Universidad de La Habana
Facultad de Matemática y Computación

# Automatizar la creación de Generadores Multilenguajes

Autor: **Yasmany Arcia Corcho**

Tutor: **MSc. Fernando Rodríguez Flores**

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencia de la Computación

Mayo 2017

*A mi mamá por su confianza,*
*su infinita dedicación y su*
*amor incondicional.*

# Agradecimientos

Agradezco especialmente a mi Mamá y a mi familia. A mi tutor Fernando y a su mamá por su dedicación y apoyo. A todas las personas que forman parte importante de mi vida.

# Opinión del tutor

En los últimos años he sido testigo del desarrollo de varios "Generadores Multilenguajes" (como son llamados en este documento) y yo mismo he desarrollado algunos. A cada uno de ellos le tomó un par de meses llegar a un estado medianamente funcional, y un poco más de un semestre para ser realmente útil. Con el trabajo que se defiende en este documento de tesis, esos tiempos se pueden reducir a un par de días y un par de semanas, respectivamente. Eso es posible gracias al trabajo de Yasmany.

Durante la realización de este trabajo Yasmany mostró una excelente capacidad de abstracción, habilidades en la detección y solución de problemas, y capacidad para asimilar nuevos lenguajes de programación y nuevas formas de pensar, completamente diferentes a las adquiridas durante las asignaturas de la carrera.

Creo que estamos en presencia de un excelente trabajo que evidencia que Yasmany Arcia Corcho reúne todos los requisitos para desempeñarse como un excelente graduado de Ciencia de la Computación.

# Resumen

La implementación de un Lenguaje de Dominio Especifico (DSL) requiere diseñar un programa capaz de leer líneas de texto escritas en dicho DSL, analizarlas, procesarlas y hacer posible su generación de código o implementación hacia otros lenguajes. En dependencia de su propósito, desarrollar un DSL puede necesitar varias etapas, pero la mayoría de estas fases son comunes en todas las implementaciones. En este trabajo se presenta una herramienta para automatizar la creación de DSLs que generen códigos hacia múltiples lenguajes. Dicha herramienta permite a los desarrolladores abstraerse de la implementación de analizadores lexicográficos y semánticos, automatizar la construcción de los nodos del Árbol de Sintaxis Abstracta del lenguaje a diseñar y simplificar la generación de código a los lenguajes que se desee exportar.

# Abstract

The implementation of a Specific Domain Language (DSL) requires designing a program capable to read lines of text written in that DSL, analyze and process them and make possible its code generation or implementation into other languages. Depending on the purpose of the DSL, developing it may need several stages, but most of these stages are common in all implementations. This paper presents a tool to automate the creation of DSLs that generate codes into multiple languages. This tool allows developers to abstain from the implementation of lexical and semantic analyzers, automate the construction of the Abstract Syntax Tree nodes of the language to be designed and simplify the code generation of the languages to be exported.

# Índice general

# Introducción

Un lenguaje de dominio específico (DSL, por sus siglas en inglés) es un conjunto de construcciones y operaciones que brindan una mayor expresividad para representar soluciones de un dominio en particular [15]. Los DSLs pueden ser internos en un lenguaje de programación o externos [24]. Algunos ejemplos de DSLs son: BNF (*Backus Normal Form*) para describir la sintaxis de los lenguajes de programación [3]; Matlab [27], utilizado para el trabajo con matrices, y el lenguaje de programación R, diseñado por estadísticos y para estadísticos [2]. Por lo general, a partir de programas escritos en estos DSLs, se genera el código ejecutable de los mismos, o se transforman en código fuente de un lenguaje de programación específico y este último se compila [15] [32] [24].

Para el diseño e implementación de DSLs, se pueden usar herramientas como DSL Tools para la plataforma .NET [38] y el framework XText de Java [21].

Existen escenarios en los que se desea representar en distintos formatos (o lenguajes de programación) el código de programas escritos en un DSL específico. Por ejemplo, cuando se quiere representar el mismo documento en varios formatos como HTML [29] y PDF [19], o cuando se desea obtener el código fuente en varios lenguajes de programación de bibliotecas de Diferenciación Automática implementadas en un DSL [14]. Para el primer escenario, existen aplicaciones como TexInfo [31], Markdown [39], y el Modo Org de Emacs [35], que a partir de la representación de un documento en un lenguaje de marcado específico, permiten obtenerlo en varios formatos; y para el segundo escenario, existen aplicaciones como ADOL* [7]. Este tipo de herramientas constituyen el objeto de estudio de este trabajo, y serán llamadas Generadores Multilenguajes.

Un Generador Multilenguaje es una aplicación que define un DSL, y permite representar en múltiples formatos los programas escritos en él. Dichos

formatos serán llamados lenguajes de salida. Para construir un Generador Multilenguaje, es necesario diseñar e implementar un DSL que represente su dominio y posteriormente especificar cómo se escribe cada componente del DSL en los lenguajes de salida.

En los últimos años, en la Facultad de Matemática y Computación de la Universidad de La Habana (MATCOM) se han desarrollado Generadores Multilenguajes en varios dominios como *ADOL** para la creación de bibliotecas de Diferenciación Automática; *LAML* que permite expresar modelos de programación matemática [17]; y un lenguaje para la descripción de criterios de vecindad en problemas de enrutamiento de vehículos (IVNS) [26]. Todas estas aplicaciones se han implementado como DSLs internos en el lenguaje de programación Common Lisp, aprovechando la notación infija del mismo, y utilizando como palabras clave del DSL los constructores de las clases que representan los nodos de los posibles Árboles de Sintaxis Abstracta (ASTs).

En la construcción de un Generador Multilenguaje se puede identificar un conjunto de pasos necesarios para el desarrollo del mismo:

1. Determinar el dominio que se quiere representar en el DSL.
2. Identificar los elementos que se desean representar en el DSL.
3. Diseñar el lenguaje del DSL.
4. Implementar una jerarquía de clases que permita representar cada uno de los elementos del DSL como nodos de un AST.
5. Construir el analizador lexicográfico y el sintáctico.
6. Implementar la generación de código de cada uno de los nodos del AST, en cada uno de los lenguajes de salida.

Cuando el DSL del Generador Multilenguaje es un lenguaje interno de Common Lisp, el paso 3 se convierte en una selección apropiada de nombres para los constructores de las clases del AST, y el paso 5 no es necesario, pues este proceso lo realiza el intérprete de Lisp. Además, los pasos 4 y 6 se pueden automatizar utilizando macros y el Sistema de Objetos de Common Lisp.

La principal contribución de este trabajo es la propuesta de una nueva herramienta en Common Lisp, denominada Automatización de Generadores

Multilenguajes (MGA[1]), para la automatización de los pasos 4 y 6 en la creación de Generadores Multilenguajes en Common Lisp. Esto se traduce en automatizar el proceso de creación de las clases para representar los elementos del dominio como nodos de un AST y simplificar la generación de código de cada uno de estos nodos en los diferentes lenguajes de salida.

Para desarrollar este trabajo fue necesario estudiar el diseño e implementación de DSLs, elementos de compilación y el lenguaje de programación Common Lisp. Dichos elementos se presentan en el Capítulo 1. Independientemente del lenguaje de programación donde se desee implementar un Generador Multilenguaje, los pasos para construirlo son similares. En el Capítulo 2 se ilustran sus pasos a través de la definición de BASAR, un Generador Multilenguaje orientado al trabajo con operaciones aritméticas. El Capítulo 3 está dedicado a presentar como se pueden implementar estos programas en Common Lisp, a encontrar patrones de código que se repiten en este proceso; y se presentan los macros `defnode` y `gcode`, cuyo uso elimina el código que se repite. El código que generan los macros `defnode` y `gcode` es completamente modificable como se muestra en el Capítulo 4. Finalmente se presentan las conclusiones, recomendaciones y la bibliografía consultada.

[1] Multi-languages Generators Automation

# Capítulo 1

# Preliminares

En este capítulo se presentan los elementos fundamentales para el desarrollo de este trabajo. En la sección 1.1 se presenta un conjunto de definiciones que serán utilizadas a lo largo del documento. Las otras secciones están dedicadas a las herramientas necesarias para automatizar la construcción de Generadores Multilenguajes en Common Lisp, que constituye el objetivo principal de este proyecto. Estas herramientas son: los Lenguajes de Dominio Específico (DSL), los Árboles de Sintaxis Abstracta (ASTs) y, por último, el lenguaje de programación Common Lisp, en el cual está implementada la herramienta propuesta en este trabajo.

## 1.1. Definiciones básicas

En la computación, existen diversos lenguajes para representar un problema de un área en particular. Esto obliga a los desarrolladores a escribir la misma solución en cada lenguaje donde desee emplearla. Un camino para simplificar el trabajo de los programadores en esta situación, sería implementar la solución en un Generador Multilenguaje.

**Definición 1** *Un Generador Multilenguaje es una aplicación que define un DSL, y permite que los programas escritos en él puedan ser representados o exportados a varios lenguajes.*

Ejemplos de Generadores Multilenguajes son: Markdown [39] y ORG-Mode [35] que se especializan en exportar documentos a distintos formatos

como HTML [29], LaTeX [16], u ODT [1], a partir de un lenguaje de marcado que ellos definen; ADOL* es implementado en Common Lisp para representar Rutinas de Diferenciación Automática en varios lenguajes de programación [7]; Problemas de enrutamiento de vehículos (IVNS) [26]; Problemas de programación matemática (LAML) [17].

