DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCION I.- TESIS

TRABAJO PROFESIONAL

"MODELO CONCEPTUAL PARA LA IMPLEMENTACIÓN DE LENGUAJES DE PROGRAMACIÓN NATURALÍSTICOS DE PROPÓSITO GENERAL"

QUE PARA OBTENER EL GRADO DE:

DOCTOR EN CIENCIAS DE LA INGENIERÍA

PRESENTA:
OSCAR PULIDO PRIETO

DIRECTOR DE TESIS:
DR. ULISES JUÁREZ MARTÍNEZ

ORIZABA, VER. MÉXICO

DICIEMBRE 2019

FECHA: 02/12/2019
DEPENDENCIA: **POSGRADO**
ASUNTO: **Autorización de Impresión**
OPCIÓN: **I**

C. **OSCAR PULIDO PRIETO**
CANDIDATO A GRADO DE DOCTOR EN:
CIENCIAS DE LA INGENIERIA

De acuerdo con el Reglamento de Titulación vigente de los Centros de Enseñanza Técnica Superior, dependiente de la Dirección General de Institutos Tecnológicos de la Secretaría de Educación Pública y habiendo cumplido con todas las indicaciones que la Comisión Revisora le hizo respecto a su Trabajo Profesional titulado:

"MODELO CONCEPTUAL PARA LA IMPLEMENTACION DE LENGUAJES DE PROGRAMACION NATURALISTICOS DE PROPOSITO GENERAL".

Comunico a Usted que este Departamento concede su autorización para que proceda a la impresión del mismo.

A T E N T A M E N T E

MARIO LEONCIO ARRIOJA RODRIGUEZ
JEFE DE LA DIV. DE ESTUDIOS DE POSGRADO

EDUCACIÓN | TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE ORIZABA
DIVISIÓN DE ESTUDIOS PROFESIONALES

"2019, Año del Caudillo del Sur, Emiliano Zapata"

FECHA : **25/11/2019**

ASUNTO: **Revisión de Trabajo Escrito**

C. MARIO LEONCIO ARRIOJA RODRIGUEZ
JEFE DE LA DIVISION DE ESTUDIOS
DE POSGRADO E INVESTIGACION.
P R E S E N T E

Los que suscriben, miembros del jurado, han realizado la revisión de la Tesis del (la) C. :

OSCAR PULIDO PRIETO

la cual lleva el título de:

"MODELO CONCEPTUAL PARA LA IMPLEMENTACION DE LENGUAJES DE PROGRAMACION NATURALISTICOS DE PROPOSITO GENERAL".

Y concluyen que se acepta.

A T E N T A M E N T E

PRESIDENTE	: **DR.**	**ULISES JUAREZ MARTINEZ**	FIRMA
SECRETARIO	: **DR.**	**GINER ALOR HERNANDEZ**	FIRMA
PRIMER VOCAL	: **DR.**	**GUILLERMO CORTES ROBLES**	FIRMA
SEGUNDO VOCAL	: **DR.**	**CUAUHTEMOC SANCHEZ RAMIREZ**	FIRMA
TERCER VOCAL	: **DR.**	**GALO RAFAEL URREA GARCIA**	FIRMA
VOCAL SUP.	: **DRA.**	**MARIA KAREN CORTES VERDIN**	FIRMA

EGRESADO(A) DEL DOCTORADO EN **CIENCIAS DE LA INGENIERIA**

OPCION: **I** **Tesis**

Agradecimientos

A mi madre, por todo el apoyo tanto moral como material que me brindó durante el transcurso de estos años, realmente no tengo palabras para expresarte mi agradecimiento, así que sólo pongo esto para llenar el espacio.

A mi amiga Luisa, quien me apoyó desde 2016 con pruebas no oficiales, con su proyecto de residencias en 2017, actualmente su tesis de maestría y además, durante todos estos años se comprometió a mantener mi cordura en lo referente a asuntos no académicos. Gracias por arriesgarte a batallar con SN, por incomodar gente, por alejar problemas y sobre todo, por siempre confirmar mis observaciones tanto académicas, como personales.

