separados por coma. Como se muestra en la figura 5.29:

Figura 5.29: Declaración de variables de instancia

```
class A {
    attr a, b

    cmd foo {
        return @a + @b
    }
}
```

Variable de clase

Las variables de clase, siempre serán privadas, a menos que se definan un *getter* y/o *setter*. Éstas pueden ser definidas en cualquier momento de la definición de clase (con tan sólo escribirla). Por ejemplo, la figura 5.30.

Figura 5.30: Declaración de variables de clase

```
class B {
    @@var_class = 27

    cmd foo {
        ...
    }
}
```

5.12.3. Instanciación

En O^HT_EX, los *dos-puntos* dobles (::), representan la instanciación de una clase. Es el constructor de objetos y la referencia a esa instancia es dada por el nombre (identificador) del lado izquierdo (*lhs*). Por ejemplo, la figura 5.31:

Figura 5.31: Instanciación

```
doc1 :: Article {
    ...
}
```

Al construir un objeto, el mismo debe inicializarse (en el caso de que sea necesario), dependiendo si tiene o no valores por defecto. Para ello, se usan las llaves, donde se encierra una secuencia (puede ser vacía) de identificadores que representan algunos (o todos) los atributos del objeto. Cada nombre es seguido del símbolo *dos-puntos* (:) y luego el valor que inicializará el atributo; cada una de estas inicializaciones es separada por salto de línea (\n) o por punto-y-coma (;). En la figura 5.32 se muestra una inicialización (*singleton*):

Figura 5.32: Inicialización

```
font : 10pt
```

5.12.4. Herencia

En el caso de que se desee especificar una clase padre, se usa el símbolo menor-que (<) seguido de la clase padre y entre llaves estarán definidos los atributos y instrucciones. Como se muestra en la figura 5.33

Figura 5.33: Herencia de clases

```
class <nombre_de_la_clase> < <clase_padre> {
    ...
}
```

Cabe destacar que la herencia es simple, toda clase sólo puede tener un padre.

5.13. Estructuración del documento

La estructura de un documento L^AT_EX, está dividida en partes, capítulos, secciones, subsecciones, sub-sub-secciones, párrafos y sub-párrafos. En O^HT_EX existe una clase para cada una de estas divisiones, haciendo la estructura como un árbol (más parecida a un índice), en vez de forma secuencial. Por ejemplo, la figura 5.34.

Figura 5.34: Seccionamiento de un documento

```
ohtex :: Chapter {  
    ...  
}  
  
estructuracion :: Section {  
    ...  
}  
  
ohtex << estructuracion  
    ...
```

Capítulo 6

Conclusiones y Recomendaciones

Luego de diseñar un lenguaje orientado a objetos destinado a la composición tipográfica, es factible decir que el marcado de un contenido es posible a través de abstracción y jerarquía de elementos tipográficos tratados como objetos no secuenciales; para la posterior generación de un documento presentable, con un diseño de calidad. Es razonable decir que existe una pérdida de flexibilidad, en cuanto a las posibilidades de cambiar el formato del contenido, pero se abarca una gran parte de esas cualidades para obtener un nivel de flexibilidad suficiente. Por otro lado, *T_EX* posee una gran flexibilidad pero es complicado de aprender, así que toda esa amplia posibilidad de formatos no son comprendidas, sino sólo después de un tiempo considerable. En muchas ocasiones, el tiempo no sobra y la disposición no es suficiente; y al final, se desea que sea un sistema accesible a quien lo deseé y amigable al usuario.

En este trabajo se usó *Ruby* como lenguaje de apoyo, para tener un sistema de objetos acorde a los requerimientos y sería recomendable eliminar ese intermediario en el proceso de traducción. En este caso, no representa un problema porque la idea era ilustrar un concepto, no crear una herramienta de uso industrial. Sin embargo, podría considerarse una implementación libre de intermediarios en el momento de que se deseé implementar de manera eficiente y de uso regular. Se tomarían en consideración cambios en el diseño, siempre manteniendo consistencia en el lenguaje.

Por otro lado, el uso de L^AT_EX como herramienta de composición tipográfica está basado en su aceptación general, gran flexibilidad y soporte. Pero dado que existe otro alternativo derivada de T_EX, que es ConT_EXt, debería plantearse la posibilidad de integrarla con O^HT_EX, ya sea en sustitución de L^AT_EX o como complemento. Aunque trabajar con T_EX directamente no sería una decisión ligera de tomar pero podría ser ideal.

Por último, T_EX es un lenguaje poderoso, único y es probable que sea así por varios años. Éste es capaz de generar documentos de gran calidad estética y para el usuario (programador) es importante que el mismo pudiese ser sintácticamente agradable; usar el lenguaje por gusto más que por necesidad. Al momento de programar se usa el ingenio, creatividad y cada lenguaje debería incentivar esas aptitudes. Terminando con una cita a *Donald Knuth*, en su libro *Computer Programming as an Art*:

“To summarize: We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the *state of the Art*.”

Glosario de términos

DTP: Desktop Publishing. Se refiere a la creación de documentos usando el diseño y disposición de una página en una computadora personal.

DSL: Domain specific language. Lenguaje de programación aplicable a un dominio específico.

GNU/Linux: Sistema operativo comúnmente conocido como Linux, donde se combinan el *software* GNU y el *kernel* de Linux

GPL: General purpose language. Lenguaje de programación aplicable a distintos áreas o dominios.

L^AT_EX: Macro paquete del lenguaje de programación T_EX, que introduce el concepto de un lenguaje de marcado descriptivo.