**Definición 2** *El dominio de un Generador Multilenguaje es el área de trabajo en que se especializa.*

El dominio de ADOL* es la Diferenciación Automática; el de Markdown, la edición de documentos; y el de LAML, los modelos de programación matemática [17].

**Definición 3** *Un elemento del dominio representa una parte específica del área de trabajo.*

Algunos elementos en el dominio de Markdown son los textos, las listas de viñetas, y las referencias a documentos. En el dominio de (IVNS) se pueden encontrar los vehículos, los criterios de vecindad y las posibles rutas para los vehículos.

**Definición 4** *Un lenguaje de salida es un lenguaje en el cual se desea representar los programas escritos en un Generador Multilenguaje.*

Entre los lenguajes de salida de Markdown se encuentran HTML, LaTeX, y ODT [1]. En el caso de ADOL*, un lenguaje de salida, es cualquier lenguaje de programación que posea sobrecarga de operadores.

**Definición 5** *Un Generador Multilenguaje es extensible si permite agregar nuevos lenguajes de salida.*

ADOL*, ORG-Mode y Markdown constituyen Generadores Multilenguajes extensibles. ADOL* permite agregar cualquier lenguaje de programación con sobrecarga en los operadores. ORG-Mode define una interfaz para implementar nuevos exportadores.

## 1.2. Lenguajes de Dominio Específico

Los ingenieros de software se encargan de encontrar herramientas para representar soluciones a problemas de las ciencias computacionales. Unas de

las principales tendencias para la creación, modernización y mantenimiento de programas es el modelado de dominio especifico (*DSM*) [30]. Esta técnica propone crear abstracciones de los problemas para que los usuarios expresen fácilmente sus soluciones y ha sido utilizada en el desarrollo de lo que se conoce como Lenguajes de Dominio Específico (DSL).

Un DSL es un lenguaje de programación destinado a aumentar la productividad de los usuarios al resolver un problema dentro de un contexto específico [32]. Estos tienen la característica de ser típicamente más pequeños que los lenguajes de propósito general, pero al complementarse con otros ambientes de producción brindan un mejor nivel de abstracción sintáctico y semántico para la representación de un problema, permitiendo de esta forma ocultar las complejidades que aparecen en el proceso. Precisamente por ese motivo, y porque las herramientas actuales permiten crear nuevos lenguajes de programación con relativa facilidad, el uso de los DSLs es cada vez más frecuente [24]

En las siguientes secciones se presentan características de los DSLs relevantes para este trabajo: si son externos o internos y las etapas en su construcción.

### 1.2.1. DSLs internos o externos

Desde el punto de vista de la construcción del lenguaje, los DSLs se pueden clasificar en *internos* o *externos*. Los *internos* forman parte de la sintaxis de un lenguaje de propósito general existente, usualmente en forma de librerías. Entre ellos se pueden mencionar Rails (construido en Ruby) [20] y ASP.NET Core [10], ambos orientados al desarrollo de aplicaciones web; el Sistema de Objetos de Common Lisp (CLOS) [33] y Entity Framework [9], un ORM (por sus siglas en inglés Object Relational Mapping) para .NET.

Los *externos* se caracterizan por tener su propia sintaxis. Producto de que son construidos desde cero, estos tienen una sintaxis más especializada que los internos, pero requieren de la construcción de un compilador. Algunos ejemplos son MATLAB, para el trabajo con matrices y cálculos científicos [27]; Graphviz, orientado al dibujo de grafos [11], y R, cuyo propósito es brindar un lenguaje de programación orientado al trabajo estadístico [2].

### 1.2.2. Construcción de un DSL

En la literatura sobre Lenguajes de Dominio Específico se han identificado cinco etapas en la construcción de un DSL: *Decisión*, *Análisis*, *Diseño*, *Implementación* y *Despliegue* [15] [32] [24].

En la etapa de *Decisión* se investiga si es factible o no crear un DSL para el dominio seleccionado; posteriormente, en la etapa de *Análisis* se investigan los elementos necesarios para modelar dicho dominio. Durante la etapa de *Diseño*, se elige un lenguaje de programación en el cual implementar el DSL y se considera qué patrón de diseño utilizar [15]. Terminadas las etapas anteriores, corresponde la etapa de Implementación del DSL. Por último, se desarrolla la etapa de *Despliegue*, que según Meyer [25], es la etapa más compleja durante la vida útil de un software.

En este trabajo, se implementó una herramienta para diseñar DSLs internos en Lisp, asumiendo que las etapas de *Decisión*, *Análisis* y *Diseño* para la creación de un DSL están cumplidas correctamente. En la siguiente sección se presentan los Árboles de Sintaxis Abstracta, que resultan vitales para la etapa de Implementación de un DSL, y para la generación de código en los Generadores Multilenguajes.

## 1.3. Árbol de Sintaxis Abstracta

Un Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés) es una estructura arbórea utilizada para representar segmentos de código escritos en programas computacionales. Cada nodo del árbol contiene información recolectada durante el análisis lexicográfico y el sintáctico de un programa específico [5]. En la figura 1.1 se muestra un AST correspondiente a la expresión $X = (Y + 2) * (Z + 3)$. Nótese que han sido eliminados los paréntesis y el punto y coma.

Los nodos de un AST suelen ser instancias de clases diseñadas para representar cada operación del lenguaje, de forma que estos almacenen información necesaria de los elementos del dominio de un lenguaje de programación [5].

En MGA, para generar el código fuente de un programa escrito en un DSL en cualquier lenguaje de programación, es necesario obtener el AST correspondiente. Para cumplir este objetivo, existen herramientas como ANTLR [28] y YACC [8], aunque no siempre es necesario utilizarlas.

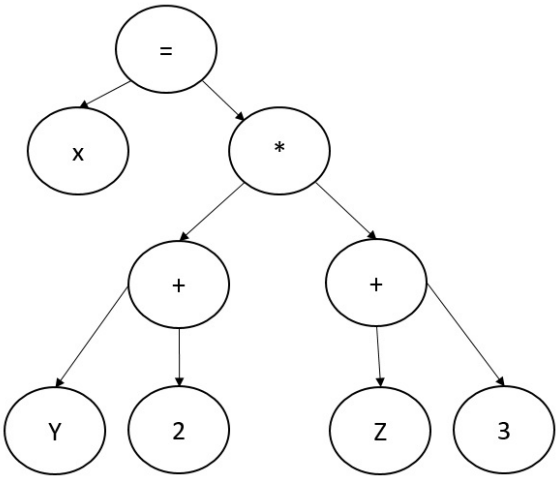


Figura 1.1: AST correspondiente a la expresión: $X = (Y + 2) * (Z + 3)$;

Por ejemplo, en lenguajes como Ruby [20] o Common Lisp [23], cuando el DSL que se está desarrollando es interno, no se requiere el uso de ninguna de las herramientas existentes para crear el AST de un programa, porque estos lenguajes brindan recursos propios, los cuales resultan más convenientes para la creación de este tipo de DSL. Además, como el objetivo de este trabajo es automatizar el proceso de Generadores Multilenguajes en Lisp, no es necesario realizar el análisis léxicográfico y sintáctico porque el intérprete de Lisp realiza esta tarea. La próxima sección está dedicada al lenguaje de programación Common Lisp.

## 1.4. Common Lisp

LISP es un lenguaje de propósito general creado en los años 50 por John McCarthy y fue desarrollado con el objetivo de crear un lenguaje que mostrara la información estructurada en listas, de ahí su nombre List-Processing [13]. Es el segundo lenguaje de alto nivel más antiguo que aún se utiliza en la actualidad, después de Fortran [37]. La variedad de dialectos Lisp existentes en 1981 (Maclisp, Interlisp, zetalisp), provocó que en 1986 se creara el estándar llamado Common Lisp [6].

Common Lisp es un lenguaje multiparadigma. En él, es posible crear programas orientados a objetos [33], completamente funcionales [12], o cualquier

combinación que se desee [6]. Posee un intérprete interactivo conocido por sus siglas REPL [34] y un Sistema de Objetos (CLOS) basado en herencia múltiple y funciones genéricas [33].

Las características que distinguen a Common Lisp del resto de los lenguajes de programación son su Sistema de Objetos y un Sistema de Macros que permite modificar el lenguaje para ajustarlo a las necesidades del usuario [34] [33]. Estas son las razones fundamentales por la que se seleccionó Common Lisp para el desarrollo de este proyecto de tesis. El Sistema de Objetos simplifica la generación de código fuente en múltiples lenguajes de salida; y el Sistema de Macros, automatiza la construcción de lenguajes para los DSLs y la definición de los nodos de AST.

Para presentar las herramientas que facilitaron la implementación de MGA, las siguientes secciones se dedican a la sintaxis de Common Lisp, a su Sistema de Objetos, y su Sistema de Macros.