A la maestra Beatriz A. Olivares Zepahua por sus comentarios sobre SN, su retroalimentación respecto a la parte industrial fue muy útil, a pesar de centrarse sólo en la parte de requisitos y viendo sólo el ejemplo de base de datos.

A la maestra María Jacinta Martínez Castillo, a Sergio "Checo" Salgado Orozco, a Alfonso "Poncho" Flores Leal y a Jorge Luís "el Pani" Villa Paniagua, por facilitarme el apoyo con sus alumnos para realizar las encuestas sobre la expresividad de SN.

A todos aquellos que participaron en las encuestas para medir la expresividad de SN.

A todos aquellos que no creyeron en este proyecto y que directa o indirectamente me motivaron para mejorarlo.

A Roman Knöll por su platica "desmotivadora" en junio de 2017, que un experto diga "tu propuesta es suficientemente realista para un doctorado" en un momento difícil te ayuda a continuar. *Vielen Dank für deine "demotivierenden" Worte (Sorry if I misspelled it, if you ever read this).*

A *CONACyT* y al *TecNM* por los apoyos otorgados para el desarrollo de este proyecto.

Índice general

Índice de figuras

Índice de tablas

Resumen

La expresividad, que consiste en la capacidad de un lenguaje de programación para describir las ideas que se representan por medio de las instrucciones, pero el paradigma de programación limita dicha expresividad. Aunque en la actualidad los lenguajes que se basan en los paradigmas de la programación orientada a objetos y la programación orientada a aspectos resolvieron gran parte de los problemas de abstracción, aún carecen de los medios necesarios para realizarlo de forma suficientemente expresiva. Surgieron diversos paradigmas para solventar dicho problema desde la perspectiva de dominios específicos o formales, pero sin considerar que muchas de las abstracciones se obtienen de diálogos, entrevistas o documentos que se generan en lenguaje natural. Algunos autores tomaron esto en cuenta y desarrollaron herramientas de apoyo para el desarrollo de software por medio del uso de lenguaje natural. En esta tesis se presentan los conceptos que fundamentan la programación naturalística; así como una revisión de los diversos trabajos relacionados que se reportan, tanto de las herramientas que emplean algún nivel de lenguaje natural; hasta lenguajes de dominio específico que poseen un nivel de expresividad similar a los lenguajes naturales. Como aportación, se propone un modelo conceptual que define los elementos que se requieren para el diseño e implementación de lenguajes de programación que posean elementos de los lenguajes naturales y por tanto, reduzcan la brecha entre los dominios del problema y la solución.

Abstract

Expressivity is the ability of a programming language to describe the ideas represented by instructions, but the programming paradigm limits that expressiveness. Although today the languages based on the paradigms of object-oriented programming and aspect-oriented programming solved much of the abstraction problems, they still lack the means to do so in a sufficiently expressive way. Different paradigms emerged to solve this problem from the perspective of specific or formal domains, but disregarding that many abstractions are obtained from dialogues, interviews or documents generated in natural language. Some authors took this into account and devised support tools for software development through the use of natural language. This thesis presents the concepts that support naturalistic programming; as well as a review of the diverse related works that are reported, from the tools that use some level of natural language; to domain-specific languages that possess a level of expressiveness similar to natural languages. As a contribution, a conceptual model is proposed that defines the elements required for the design and implementation of programming languages that possess elements of natural languages and therefore, reduce the gap between the problem domains and the solution.

Introducción

Desde hace varias décadas, los programadores buscan medios para emplear lenguajes naturales y de esta forma desarrollar sistemas. La mayor limitante para lo anterior es que las computadoras poseen una capacidad restringida para detectar y resolver ambigüedades, ya que esto depende en gran medida del proceso cognitivo de los interlocutores. Para resolver lo anterior, diversos autores definieron versiones controladas y formalizadas de un lenguaje natural de modo que una computadora los procese sin ambigüedad, tales como *Pegasus* o *Attempto Controlled English*, por mencionar algunos.