LML: Lightweight Markup Language. Lenguajes ligeros de marcado.

Markdown: Lenguaje de marcado ligero que usa una sintaxis para darle formato a un texto plano.

Markup: Familia de lenguajes de marcado. Sistema para realizar anotaciones en un documento que sea distinguible del texto.

O^TE_X: Lenguaje de programación orientado a objetos destinado a la composición tipográfica, incorporado con L^AT_EX.

PostScript: Lenguaje de programación para la creación de vectores gráficos usado para la publicación de documentos.

Syntactic sugar: Es una sintaxis de un lenguaje diseñada para que sea sencilla de entender y leer.

Tag: Palabra o término asignado a una pedazo de texto, que brinda información que describe el mismo.

T_EX: Sistema de composición tipográfica, implementando un marcado procedural.

Type: Pieza de algún material que representa un carácter o símbolo en particular.

Typesetting: Es la composición de texto, haciendo uso de *types* o sus equivalentes en digital.

WYSIWYG: “what you see is what you get”. Paradigma en el diseño de un procesador de texto donde el contenido que se muestra en el editor corresponde de manera similar al producto final.

WYSIWYM: “what you see is what you mean”. Paradigma en el diseño de un procesador de texto donde el contenido del documento se escribe enfocado en la estructura y la presentación es manejada por otra herramienta.

Apéndice A

Ejemplo

```
use inputenc { encode: utf8 }

first :: Article {
    font: 10pt
    paper: letter
}

titulo :: Text {
    align: center
    size: LARGE
    text: '\vspace{4em}\OhTeX\vspace{4em}'
}

texto :: Text {
    text: '\OhTeX{} es un lenguaje de programación orientado a objetos para la composición tipográfica, haciendo uso del sistema de preparación de documentos \LaTeX{}. Ofrece la facilidad de manejar elementos tipográficos de una manera mas abstracta y expresiva, facilitando la comprensión y aprendizaje del mismo. El objetivo no es sustituir a \LaTeX{} o \TeX{}, sino demostrar que estas herramientas podrian ser sencillas de usar y seguir siendo flexibles.'
}

lista :: List { items: ['Modular','Sencillo','Abstracto'] }
```

```
first << titulo << texto << lista  
first >> 'first.tex'
```

Archivo de salida:

OTEX

OTEX es un lenguaje de programación orientado a objetos para la composición tipográfica, haciendo uso del sistema de preparación de documentos LATEX. Ofrece la facilidad de manejar elementos tipográficos de una manera más abstracta y expresiva, facilitando la comprensión y aprendizaje del mismo. El objetivo no es sustituir a LATEX o TEX, sino demostrar que estas herramientas podrían ser sencillas de usar y seguir siendo flexibles.

- Modular
- Sencillo
- Abstracto

Bibliografía

- [1] Frank Mittelbach. L^AT_EX - a document preparation system. <https://www.latex-project.org/>.
- [2] Robin Laakso. T_EX users group. <https://tug.org/>.
- [3] SCSK Corporation. Curl language. <http://www.curl.com/products/prod/language/>.
- [4] Ruby community. Ruby programming language. <https://www.ruby-lang.org/>.
- [5] Richard Eckersley, Richard Angstadt, Charles M Ellertson, and Richard Hendel. *Glossary of typesetting terms*. University of Chicago Press, 2008.
- [6] Joseph Needham and Colin A Ronan. *The shorter science and civilisation in China*, volume 5. Cambridge University Press, 1995.
- [7] Pow-key Sohn. Early korean printing. *Journal of the American Oriental Society*, 79(2):96–103, 1959.
- [8] Elizabeth L Eisenstein. *The printing revolution in early modern Europe*. Cambridge University Press, 2005.
- [9] Monotype GmbH. Linotype history. <http://www.linotype.com/49/history.html>.
- [10] Ralph Corderoy. troff - the text processor for typesetters. <http://troff.org/>.
- [11] Ken Thompson, Joe Condon, and Brian Kernighan. Experience with the mergethaler linotron 202 phototypesetter, or, how we spent our summer vacation. 1980.
- [12] LA Zadeh, In J Belzer, A Holzman, and A Kenit. Encyclopedia of computer science and technology. *Mareel Dekker, New York*, 8, 1977.

- [13] Kim B Bruce. *Foundations of object-oriented languages: types and semantics*. The MIT Press, 2002.
- [14] Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.
- [15] Thomas A. Cargill. The case against multiple inheritance in c++. In *The evolution of C++*, pages 101–109. MIT Press, 1993.
- [16] James H Coombs, Allen H Renear, and Steven J DeRose. Markup systems and the future of scholarly text processing. *Communications of the ACM*, 30(11):933–947, 1987.
- [17] John Gruber. Markdown project. <https://daringfireball.net/projects/markdown/>.
- [18] Martin Fowler. Language workbenches: The killer-app for domain specific languages? <http://martinfowler.com/articles/languageWorkbench.html>, 2005. Revisado: 2016-02-18.
- [19] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
- [20] Donald E Knuth. The TEXbook, volume A of computers and typesetting, 1984.
- [21] Comunidad Wikibooks. LATEX wikibook. <https://en.wikibooks.org/wiki/LaTeX>.
- [22] Donald E Knuth. Digital typography, volume 78 of CSLI lecture notes. *CLSI Publ*, 1999.
- [23] Patrick Gundlach. ConTEXt wiki. http://wiki.contextgarden.net/Main_Page.
- [24] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.