### 1.4.1. Sintaxis

La sintaxis de Common Lisp es uniforme. Las funciones, los métodos, los macros y los operadores poseen todos la misma sintaxis.

```
(name arg1, arg2 ... argN)
```

Common Lisp no hace distinción entre las funciones propias del lenguaje y las funciones creadas por el usuario[12]. Esta característica permite crear y utilizar DSLs en Lisp con mucha facilidad [6]. Por ejemplo, empleando funciones con nombres adecuados se puede construir un mini-lenguaje para elaborar páginas web en Lisp. Este lenguaje estará formado por las funciones `html`, `body` y `bold`, donde cada una devuelve la estructura correspondiente en HTML. Entonces una página sencilla se puede escribir como:

```
(html
  (body
    "Un GM para" (bold "HTML")))
```

### 1.4.2. Funciones

En Common Lisp, las funciones son Ciudadanas de primera clase [12]. En el contexto de los lenguajes de programación se dice que una entidad

es Ciudadana de primera clase si puede ser utilizada en operaciones que comúnmente se realizan sobre los números o cadenas de texto: crear una nueva entidad en una rutina, almacenarla en variables o en estructuras de datos, incluirla en los parámetros de funciones o devolverla en llamados a funciones o macros [12]. Esta propiedad de las funciones permite automatizar las definiciones de los lenguajes para los DSLs.

Una función en Common Lisp acepta parámetros posicionales, opcionales, con nombre, o un número arbitrario de ellos. El conjunto de especificaciones en la definición de una función se conoce como Lambda List [34] (en otros lenguajes de programación se conoce como la signatura del método). Una selección apropiada de los nombres de las funciones y los parámetros de las mismas, en combinación con un uso adecuado de los elementos que brinda el lenguaje, permite crear código legible y fácilmente adaptable a cualquier dominio, como el que se muestra a continuación:

```
(make-automation-recognizing (strings-over {0 1})
                             (starting-with 1)
                             (with-at-least 5 0)
                             (ending-with 0))
```

En la siguiente sección, se presenta el Sistema de Objetos de Common Lisp, que resulta muy conveniente para representar los nodos de un AST y realizar la generación de código a múltiples lenguajes.

### 1.4.3. Sistema de Objetos

El Sistema de Objetos de Common Lisp (CLOS, por sus siglas en inglés) es un conjunto de herramientas para desarrollar programas orientados a objetos [33]. Está formado por clases que pueden tener herencia múltiple, instancias, funciones genéricas y métodos [34]. A diferencia de otros lenguajes de programación orientados a objetos, en CLOS los métodos no pertenecen a las clases, y no existe una sintaxis especial para referirse a las propiedades de sus instancias. A continuación, se describen los elementos de CLOS utilizados en este trabajo: clases, funciones genéricas y métodos.

#### Clases en Common LISP

El primer paso para escribir un programa en CLOS es implementar las clases, que definen la estructura y el comportamiento de objetos del mismo

tipo [33][23].

Una clase en Common Lisp utiliza en su definición un conjunto de campos o propiedades (*slots* en la terminología de CLOS) y opciones de clases. En dicho sistema de objetos se pueden construir clases a partir de otras clases (clases padres o super clases en la terminología de CLOS), de las cuales heredan sus *slots* y comportamiento [23][22].

Un ejemplo de la implementación de las clases *operador binario* y *suma* es:

```
(defclass binary-operator ()
    ((left-hand :accessor left-hand
          :documentation "The left value of the operator.")
     (right-hand :accesor right-hand
          :documentation "The right value of the operator."))
     (:documentation: "A binary node in the AST"))

(defclass sum-node (binary-operator)()
     (:documentation "A sum operator node in the AST"))
```

La clase `binary-operator` representa un operador binario. Un operador binario tiene dos *slots*, uno para el operador izquierdo y otro para el derecho. La clase `sum-node` hereda de la clase `binary-operator`.

Los *slots* se definen para almacenar información del estado de una instancia en particular [22]. Para interactuar con ellos, se define un conjunto de opciones: `accessor` define el nombre de la función para acceder al valor del *slot* correspondiente, `initarg` declara el nombre del parámetro, con el cual se inicializa el valor de un *slot* en el momento de instanciar una clase, y el campo `documentation` almacena la documentación del *slot* o la clase [33]. Estas características de las clases serán automatizadas en la herramienta propuesta en este trabajo.

Para crear una instancia de una clase en Lisp, se utiliza la función `make-instance`[33]. Un patrón común para instanciar clases es utilizar funciones, métodos o macros ordinarios que reciben los valores con los que se desea instanciar una clase. El resultado de evaluar esas funciones, métodos o macros, son objetos creados a través de llamados a la función `make-instance`. En el resto de este documento estas funciones serán llamadas funciones constructoras.

Gracias a estas funciones constructoras, puede ser muy sencillo construir

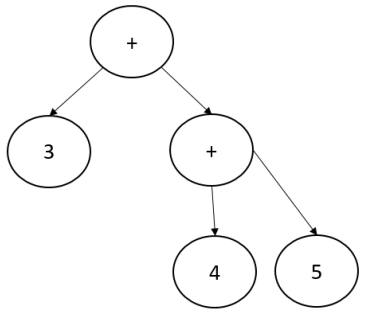


Figura 1.2: AST correspondiente a la expresión: `(+ 3 (+ 4 5))`

un AST en Lisp. Por ejemplo, la función constructora para la clase `sum-node` puede escribirse como:

```
(defun + (left-hand right-hand)
         (make-instance 'sum-node
                 :left-hand left-hand
                 :right-hand right-hand))
```

Con esta definición, el resultado de evaluar `(+ 3 4)` es una instancia del nodo AST `sum-node`. Anidando llamados a las funciones constructuras se puede construir un AST de manera sencilla. En la figura 1.2 de la página 13, se puede observar el AST correspondiente al la expresión `(+ 3 (+ 4 5))`.

Una clase en Lisp puede heredar de varias clases [33]. Esta característica resulta muy conveniente en el diseño de las jerarquías de nodos del AST y de los lenguajes de salida, lo que se puede observar en la sección 3.3.2.

El Sistema de Objetos de Common Lisp utiliza funciones genéricas para especificar las operaciones sobre las instancias de una misma clase [34].

### Funciones Genéricas y Métodos

Una función genérica define solo una interfaz. Su comportamiento depende de la definición de su Lambda List, y su implementación se distribuye entre un conjunto de métodos[36].

La función genérica que se presenta a continuación, se utiliza en ADOL* [7], LAML [17] y IVNS [26] en la etapa de generación de código.

```
(defgeneric generate-code node language stream)
```

Esta función genérica es la clave para generar código hacia varios lenguajes. Recibe tres parámetros: `node`, `language` y `stream`; y sus métodos se especializan en generar el código correspondiente a cada nodo `node` del AST, en cada uno de los posibles lenguajes de salida, en un flujo de salida `stream`.

Por ejemplo, si se quiere generar el código del nodo `sum-node` en el lenguaje C#, resulta necesario definir un método que implemente la función genérica `generate-code` y se especialice en el nodo `sum-node` y en el lenguaje C#, como se muestra a continuación:

```
(defmethod generate-code ((node sum-node)(language csharp) stream)
    (format  stream "~a + ~a"
        (generate-code (left-hand node) language nil)
        (generate-code (right-hand node) language nil)))
```

En CLOS, los métodos pueden ser extendidos por otros métodos llamados auxiliares[18]. Estos se definen insertando una palabra clave `(:before, :after o :around)` después de su nombre y especificando el Lambda List del método que extienden [33]. Dichos métodos, se utilizan para automatizar las características comunes en la sintaxis de los lenguajes de salida, lo que se ejemplificará en la sección 4.2.

A pesar de ser una herramienta extremadamente poderosa [33], CLOS es solo un DSL interno en Common Lisp para el desarrollo orientados a objetos [33]. Este sistema de objetos se pudo incluir en todos los dialectos de Lisp gracias a las funcionalidades del Sistema de Macros que se presentan a continuación.

### 1.4.4. Sistema de Macros

Los macros en Lisp son estructuras sintácticas que permiten transformar fragmentos de código Lisp en otros fragmentos de código [6]. Aunque los macros están presentes en varios lenguajes de programación como C++ y BASIC, los macros de Lisp trabajan de forma completamente diferente y a un nivel más sofisticado [6].

El Sistema de Macros de Common Lisp constituye una forma de extender la sintaxis del lenguaje [12]. Por ejemplo, en Common Lisp no está definida la

sentencia `while` que posee el lenguaje de programación C#. Sin embargo, en Common Lisp se puede definir un macro que implemente esta funcionalidad y permita escribir código como:

```
(while (< x 10)
   (setf x (+ x 1)))
```

La instrucción `while` representa solo una básica sustitución de código, pero con esta idea se pueden crear DSLs para definir iteraciones complejas como el `Loop` de Common Lisp [12], un generador de HTML [13] o un intérprete de PROLOG, este último en solo 145 líneas de código [12].

Los macros se procesan en un tiempo diferente al que se evalúan las funciones o demás operaciones de un programa. Una función en Common Lisp se evalúa cuando se ejecuta un programa que la contenga (*runtime*). Por otro lado, los macros se ejecutan cuando se lee y se compila el programa. Dicho momento se conoce como tiempo de expansión de macros (*macro expansion time*, en inglés). Los parámetros de un macro no se evalúan en el momento en que estos son llamados, sino después de haber compilado el fragmento de código expandido por el macro [12]. La expansión de la llamada anterior al macro `while` sería:

```
(loop while (< x 10)
    doing (setf x (+ x 1)))
```

En este proyecto, los macros se utilizan para crear funcionalidades que permiten simplificar la creación de nodos del AST, definir automáticamente las funciones constructoras y encapsular fragmentos que se repiten en la generación de código.