En los últimos años diversos paradigmas permitieron un desarrollo de software más eficiente, pero aún se observan limitantes durante las tareas de mantenimiento y evolución del software. Una de ellas consiste en que los programadores generalmente carecen de la disciplina o tiempo necesarios para documentar su código, de modo que si otro programador, o él mismo dan mantenimiento a código, se requiere invertir tiempo y esfuerzo para indagar qué hace cada sección del mismo. Otra limitante consiste en la formalidad rigurosa del código, lo que implica que el programador debe adaptar las ideas de los clientes al lenguaje o inclusive al paradigma. Se prestó atención a la formalidad por medio de lenguajes de dominio específico que ofrecen la expresividad necesaria, mientras que el problema de la evolución se abordó con nuevos paradigmas que permiten abstracciones más flexibles.

El reto principal consiste en que se requiere de abstracciones suficientemente genéricas pero a la vez suficientemente expresivas para que se implementen en lenguajes de programación de propósito general. La programación orientada a objetos permitió avances significativos, ya que las abstracciones se encapsulan sin considerar el dominio, pero con el inconveniente de que ciertos elementos se encuentran dispersos. La programación orientada a aspectos permite encapsular dichos elementos, pero ninguno de los dos paradigmas resolvió el problema de la falta de expresividad de los lenguajes de programación tradi-

cionales respecto al lenguaje natural. El paradigma de la programación intencional busca resolver ambos problemas, pero para esto emplea un conjunto de lenguajes de dominio específico que son expresivos respecto a su dominio, pero no respecto al lenguaje natural.

Así que mientras los lenguajes de dominio específico poseen expresividad, pero se enfocan a un dominio particular, los objetos y aspectos cubren de forma relativamente eficiente todos los dominios, pero carecen de la expresividad de los primeros. La programación naturalística parte de la idea de que en los lenguajes naturales las cosas a las que se hace referencia no se limitan a abstracciones, sino también a la propia estructura del lenguaje, de modo que el programador escriba código que haga referencia a objetos, funciones, componentes e inclusive sobre aspectos. Por tanto, las abstracciones naturalísticas requieren de una capacidad de reflexividad que no se limite a variables, sino también a la posición y el tiempo en el que se defina la información, de modo que el código naturalístico sea reflexivo en términos estructurales y temporales. Por reflexividad estructural se entiende que el código posee soporte para identificar y hacer referencia a otras secciones de sí mismo, mientras que la reflexividad temporal indica el momento en el que se define o utiliza un valor. La programación naturalística ofrece un incremento en la expresividad más allá de los paradigmas actuales. En este documento se presenta un modelo que permite crear lenguajes de programación de propósito general con características similares a los lenguajes naturales, pero formalizados de modo que la ambigüedad se elimine para describir abstracciones en términos de elementos propios de los lenguajes naturales.

En el capítulo 1 se presentan, los objetivos, la hipótesis, el planteamiento del problema y la justificación del proyecto. En el capítulo 2 se presentan los fundamentos teóricos que sustentan este proyecto. En el capítulo 3 se describen los trabajos relacionados a este proyecto. En el capítulo 4 se presenta el modelo que se definió y a partir del mismo se diseñó e implementó el lenguaje naturalístico SN, se describe al prototipo, su gramática, proceso de compilación y se presentan escenarios de prueba. En el capítulo 5 se presentan las conclusiones y recomendaciones que se derivan de este trabajo.

Capítulo 1

Antecedentes

A continuación se presentan el problema a resolver, los objetivos y la justificación del porqué de la investigación que se presenta en este documento.

1.1. Objetivos

En esta sección se presenta el objetivo general y los objetivos específicos.

1.1.1. Objetivo general

Desarrollar un modelo conceptual naturalístico para implementar lenguajes de programación de propósito general por medio de deixis y que permita el desarrollo de abstracciones expresivas respecto al idioma inglés.

1.1.2. Objetivos específicos

- Realizar un análisis de los lenguajes que integran elementos naturalísticos a su gramática.
- Definir un modelo conceptual para el desarrollo e implementación de lenguajes naturalísticos.
- Diseñar una gramática que se base en el modelo conceptual naturalístico.