Con los elementos presentados en este capítulo es posible automatizar la creación de Generadores Multilenguajes en Common Lisp. En el siguiente capítulo se describe en detalle el proceso de creación de un Generador Multilenguaje, para posteriormente presentar cómo se puede automatizar cada uno de sus pasos.

# Capítulo 2

# Implementación de un Generador Multilenguaje

La implementación de un Generador Multilenguaje se puede dividir en dos etapas fundamentales. Primero, se debe diseñar e implementar el DSL, y luego definir cómo se llevará a cabo la generación de código de cada elemento del dominio en los lenguajes de salida.

A su vez, estas dos etapas se pueden desglosar en un conjunto de pasos lógicos, descritos a continuación:

1. Determinar el dominio que se quiere representar en el DSL.
2. Identificar los elementos que se desean representar en el DSL.
3. Diseñar el lenguaje del DSL.
4. Implementar una jerarquía de clases que permita representar cada uno de los elementos del DSL como nodos de un AST.
5. Construir el analizador lexicográfico y el sintáctico.
6. Implementar la generación de código de cada uno de los nodos del AST, en cada uno de los lenguajes de salida.

En las siguientes secciones de este capítulo, se ilustran estos pasos a partir de la creación de un Generador Multilenguaje muy simple, cuyo dominio es el trabajo con operaciones aritméticas. A este Generador Multilenguaje se le llamará BASAR (del inglés Basic Arithmetic Operations).

## 2.1. BASAR

BASAR es un Generador Multilenguaje donde se pueden escribir expresiones aritméticas básicas, que contengan variables y asignaciones a las mismas. Con BASAR se pueden escribir expresiones como:

```
X = 5 + Y * 2
```

Cuando se genere su código a C#, R, y a Common Lisp se obtendría, respectivamente:

```
X = 5 + Y * 2;
```

```
X <- 5 + Y * 2
```

```
(setf X (+ 5 (* Y 2)))
```

Una vez que se tiene bien definido el dominio que se quiere representar, el siguiente paso para la construcción de un Generador Multilenguaje es identificar qué elementos del dominio deben ser incluídos.

## 2.2. Dominio

En el caso de BASAR, los elementos que se desean incluir son:

- Suma
- Multiplicación
- Asignación a variable
- Referencia a variable
- Imprimir

Definidos estos elementos, en el siguiente paso se diseña un lenguaje que permita expresar los elementos del dominio.

## 2.3. Lenguaje

En este paso resulta necesario definir cómo se representa en el DSL cada uno de los elementos del dominio que se quieren incluir. En el caso de BASAR, cada elemento se pudiera representar como se muestra a continuación:

- Suma: Será una función `SUMA` que posee dos parámetros.
  Ejemplo: `SUMA(A,B)`
- Multiplicación: operador infijo `*`.
  Ejemplo: `A*B`
- Asignación a variable: operador `:=`.
  Ejemplo: `A := B`
- Referencia a variable: nombre de la variable.
  Ejemplo: `A`
- Imprimir: función Print.
  Ejemplo: `Print A`

Con este diseño del lenguaje, la expresión $X = 5 + Y * 2$ se escribiría:

```
X := SUMA(5, Y*2)
```

En el caso de MGA, los DSLs se van a diseñar como lenguajes internos en Common Lisp, por lo que la sintaxis siempre será la misma: todos los elementos se representarían como:

```
(name-funtion arg1, arg2,..., argN).
```

En este caso solo habría que definir qué nombre tendrá la función que representa a cada uno de los elementos del dominio, de forma que si la asignación se representa por la función `assign-to`, y la suma y el producto por los operadores `+`, y `*`, la expresión $X = 5 + Y * 2$ se representaría como:

```
(assign-to X (+ 5 (* Y 2))
```

De esta forma se termina de definir el lenguaje. El siguiente paso es representar cualquier cadena del lenguaje en un AST, lo que se ejemplifica en la próxima sección.

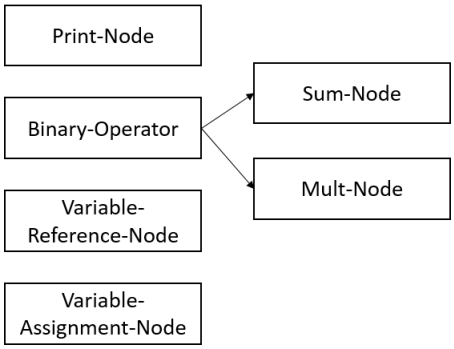


Figura 2.1: Jeraquía de nodos del AST para BASAR

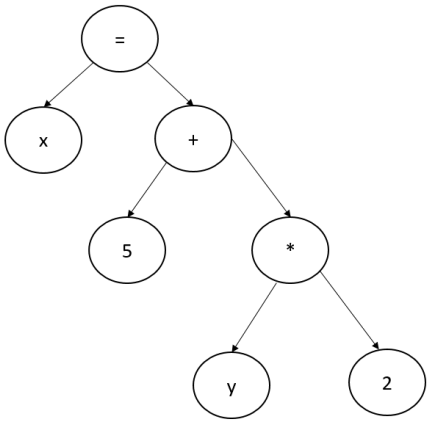


Figura 2.2: AST correspondiente a la expresión: (ASSIGN-TO X (+ 5 (* Y 2))

## 2.4. Nodos del Árbol de Sintaxis Abstracta

Una vez diseñado el lenguaje para un Generador Multilenguaje, se debe crear una jerarquía de clases para los nodos del AST, que dependa de las especificidades del problema. En el caso de BASAR, se define una clase por cada operación que se desea representar. Además se crea la clase `binary-operator`, de la que heredan `sum-node` y `mult-node`. En la figura 2.1 se muestra la jerarquía de clases para este problema, y en la figura 2.2, se muestra el AST para el ejemplo de la sección anterior.

En dependencia del lenguaje de programación en el que se desee im-

plementar un Generador Multilenguaje, puede ser necesario implementar el analizador lexicográfico y el sintáctico, para construir el AST, a partir de las cadenas del DSL. Como en MGA los lenguajes para los DSLs tendrán la misma sintaxis `(nombre arg1, arg2, ..., argN)` y los ASTs se pueden obtener mediante funciones constructoras, el intérprete de Common Lisp realizará estos análisis.

Terminada la construcción del AST, se procede a generar el código de sus nodos a los lenguajes de salida.

## 2.5. Generación de código

Para generar el código de programas escritos en BASAR en cualquiera de los lenguajes de salida, basta con especificar cómo se escribe cada nodo del AST en el lenguaje deseado. De esta forma, para escribir el código de un programa completo se recorre el AST que lo representa, generando el código de cada nodo.

Por ejemplo, para escribir en C# el código correspondiente al AST ilustrado en la figura 2.2, se necesita especificar cómo generar el código de cada uno de los nodos. Si `G(X)` representa la generación de código del elemento `X` se tiene que:

- La asignación en C# es:

```
G(parte-izq) = G(parte-der)
```

- La suma en C# es:

```
G(parte-izq) + G(parte-der)
```

- La multiplicación en C# es:

```
G(parte-izq) * G(parte-der)
```

- La referencia a variable en C# es:

```
Nombre-de-la-variable
```

- Imprimir un elemento en C#:

```
Console.WriteLine(G(A))
```

Una vez definida la generación de código para cada nodo y para cada lenguaje, se puede recorrer el AST generando el código. Los resultados de generar el código del AST ilustrado en 2.2 en C# y en Common Lisp serían:

```
X = 5 + Y * 2
```

```
(setf x (+ 5 (* (Y 2))))
```

Una de las características principales de un Generador Multilenguaje es que permite definir con facilidad nuevos lenguajes de salida. Por ejemplo, una vez definido el DSL y la generación de código hacia C#, si se quiere incluir Common Lisp como un lenguaje de salida, solo habría que definir cómo se escribe cada nodo del AST en este nuevo lenguaje. Si se quisiera hacer la generación de código para otro lenguaje que posea elementos comunes con C# o Common Lisp, se puede reutilizar la generación de código que ya existe.

Los pasos presentados para desarrollar la implementación de un Generador Multilenguaje serán aplicados en el próximo capítulo, cuando se presente la herramienta MGA, que permite agilizar estos pasos.

# Capítulo 3

# MGA

Los Científicos de la Computación estudian para crear programas y algoritmos que minimicen el tiempo de resolución de cualquier problema. Para ello, crean procedimientos y funcionalidades que sean capaces de simplificar procesos mecánicos y repetitivos para el desarrollo de un software. Con esa filosofía, se presenta en este capítulo la herramienta MGA, para aumentar la eficiencia de los usuarios de Common Lisp en la creación de Generadores Multilenguajes.

En la implementación de un Generador Multilenguaje aparecen fragmentos de código que se repiten con frecuencia, y que pueden ser automatizados utilizando el Sistema de Macros de Lisp y las herramientas que brinda CLOS. Por ese motivo, en la siguiente sección se presenta una metodología que permite implementar este tipo de programas en Common Lisp. La sección 3.2 está dedicada a analizar fragmentos de código repetidos durante el diseño de clases y funciones constructoras. Por último, se presenta la técnica utilizada para generar código hacia distintos lenguajes y patrones que están presentes en su implementación.

## 3.1. Generación Multilenguaje en MGA

El desarrollo de Generadores Multilenguajes es el objeto de estudio de este trabajo. En la presente sección se explica cómo puede implementarse este tipo de programa en Common Lisp.

La metodología presentada a continuación para implementar un Generador Multilenguaje en Common Lisp fue utilizada en la definición de varios

programas como: ADOL* [7], LAML [17] y (IVNS) [26]. Precisamente, los pasos repetidos en sus implementaciones, son los que se pueden automatizar con la herramienta propuesta en este trabajo.

El grado de dificultad de implementar un programa computacional está condicionado por el lenguaje de programación seleccionado para su desarrollo, ya que cada uno posee características distintas. Independientemente del lenguaje de programación elegido, los pasos a seguir para implementar un Generador Multilenguaje, usualmente son los mismos:

1. Determinar el dominio que se quiere representar en el DSL.
2. Identificar los elementos que se desean representar en el DSL.
3. Diseñar el lenguaje del DSL.
4. Implementar una jerarquía de clases que permita representar cada uno de los elementos del DSL como nodos de un AST.
5. Construir el analizador lexicográfico y el sintáctico.
6. Implementar la generación de código de cada uno de los nodos del AST, en cada uno de los lenguajes de salida.

Esta investigación se centra en los pasos 4 y 6, porque los pasos 1 y 2 dependen del dominio específico y se supone que ya han sido realizados. El paso 5 no se tiene en cuenta, debido a que los DSLs serían internos en Common Lisp y sus respectivos lenguajes estarán formados por funciones constructoras, por eso, su mismo intérprete realiza el análisis lexicográfico y el sintáctico.

En Common Lisp, los elementos identificados en el paso 2 de la lista anterior, se pueden representar con clases, donde los *slots* de cada una son las propiedades de dicho elemento; estas clases representarán los nodos del AST. Los elementos del lenguaje (palabras clave, declaraciones, instrucciones y funciones) pueden ser las funciones constructoras de las clases. Debido a que estas funciones devuelven instancias de los nodos del AST, y escribir un programa en el DSL consiste en realizar un conjunto de llamados a estas funciones, el resultado de evaluar un segmento de código de ese lenguaje es el AST correspondiente.

Para los usuarios de Common Lisp, implementar un Generador Multilenguaje se traduce en:

1. Diseñar una clase para cada elemento del dominio que se quiera representar en el DSL.

2. Crear una función constructora para cada clase, donde estas funciones constructoras serán los elementos del lenguaje.

3. Implementar la generación de código de cada uno de los nodos del AST, en cada lenguaje de salida.

En todos estos pasos aparecen segmentos de código que se repiten. A continuación, se muestran algunos de ellos en la definición de los nodos del AST.

## 3.2. Definición de los nodos del AST

En la metodología referenciada en la sección 3.1 para implementar Generadores Multilenguajes en Common Lisp, los nodos del AST se obtienen como resultado de evaluar las funciones constructoras de las clases que representan los elementos del DSL. Definir los nodos de un AST en Common Lisp es un proceso mecánico que puede ser automatizado.

Implementar una clase y una función constructora en CLOS se puede ver como dos plantillas a rellenar. Estas plantillas se rellenan con palabras clave especificadas en Common Lisp e información introducida por el programador para definir una clase o función constructora. Por ejemplo, la implementación de una clase requiere la palabra clave `defclass` y la información suministrada por el programador es el nombre de la clase y los *slots* de la misma.

### 3.2.1. Construcción de clases sin MGA

Las clases en Common Lisp poseen un patrón común que puede encapsularse utilizando macros. Para ello, se debe identificar la manera en que se escriben sus plantillas. Con el objetivo de mostrar la estructura de dicha plantilla para definir clases, a continuación, se muestra una implementación de las clases `sum-node`, `mult-node`, `variable-assignment-node`, `variable-reference-node` y `print-node`; correspondientes a los elementos de BASAR: Suma, Multiplicación, Asignación a Variable, Referencia a variable e Imprimir.

```
(defclass binary-operator ()
       ((left-hand
                 :accessor left-hand
                 :initarg :left-hand)
        (right-hand
                 :accessor right-hand
                 :initarg :right-hand)))

(defclass sum-node (binary-operator) ())

(defclass mult-node (binary-operator)())

(defclass variable-assignment-node ()
          ((variable-name
                    :accessor variable-name
                    :initarg :variable-name)
            (value
                     :accessor value
                     :initarg :value)))

(defclass variable-reference-node ()
          ((variable-name
                     :accessor  variable-name
                     :initarg  :variable-name)))

(defclass print-node ()
     ((value
                 :accessor value
                 :initarg :value)))
```

Como se puede observar en la implementación anterior, la estructura para rellenar una plantilla de clase en Common Lisp es la siguiente:

Primero, se debe escribir la palabra clave `defclass`, seguida por el nombre de la clase, una lista con los nombres de sus super clases, y otra con la definición de los *slots*. Cada definición de *slot* posee las palabras clave que identifican las opciones de los *slots* con sus respectivos nombres o valores. Para finalizar, se deben especificar las opciones de las clases, de forma similar a como se realiza con las opciones de los *slots*.

Usualmente en la definición de los *slots* de las clases, los valores de las opciones de los *slots* `accessor` y `initarg`, reciben el mismo nombre que sus correspondientes *slots*. Este patrón se utiliza en la sección 3.2.3.

Concluida la representación de un elemento del dominio en una clase, se implementa su función constructora.

### 3.2.2. Funciones constructoras sin MGA

Para instanciar cualquier tipo de clase, CLOS solo provee la función `make-instance`. Por este motivo, en el cuerpo de toda función constructora debe existir un llamado a `make-instance`.

La función `make-instance` puede ser vista como una plantilla que recibe el nombre de la clase y una lista en la que se asigna a cada *slot* de la clase un valor.

```
(make-instance 'sum-node
               :left-hand  4
               :right  5))
```

Conocida la estructura para la función `make-instance`, se puede crear fácilmente la plantilla de las funciones constructoras. Esta comienza con la palabra clave que identifica el tipo de función constructora (`defun`) y el nombre elegido para dicha función. Después se especifican sus parámetros en correspondencia con los *slots* de la clase a instanciar. En el resto de la plantilla, el programador debe escribir el cuerpo de la función, que debe incluir un llamado a la función `make-instance`. A continuación, se implementan las funciones constructoras para las clases `sum-node`, `mult-node`, `variable-assignment-node`, `variable-reference-node` y `print-node`.

```
(defun sum-node (left-hand right-hand)
          (make-instance 'sum-node
                :left-hand left-hand
                :right-hand right-hand))

(defun mult-node (left-hand right-hand)
        (make-instance 'mult-node
              :left-hand  left-hand
              :right-hand  right-hand))
```

```
(defun variable-assignment-node (variable-name value)
            (make-instance 'variable-assignment-node
                   :variable-name variable-name
                   :value value))

(defun variable-reference-node  (variable-name)
            (make-instance 'variable-reference-node
                   :variable-name variable-name))

 (defun print-node (value)
             (make-instance 'print-node
                   :value value))
```

Nombrando adecuadamente las funciones constructoras y sus parámetros para representar los nodos del AST y combinándolas con las funciones propias de Common Lisp, se pueden crear Generadores Multilenguajes internos con sintaxis y semántica simple. Utilizando estas funciones, desarrollar un Generador Multilenguaje en MGA resultará un proceso sencillo para los usuarios de Common Lisp.

En la sección 3.2.1 se mostró la definición de clases y en la sección 3.2.2 la definición de funciones constructoras a través de plantillas. Estas plantillas se pueden simplificar si se eliminan las especificaciones de Common Lisp para definir las clases y las funciones. De forma que para diseñar los nodos del AST, solo se escriban los nombres de las clases, los de sus super clases, y los de sus correspondientes `slots`. Esta idea se utiliza en el macro `defnode` para automatizar la construcción de clases y funciones constructoras. En la próxima sección se presenta al macro. `defnode`.

### 3.2.3. El macro defnode básico

El tiempo requerido para desarrollar un Generador Multilenguaje aumenta de forma proporcional al número de elementos que se quieran representar en su DSL.

Como se puede apreciar en las secciones 3.2.1 y 3.2.2, implementar la clase y la función constructora para cada elemento de un DSL puede requerir muchas líneas de código. Para simplificar este procedimiento se propone utilizar el macro `defnode` que a partir del nombre de la clase, sus super clases, y sus *slots*, genera automáticamente la definición de la clase y su función constructora. Por ejemplo:

```
(defnode sum-node () (left-hand right-hand))
```

A partir del código anterior se obtiene:

```
(defclass sum-node ()
             ((left-hand
                    :accessor left-hand
                    :initarg :left-hand)
              (right-hand
                    :accessor right-hand
                    :initarg :right-hand)))

(defun sum-node (left-hand right-hand)
                (make-instance 'sum-node
                     :left-hand  left-hand
                     :right-hand right-hand))
```

En la función constructora de `sum-node`, su nombre y sus parámetros fueron generados en correspondencia con el nombre de la clase y el de sus *slots*. Esto también ocurre cuando se genera el código correspondiente a las opciones de los *slots* de una clase. El código generado por `defnode` es completamente modificable. En la sección 4.1 se presentan las opciones que permiten adaptar este código a las necesidades del usuario.