- Modelar la gramática por medio del lenguaje funcional Haskell.
- Validar que la definición de la gramática es coherente con el modelo que se propone.
- Generar una interfaz de programación de aplicaciones mínima para la implementación del lenguaje.
- Diseñar un editor que permita trabajar con el lenguaje.
- Realizar pruebas y depuración del compilador del lenguaje.
- Identificar escenarios para mostrar las características del lenguaje naturalístico.

1.2. Hipótesis

Un modelo naturalístico con soporte adecuado de temporalidad y reflexividad permitirá implementar lenguajes de propósito general donde las abstracciones permitan modelar conceptos del dominio del problema.

1.3. Planteamiento del problema

Los lenguajes de programación naturalísticos que se reportan se enfocan a que el programador defina instrucciones por medio del lenguaje natural, pero no reportan qué elementos se consideran necesarios para diseñar otros lenguajes naturalísticos porque dada la complejidad de los lenguajes naturales, los autores restringen sus trabajos a resolver problemas de dominios particulares. Dichas restricciones son a nivel gramatical por medio de una sintaxis limitada, como en el caso de NLC [Ballard y Lusth, 1983], o de análisis al definir un contexto particular para el texto, como en el caso de NLCI [Landhäußer, Weigelt, y Tichy, 2016].

Los lenguajes naturalísticos que se enfocan a dominios particulares resuelven los problemas de forma expresiva desde la perspectiva del lenguaje natural gracias a que limitan su alcance al dominio correspondiente y por tanto, eliminan la ambigüedad de las oraciones debido a que su enfoque es hacia un contexto particular. Con base en esto, establecer un contexto para las abstracciones al momento de programarlas permite a los desarrolladores

crear código sin importar el dominio del problema. Pero, para lograr lo anterior se requiere de instrucciones flexibles, donde a diferencia de la programación orientada a objetos, se dé peso no sólo a los sustantivos, sino también a los verbos, adjetivos, sintagmas e inclusive al contexto particular de cada oración, de modo que un lenguaje naturalístico de propósito general consiste en instrucciones flexibles pero controladas de forma que la ambigüedad se elimine o, en su defecto, se reduzca al punto en el que no se requieran de algoritmos complejos para resolverla.

Por otro lado, los lenguajes naturales poseen la limitante de ser poco descriptivos cuando se trata de conceptos formales, algo que descartaron los autores de los lenguajes naturalísticos que se reportan debido a que sus lenguajes y harramientas se centran en dominios particulares, de modo que la gramática mantiene el soporte para los formalismos del dominio. Por el contrario, se espera que un lenguaje naturalístico de propósito general permita integrar no sólo las propiedades de los lenguajes naturales, sino también los formalismos de cualquier dominio, algo que por sí mismo un lenguaje de programación no posee. Por tanto, se requiere de un mecanismo que permita establecer dichos formalismos sin reducir la expresividad de los lenguajes de programación respecto al lenguaje natural, de modo que el código se estructure de forma similar a los documentos científicos, donde los formalismos se separan de forma clara, ya sea cambiando el tipo de letra o en un espacio separado. Un mecanismo como el que se describe permite que un lenguaje de programación posea una gramática embebida para establecer formalismos, que normalmente se especifican utilizando lenguajes de dominio específico.

Definir un lenguaje de programación naturalístico de propósito general con una gramática formal implica definir características reproducibles en otros lenguajes naturalísticos. Lo anterior deriva en una familia de lenguajes naturalísticos con elementos comunes que, además, requieren de un proceso de ingeniería propio, dado que al reducir la brecha entre los dominios del problema y la solución, se espera que se requieran menos transformaciones en los requerimientos y por tanto, el ciclo de desarrollo contenga una menor cantidad de pasos. De este modo, una familia de lenguajes naturalísticos con sus propias técnicas y herrramientas de ingeniería da como resultado un paradigma de programación.

Esta tesis busca resolver las siguientes preguntas:

- ¿Un modelo naturalístico resuelve parte del problema de acercar el dominio del problema y el de la solución?