Similarmente, implementar todas las clases de BASAR y sus correspondientes funciones constructoras sería:

```
(defnode mult-node () (left-hand right-hand))

(defnode variable-assignment-node () (variable-name value))

(defnode variable-reference-node () (variable-name))

(defnode print-node () (value))
```

Como se puede inferir de la anterior implementación de las clases que identifican el dominio de BASAR, una representación más elegante para dicho dominio sería construir la clase abstracta `binary-operator` y definirla como super clase de `sum-node`, `mult-node` y `variable-assignment-node`.

Utilizar el macro `defabsnode`, define la clase `binary-operator` como una clase abstracta:

```
(defabsnode binary-operator () (left-hand right-hand))

(defnode sum-node () (binary-operator) ())

(defnode mult-node () (binary-operator) ())

(defnode variable-assignment-node (binary-operator) ())
```

Existen escenarios complejos cuyo dominio está formado por numerosos elementos. Un ejemplo de ello son los Generadores Multilenguajes ORG-Mode, Markdown o ADOL* que en su definición cuenta con varias clases similares a `class-definition-node`. La clase `class-definition-node` fue implementada en ADOL* por la siguiente razón:
Si se quiere hacer una biblioteca de Diferenciación Automática [14], siempre es necesario definir al menos una clase que posea los campos valor y derivada, y sobrecargar todos los operadores aritméticos. Por ese motivo, un DSL para diseñar bibliotecas de Diferenciación Automática tiene que permitir definir clases y para ello, debe existir en el AST un nodo "Definición de Clases". Una implementación equivalente a la realizada en ADOL* sería:

```
(defclass class-definition-node ()
      ((class-name
            :accessor class-name
            :initarg :class-name)
       (class-parent
            :accessor class-parent
            :initarg :class-parent
            :initform nil)
       (slots-definition
            :accessor slots-definitions
            :initarg :slots-definitions
            :initform '())
      (constructor
            :accessor constructor
            :initarg :constructor
```

```
        :initform '())
    (operator-overloads
        :accessor operator-overloads
        :initarg :operator-overloads
        :initform '())
    (functions-definition
        :accessor functions-definitions
        :initarg :functions-definitions
        :initform '())
    (methods-definition
       :accessor methods-definitions
       :initarg :methods-definitions
       :initform '())))
```

Igualmente, definir la clase `class-definition`, y su función constructora sería:

```
(defnode class-definition (instruction-container)
      (class-name
       class-parent
       slots-definitions
       constructor
       operator-overloads
       functions-definitions
       methods-definitions))
```

Una vez construido el AST para representar segmentos de código escritos en un DSL, se debe especificar cómo se escribe cada uno de sus nodos en los distintos lenguajes de salida. Esto se realiza en MGA, utilizando la función genérica que se presenta en la siguiente sección.

## 3.3. Generación de código en MGA

La generación de código en MGA está basada en la función genérica `generate-code`:

```
(defgeneric generate-code node language stream)
```

Los métodos que implementan esta función genérica se especializan en generar el código del nodo `node`, en el lenguaje de salida `language`, en el flujo

de salida `stream`. Definir cómo se escriben los programas representados en un AST, en un lenguaje de salida dado, requiere agregar un nuevo método a la función genérica `generate-code` por cada nodo del AST. Por ejemplo, a continuación se muestra cómo se puede implementar la generación de código del nodo `sum-node` en el lenguaje C#:

```
(defmethod generate-code ((node sum-node) (language csharp)
                                                    stream)
  (format stream "~a + ~a"
            (generate-code (left-hand node) language nil)
            (generate-code (right-hand node) language nil)))
```

Un llamado al método `sum-node` sería:

```
(generate-code (sum-node 5 3) csharp T)
```

En la construcción de estos métodos existe un conjunto de nodos del AST, cuya implementación es un patrón mecánico que puede ser automatizado. La generación de código de un nodo de este conjunto puede verse como una cadena de texto con variables, donde las variables son sustituidas por los resultados de la generación de código de cada uno de sus descendientes en una instancia del AST. En la implementación anterior de la función genérica `generate-code` para el nodo `sum-node` se puede observar este patrón, que fue encapsulado por ADOL* en el macro de nombre `gcode`. Con `gcode`, el código anterior sería:

```
(gcode sum-node csharp "~a+~a" left-hand right-hand)
```

Este macro `gcode` recibe como primer argumento el nodo cuyo código se desea generar (en este caso `sum-node`), como segundo, el lenguaje de salida (en este caso `csharp`), como tercero, una cadena válida para la función `format` [34] (en este caso `"~a+~a"`), y luego un número arbitrario de parámetros que representan los `slots` que deben ser incluidos en la generación de código (en este caso, `left-hand` y `right-hand`).

El uso de `gcode` deja de ser factible cuando la generación del código de un nodo no cumple exactamente el patrón anterior.

Un ejemplo puede verse en Python, donde la indentación es obligatoria para el correcto funcionamiento de los programas. Por ejemplo, el ciclo

`while` crea un nuevo contexto y las operaciones de su cuerpo requieren cierta indentación.

Si se desea generar el código del nodo `while` en Python se debe aumentar la indentación de todas las instrucciones que pertenezcan a su cuerpo. Esta situación no es posible expresarla usando el macro `gcode` definido en ADOL*, y por tanto la única solución es definir explícitamente el método `generate-code` para este nodo y en el cuerpo del mismo, aumentar la indentación.

Para ampliar la funcionalidad de `gcode`, en la siguiente sección se presenta otro enfoque que permite utilizarlo en diversos escenarios.

### 3.3.1. El macro gcode de MGA

La nueva estructura para el macro `gcode` permite modificar cualquier parte en la definición del mismo, de forma que sea posible describir la generación de código de cualquier nodo del AST. Por ejemplo, una implementación para escribir el código del nodo `while` en Python sería:

```
(gcode while python
       ("while (~a):~%" indent "~a~%" deindent)
       ((condition) (body)))
```

Para generar el código del nodo `while` en Python se utilizan dos listas. La primera, contiene cadenas de texto o código Lisp; y la segunda, listas de *slots*. Por cada cadena de la primera lista se crea una instrucción `format` con el `generate-code` de los *slots* en la lista correspondiente, y si no es una cadena, se considera código Lisp y se inserta en la macro expansión del `gcode`. En el caso del ejemplo anterior, la macro expansión sería:

```
(defmethod generate-code ((node while) (lang python) stream)
   (format stream "while (~a):~%"
       (generate-code (condition node) python nil))
   indent
   (format stream "~a"
       (generate-code (body node) python nil))
   deindent
   (format stream "~%" nil))
```

En este caso, los símbolos `indent` y `deindent`, aumentan o disminuyen la indentación de la generación de código escrita en el flujo `stream`.

Utilizar el macro `gcode` y encontrar características sintácticas similares en los lenguajes de programación, hace que extender cualquier Generador Multilenguaje sea un trabajo más sencillo.

### 3.3.2. Jerarquía de lenguajes

Generar el código de un programa hacia varios lenguajes requiere definir por cada nodo del AST su correspondiente implementación en todos ellos. Representar las propiedades comunes de la sintaxis de los lenguajes de salida y de los nodos de AST en jerarquías de clase puede simplificar este trabajo.

ADOL* y GEMURAL[4] implementan distintas jerarquías de clases para la sintaxis de los lenguajes y otra específicamente propia a sus respectivos DSLs. Un subconjunto de las propiedades sintácticas de los lenguajes se representa en ADOL* con las siguientes clases: la clase `Infix-Languaje`, que describe lenguajes que usan la notación infija para las expresiones aritméticas; la clase `Camel-Case-Language`, donde las variables y funciones se escriben en formato Camel-Case; `Indent-Languaje` y `Symbol-Separated-Language` representan los lenguajes con indentación (espacios en Python y C#) y separación de instrucciones (`';' en C#`), respectivamente.

Los lenguajes de programación de propósito general y DSLs para el trabajo matemático, como Matlab, poseen propiedades sintácticas similares para identificar los operadores aritméticos. Por ejemplo, las operaciones suma, multiplicación, división y comparación se representan con los símbolos `+, *, /, <, >,` . Estos operadores se pueden agrupar en una clase llamada `Basic-Operation-Language`.

La herencia múltiple presente en Common Lisp y las jerarquías para representar lenguajes similares, facilitan la generación de código fuente de bibliotecas implementadas en un DSL hacia nuevos lenguajes. Por ejemplo, si se define el lenguaje `C-like` y se hace heredar de las clases `Infix-Language`, `Indent-Language`, `Symbol-Separated-Languaje` y `Basic-Operation-Language`, significa que los lenguajes `C-like` son lenguajes con instrucciones en notación infija, separadas por algún símbolo y que contienen un conjunto de símbolos para representar operaciones aritméticas. Agregar los lenguajes `C#` y `Java` consistiría en definir la clase `C-like` como super clase de las clases que representan a estos lenguajes, puesto que `C#` y Java poseen características comunes a `C-like`. Luego, solo restaría definir los nuevos elementos

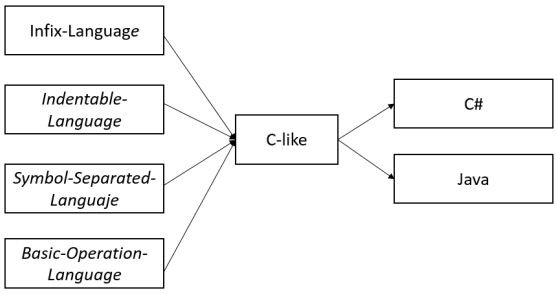


Figura 3.1: Jerarquía de lenguajes.

que estos lenguajes incorporan y realizarles la correspondiente generación de código hacia `C#` o `Java`.

La organización jerárquica de clases permite reutilizar código de los lenguajes y de nodos del AST definidos. El uso combinado de estructuras jerárquicas y el macro `gcode` constituyen las principales herramientas para realizar la generación de código a varios lenguajes. En la sección 4.2 se enuncian algunas ideas que permiten automatizar las características sintácticas de los lenguajes.

En este capítulo se presentaron las ideas fundamentales para implementar un Generador Multilenguaje en Common Lisp utilizando los macros `defnode` y `gcode`. En el próximo capítulo se presentan los principales recursos que permiten cambiar la estructura del código generado por dichos macros. Además se presentan las ideas que permiten automatizar características sintácticas de los lenguajes de salida.

# Capítulo 4

# Opciones de MGA

Los macros `defnode` y `gcode` utilizan la información de sus parámetros, para definición de clases, funciones constructoras, o realizar la generación de código. Por defecto, si no se especifican las opciones de los *slots*, las palabras clave, el Lambda List y el cuerpo de las funciones constructoras y de los métodos; esta información será rellenada automáticamente. Debido a que el usuario puede necesitar distintas funcionalidades en el dominio de su DSL, en la próxima sección se presentan otras funcionalidades del macro `defnode` que permiten modificar el código generado por él. Además, en la sección 4.2 se muestra cómo se pueden automatizar las propiedades sintácticas de los lenguajes de salida si se añade el código necesario en la macro expansión de `gcode`.

## 4.1. Opciones de defnode

En la definición de las clases y funciones constructoras existe un conjunto de propiedades que `defnode` completa con valores prefijados. El Lambda List de este macro tiene como primer parámetro el nombre de la clase a definir, como segundo, los nombres de sus super clases, como tercero, una lista con el nombre de sus *slots* y luego un conjunto de parámetros opcionales que permiten modificar el código generado por él. La definición del nombre de la clase y de la lista de herencia es igual a la que se debe realizar con el macro `defclass`, pero en la definición de *slots* y parámetros opcionales se presentan cambios. En las siguientes secciones se presentan estas modificaciones.

### 4.1.1. Slots

En el macro `defnode` cada elemento de la lista de nombres para los *slots* hace referencia a la declaración de un *slot* en la definición de la clase que se pretende crear, y estos pueden ser un símbolo o una lista. Si dicho elemento es un símbolo, la macro expansión de `defnode` rellena las opciones `accessor` e `initarg` con el mismo nombre del *slot* (valores por defecto); si es una lista, se deben definir las opciones de *slots* que se deseen y sus correspondientes valores.

En el siguiente ejemplo se muestra la definición de una clase con tres *slots*:

```
(defnode collection-definition-node ()
 (collection-name
  collection-type
 (collection-capacity
                     :initarg :capacity
                     :initform 100)))
```

En este caso, los dos primeros *slots* (`collection-name y collection-type`) serán rellenados con los valores por defecto, pero en el caso del tercero `collection-capacity`, se definen las propiedades `initarg` e `initform`, por lo que estos valores sustituirán los generados por defecto. Este comportamiento se puede apreciar en un fragmento de la macro expansión del código anterior:

```
(defclass collection-definition-node ()
     ((collection-name
            :accessor  collection-name
            :initarg  :collection-name
            :allocation instance
            :initform nil
            :documentation "empty")
      (collection-type
            :accessor  collection-type
            :initarg  :collection-type
            :allocation instance
            :initform nil
            :documentation "empty")
       (collection-capacity
```

```
:accessor collection-capacity
:initarg: capacity
    :allocation instance
:initform 100
:documentation "empty"))
```

El macro `defnode` emplea un conjunto de palabras clave para agregar o modificar el código que genera su expansión. Estas palabras clave son especificadas en el último argumento del Lambda List de dicho macro a través de parámetros nombrados, que se presentan a continuación.

### 4.1.2. Nombre de una función constructora

El nombre de una función constructora se especifica mediante la palabra clave `func-name`.
A continuación, se define la clase `collection-insert-node` y el nombre para su función constructora es `collection-insert`:

```
(defnode collection-insert-node () (name value)
    (:func-name collection-insert))
```

### 4.1.3. Lambda List de una función constructora

Por defecto, el Lambda List de una función constructora está formado por los nombres de las opciones `initarg` de los *slots* de la clase a instanciar. Estos se encuentran ordenados como fueron declarados en la definición de la clase:

```
(defnode collection-insert-node ()
                    (name value)
                    (:func-name collection-insert)))
```

Un fragmento de su macro expansión es:

```
(defun collection-insert (name  value))
```

Como en el Lambda List de la función constructora que generó `defnode` aparecen los parámetros `name` y `value`, solo se puede insertar un elemento a una instancia de `collection-insert-node`:

```
(collection-insert numbers-collection 7)
```

La palabra clave `lambda-list` permite definir el Lambda List de la función constructora. En este caso, el generado por `defnode` se sustituye por la nueva especificación de los parámetros.

Por ejemplo, si se desea que la clase `collection-insert-node` reciba un número arbitrario de argumentos, se define su Lambda List como se muestra a continuación:

```
(defnode collection-insert-node ()
                                (name value)
                                (:func-name collection-insert-range
                                 :lambda-list (name &rest value)))
```

Un fragmento de su macro expansión sería:

```
(defun collection-insert-range (name &rest value))
```

Y a partir de esta definición se podría instanciar `collection-insert-range` como:

```
(collection-insert-range  numbers-collection 4 5 6 7)
```

### 4.1.4. Cuerpo de una función constructora

El cuerpo de la función constructora se puede especificar con la palabra clave `ctr-body`:

```
(defnode collection-insert-node ()(name value)
(:ctr-body (make-instance 'collection-insert-node
               :name name)
               :value value))
```

En este caso, el valor de ctr-body puede ser cualquier fragmento de código Lisp válido.

## 4.2. Sintaxis de los lenguajes y gcode

En la sintaxis de los lenguajes de salida se encuentran características comunes que pueden ser automatizadas. Entre ellas están las que se muestran en la figura 3.1 (e identificadas en ADOL*). Para ejemplificar el proceso de automatizar una nueva característica, en esta sección se presenta cómo se puede automatizar la indentación de los lenguajes utilizando el macro `gcode`.

En los lenguajes de la familia C, para mantener un estándar de código legible, las instrucciones deben tener cierta indentación; en el caso de Python, es obligatoria. Representar esta característica requiere implementar una clase para identificar los lenguajes indentables (con indentación). Dicha clase puede almacenar el incremento del desplazamiento en cada nivel de su indentación y la indentación actual donde se debe comenzar a escribir en el parámetro `stream` de la función `generate-code`. Una posible implementación para esta clase puede ser:

```
(defnode indentable-language () (indentation indent-increment))
```

En la generación de código hacia los lenguajes indentables se pueden observar tres tipos de nodos del AST: los nodos que aumentan la indentación actual para generar el código de sus descendientes en un AST (`if`, `while` en `C#`, `Java` y `Python`), los que requieren indentación (`if`, `while` y asignación a variable en `C#`, `Java`, `Python`) y los que no (operaciones aritméticas en `C#`). La indentación de un nodo del AST no depende de él, sino del lenguaje, pues en un lenguaje puede ser indentable y en otro no.

A continuación, se muestra un escenario en Python, donde se puede observar la indentación de los nodos `sum-node`, `variable-assignmnet` y `while` del AST, correspondientes a la operación binaria suma, a la declaración asignación a variable y a la instrucción `while`, respectivamente:

```
if Y < 100:
    while X < 50:
        X = Y + 1
```

Como se puede observar en el ejemplo anterior, las instrucciones `if` y `while` aumentaron la indentación de los contextos que definen. Por ese motivo, la indentación de la instrucción `while` y de la asignación a la variable $X$ aumentaron. En el caso de la operación binaria suma $Y + 1$, no se necesitó un cambio en la indentación.

Para automatizar este proceso, se debe identificar por cada lenguaje indentable los nodos del AST que requieran indentación (nodos indentables); y para los nodos que aumentan la indentación de sus descendientes, especificar en su generación de código cúando aumentar la indentación actual del lenguaje.

Antes de generar el código de un nodo indentable se debe escribir en el flujo `stream` la cantidad de espacios (o tabulaciones) necesarios para alcanzar la indentación actual. Para ello, en Common Lisp, se puede utilizar el método extensor `:before` y especializarlo en la función genérica `generate-code` en el nodo y en el lenguaje indentable:

```
(defmethod generate-code: before ((node node-type)
                    (language indentable-language)
                                          stream))
```

Este método extensor puede ser encapsulado en un macro llamado `mark-node-as-indentable`. Los parámetros de este nuevo macro son el tipo del nodo a indentar y el tipo del lenguaje donde se quiere indentar. De esta forma, cuando se desee agregar un nuevo nodo indentable a un lenguaje, basta con realizar un llamado al macro `mark-node-as-indentable`:

```
(mark-node-as-indentable while language)
```

Para finalizar la automatización de la indentación, se debe implementar las funciones que aumenten y que disminuyan la indentación actual. Estas funciones se deben llamar cada vez que se desee cambiar el nivel actual de indentación.