- ¿Un lenguaje naturalístico de propósito general resuelve problemas independientemente del dominio?
- ¿Un lenguaje naturalístico justifica un nuevo paradigma?
- ¿Un lenguaje naturalístico de propósito general es más eficiente para definir formalismos de un domino?
- ¿Un lenguaje naturalístico es ortogonal a otros paradigmas o su definición imposibilita dicha combinación?

1.4. Justificación

Un modelo conceptual de programación sirve como base para establecer los fundamentos para el diseño de lenguajes de programación naturalísticos, esto a su vez permite que los programadores definan abstracciones que se asemejen a la forma en que se redactan los requisitos. A partir de dichas abstracciones, los programadores facilitarán a los expertos del dominio (en especial a los no programadores) involucrarse de una forma más directa en el proceso de desarrollo de software. Lo anterior se debe a que un lenguaje naturalístico posee una sintaxis más cercana a los lenguajes naturales y por tanto, es más expresivo al tiempo que mantiene la formalidad para evitar ambigüedades.

Eliminar la brecha entre los dominios del problema y la solución en el desarrollo de software es una meta que se busca desde la década de 1960 [Sammet, 1966], algo que a día de hoy sigue sin lograrse, esto se debe a la capacidad limitada de las computadoras para resolver la ambigüedad innerente de los lenguajes naturales. Con el paso de los años, surgieron lenguajes y paradigmas que buscan incrementar la expresividad tales como la programación orientada o objetos, la programación funcional o la programación procedural.

Los nuevos paradigmas y lenguajes permitieron que el desarrollo de software se extendiera al facilitar la creación de programas complejos por medio de instrucciones más cercanas a lenguajes conocidos por los programadores, en lugar del uso de lenguaje máquina o lenguajes ensambladores, que son más cercanos a la computadora. Lo anterior sirvió no sólo como un apoyo a científicos y militares, sino que en la actualidad, su campo de aplicación abarca prácticamente cualquier disciplina. Dicha masificación implica que

los programadores trabajen con multitud de formalismos de dominios tan dispares entre sí como la medicina y la arquitectura. Además, dado que los expertos del dominio generalmente desconocen los formalismos de los lenguajes de programación, los requisitos se definen en lenguaje natural y posteriormente se traducen al paradigma de programación que corresponde al lenguaje en el que se desarrollará el sistema, lo que conlleva a una dispersión de requisitos.

Al poseer instrucciones más cercanas al lenguaje natural, un lenguaje naturalístico reduce la brecha entre los dominios del problema y la solución, de modo que los analistas realicen la especificación de requisitos pensando en una implementación naturalística. Con base en la implementación naturalística, el documento de requisitos poseerá una estructura donde se coloquen oraciones descriptivas, pero al mismo tiempo procesables por la computadora dado que son parte de la gramática del lenguaje naturalístico. Por otro lado, un nivel de expresividad en el código que lo acerque a la autodocumentación facilita el análisis de código durante las tareas de mantenimiento y evolución de software, lo que implica que otros programadores analicen el código y lo comprendan con mayor rapidez y por tanto, que se reduzca el tiempo y dinero que se emplee en dichas tareas.

1.5. Aportación

La aportación a la ciencia consiste en un modelo que permitirá diseñar lenguajes de programación que posean elementos de los lenguajes naturales para mejorar el nivel de expresividad.

El modelo define los elementos que se requieren para que un lenguaje de programación se considere naturalístico, de modo que un lenguaje naturalístico permitirá el desarrollo de aplicaciones más flexible. Al integrar elementos de los lenguajes naturales, los lenguajes naturalísticos poseerán un nivel de expresividad más cercano al inglés.