MGA posee la función `recognize-pattern` para automatizar las características sintácticas de los lenguajes. Dicha función recibe dos parámetros: un símbolo `symbol` y un código `code` válido en Common Lisp. Una llamada a `recognize-pattern` le indica a `gcode`, que en cada llamado que se le realice debe sustituir en su macro expansión a `symbol` por `code`. De esta forma, si se realizan dos llamados a la función `recognize-pattern`; el primero con el símbolo `indent` y la función para aumentar la indentación actual, y el segundo con el símbolo `deindent` y la función para disminuir la indentación actual; generar el código para el nodo `while` en `Python` sería:

```
(gcode while python ("while (~a):~%" indent "~a~%" deindent)
       ((condition)(body)))
```

En este capítulo se presentaron las opciones configurables del macro `defnode`, y se ejemplificó cómo se pueden automatizar las características sintácticas de los lenguajes, a partir de la inserción de código en el macro `gcode`.

# Conclusiones

En este trabajo se identificaron las principales etapas en la construcción de un Generador Multilenguaje, se presentó una metodología para desarrollar estas aplicaciones en el lenguaje de programación Common Lisp, y se automatizó la escritura de código repetido y mecánico.

Para automatizar la escritura de código se crearon dos macros fundamentales: `defnode` que permite automatizar la definición de clases y sus funciones constructoras, y `gcode`, que simplifica la generación de código, y está inspirado en una funcionalidad similar existente en trabajos anteriores.

Utilizar los macros creados en MGA posibilita escribir, en pocas líneas de código, Generadores Multilenguajes que usualmente necesitan en su implementación grandes cantidades de líneas de código. Con la herramienta presentada en este proyecto, definir el DSL de un Generador Multilenguaje es realizar una selección adecuada de los nombres y de los campos para los elemento de su dominio, y su generación de código hacia los distintos lenguajes se convierte en encontrar patrones que se repiten, encapsularlos, y dedicarse solamente, a escribir el código diferente de cada uno de los elementos de su dominio.

# Recomendaciones

A partir de la investigación realizada se recomienda para trabajos futuros:

- Automatizar nuevas características comunes en la sintaxis de los lenguajes de programación.
- Añadir a los macros como posibles funciones constructoras.
- Agregar nuevas funciones o métodos en la macro expansión de `defnode`.

# Bibliografía


[1] *Leyendo documentos ODF (*.odt, *.ods, *.odp) con Apache OpenOffice.* [Online, accedido 12-3-2017]. `https://www.openoffice.org/es/por-que/why_odf.html`. (Citado en la página 6).

[2] *R, un lenguaje y entorno de programación para análisis estadístico.* [Online, accedido 12-3-2017]. `https://www.genbetadev.com/formacion/r-un-lenguaje-y-entorno-de-programacion-para-analisis-estadistico`. (Citado en las páginas 1 y 7).

[3] *Syntax diagrams - Backus Normal Form (BNF).* [Online, accedido 4-2-2017]. `https://ss64.com/docs/bnf.html`. (Citado en la página 1).

[4] Alayón, Antonio Rodríguez: *Generación Multilenguaje de Rutinas de Álgebra Lineal.* (Citado en la página 34).

[5] Alfred V. Aho, Jeffrey D. Ullman: *The theory of parsing, translation, and compiling*, volumen Volume I de *Prentice-Hall series in automatic computation.* Prentice-Hall, 1972, ISBN 9780139145568,0139145567,0139145648. `http://gen.lib.rus.ec/book/index.php?md5=54B3C50B63C656128EE3564FC1B7D4E3`. (Citado en la página 8).

[6] Barski, Conrad: *Land of Lisp.* No Starch Press, 2011. (Citado en las páginas 9, 10 y 14).

[7] Castillo, Manuel Alejandro Cabrera: *Una posibilidad real de utilizar Difenenciación Automática en cualquier lenguaje de programación.* (Citado en las páginas 1, 6, 13 y 24).

[8] Doug Brown, John Levine, Tony Mason: *lex and yacc.* O'Reilly Media, segunda edición, 1995. (Citado en la página 8).

[9] Driscoll, Brian: *Entity Framework 6 Recipes*. Apress, segunda edición, 2013. (Citado en la página 7).

[10] FREEMAN, ADAM: *Pro ASP.NET Core MVC*. Apress, sexta edición, 2016. (Citado en la página 7).

[11] Gansner, Emden, Eleftherios Koutsofios y Stephen North: *Drawing graphs with dot*. 2006. (Citado en la página 7).

[12] Graham, Paul: *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, 1st edición, 1993. (Citado en las páginas 9, 10, 11, 14 y 15).

[13] Graham, Paul: *ANSI Common LISP*. Prentice Hall series in artificial intelligence. Prentice Hall, 1st edition edición, 1996. (Citado en las páginas 9 y 15).

[14] Griewank, Andreas, y Andrea Walthe: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2da edición, 2008. (Citado en las páginas 1 y 30).

[15] Helander, Markus Voelter; Sebastian Benz; Christian Dietrich; Mats: *DSL Engineering. Designing, Implementing and Using Domain-Specific Languajes*. 2010-2013. (Citado en las páginas 1 y 8).

[16] Helmut Kopka, Patrick W. Daly: *A Guide to LATEX: Document Preparation for Beginners and Advanced Users (3rd Edition)*. Addison-Wesley Professional, 1999. (Citado en la página 6).

[17] Hoyos, Jorge Luis: *Un Lenguaje para representar Modelos Matemáticos en Lisp*, 2017. (Citado en las páginas 2, 6, 13 y 24).

[18] Hoyte, Doug: *Let Over Lambda : 50 years of lisp*. Doug Hoyte/HCSW and Hoytech production, 2008. (Citado en la página 14).

[19] Inc., Adobe Systems: *PDF Reference Version 1.7*. sexta edición, 2006. (Citado en la página 1).

[20] J., Adam: *Addison-Wesley Professional Ruby Series Rails Plugins: Extending Rails Beyond the Core*. 2006. (Citado en las páginas 7 y 9).

[21] L., Bettini: *Implementing domain-specific languages with Xtext and Xtend*. Packt, 2013. (Citado en la página 1).

[22] Lamkins, David B.: *Successful Lisp.How to understand and use Common Lisp*. bookfix.com, 2004. (Citado en la página 12).

[23] L.G., DeMichiel: *The Common Lisp object system. An overview*. (Citado en las páginas 9 y 12).

[24] M, Mernik: *Formal and Practical Aspects of Domain-Specific Languages Recent Developments*. Premier Reference Source. IGI Global, 2013. (Citado en las páginas 1, 7 y 8).

[25] Meyer, Bertrand: *Object Oriented Software Construction*. Prentice Hall, 2nd edición, 2000. (Citado en la página 8).

[26] Mosquera, Camila Pérez: *Primeras aproximaciones a la Búsqueda de Vecidad Infinitamente Variable*, 2017. (Citado en las páginas 2, 6, 13 y 24).

[27] Palm, William J: *A concise introduction to MATLAB*. McGraw-Hill, 2008. (Citado en las páginas 1 y 7).

[28] Parr, Terence: *The Definitive ANTLR 4 Reference, 2nd Edition*. Pragmatic Bookshelf, 2nd edición, 2013. (Citado en la página 8).

[29] Powell, Thomas: *HTML: The Complete Reference, Second Edition*. The McGraw-Hill Companies, May,June 1999. (Citado en las páginas 1 y 6).

[30] Riesco, Claudia Naveda; Alberto Cortez; Germán Montejano; Daniel: *Lenguaje Específico de Dominio para Aplicaciones de Negocios*. (Citado en la página 7).

[31] R.J., Chassell: *Texinfo 4.0*. 2000. (Citado en la página 1).

[32] Rossun; Dan Bornstein; Tom Nurkkala; Kyle Ferrio, PhD Guido van: *Languaje Implementation Patterns. Create Your Own Domain Specific and General Programming Languages*, volumen 2010-1-4. tercera edición, 2009. (Citado en las páginas 1, 7 y 8).

[33] S.E., Keene: *Object-Oriented Programming in Common Lisp*. 1989. (Citado en las páginas 7, 9, 10, 11, 12, 13 y 14).

[34] Seibel, Peter: *Practical Common Lisp*. APRESS / SPRINGER-VERLAG, 1980. (Citado en las páginas 10, 11, 13 y 32).

[35] Stallman, Richard: *Emacs manual*. 1997. (Citado en las páginas 1 y 5).

[36] Steele, Guy: *Common Lisp, The Language, 2nd Edition*. segunda edición, 1990. (Citado en la página 13).

[37] Touretzky, David S.: *Common Lisp. A Gentle Introduction To Symbolic Computation*. Benjamin-Cummings Pub Co, 1989. (Citado en la página 9).

[38] Wills, Steve Cook; Gareth Jones; Stuart Kent; Alan Cameron: *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007. (Citado en la página 1).

[39] Xie, Yihui: *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman Hall CRC R Series. Chapman Hall CRC, 2016. (Citado en las páginas 1 y 5).