Capítulo 2

Marco teórico

Los primeros programas se escribieron por medio de la numeración binaria, de modo que desde la perspectiva del programador, su expresividad era nula; para solucionarlo, se buscó la forma de dar sentido a dichas instrucciones por medio de un lenguaje intermedio, de este modo nacieron los lenguajes ensambladores que permitían a los programadores crear programas con instrucciones donde manipulaban variables y registros que posteriormente se traducían a código ejecutable por la computadora. Los lenguajes ensambladores proporcionan cierto nivel de expresividad, pero fuerzan al programador a descomponer las ideas para convertirlas en instrucciones ejecutables. Durante la tercera generación de los lenguajes de programación, se empezó a programar por medio de verbos, que son conceptos de los lenguajes naturales donde un verbo indica una tarea concreta que el programa debe realizar. Posteriormente, los lenguajes orientados a objetos utilizaron sustantivos a modo de identificadores de instancias, ya que los objetos permiten definir abstracciones del mundo real que realizan tareas específicas (operaciones que son verbos); de modo que se dio a los programadores la capacidad para trabajar con oraciones simples [Booch y cols., 2007].

La programación naturalística se relaciona con otros paradigmas tales como la programación orientada a aspectos (POA) y la programación intencional, además de tomar de los lenguajes naturales, elementos como la deixis y el uso de sintagmas complejos para ofrecer un nivel de expresividad que se acerca más a un lenguaje natural. En esta sección se hace una descripción de dichos conceptos.

2.1. Elementos computacionales

En esta sección se presentan los conceptos que se relacionan con la parte computacional de la tesis.

2.1.1. Modelos de programación

En [Van Roy y Haridi, 2004], Van Roy y Haridi describieron a los modelos de programación como un conjunto de propiedades que permiten diseñar lenguajes de programación de un paradigma determinado. los autores mencionaron que el modelo funcional es el que posee los elementos mínimos para definir un lenguaje de programación con las propiedades de una máquina universal de Turing. Un lenguaje funcional puro carece de efectos de borde, posee transparencia referencial, se basa en recursividad para resolver operaciones repetitivas y posee sistemas de tipos para trabajar con cálculo lambda, de modo que se pasan como parámetros de otras funciones o asignarse a un identificador. Además, los lenguajes funcionales únicamente emplean estructuras de datos inmutables tales como listas, registros y tuplas, que para una implementación eficiente requieren de evaluación perezosa; a diferencia de otros paradigmas, donde dichas estructuras son mutables. Estas características hacen que un lenguaje funcional sea conciso y expresivo desde una perspectiva matemática, lo que implica un trabajo fuerte en la transformación de los requisitos.

En [Booch y cols., 2007], Booch et. al. mencionaron que el modelo de objetos parte de los conceptos de abstracción, encapsulación, modularidad y jerarquía. Booch define a la abstracción como la capacidad para tomar los detalles importantes de un elemento necesario en el sistema desde la perspectiva de un dominio particular, mientras que se dejan de lado aquellos elementos sin relevancia para el dominio. Van Roy describe a la abstracción como un elemento que se deriva de la programación de orden superior, donde se utilizan registros que ocultan datos, al tiempo que dejan a la vista las funciones. Booch define la encapsulación como la capacidad para ocultar los detalles internos de un objeto, de modo que sólo se acceda a ellos de forma controlada; para Van Roy, la encapsulación no es más que una función que cifra o descifra un valor, de modo que sólo se acceda a él por medio de dichas funciones. Booch describe la modularidad como la capacidad de un programa para dividirse en módulos que se compilan por separado, pero que poseen relación entre sí. Por último, Booch define la jerarquía como una clasificación u ordenación

de abstracciones y describe dos tipos de jerarquía: la herencia, donde una clase "es un" tipo de otra, lo que implica especialización o generalización; y la agregación, donde una clase es "parte de" otra clase, lo que implica que una clase posee atributos que a su vez son clases. Mientras la herencia permite reutilización, la agregación permite acoplar y reemplazar a los elementos que formen parte de la clase.

En [Kiczales y Hilsdale, 2001], Kiczales y Hilsdale describieron a la POA como un paradigma complementario a POO dada su capacidad para encapsular los elementos que quedan dispersos entre objetos. El modelo de puntos de unión que presentan los autores describe al punto de unión (*join point*) como un punto de ejecución identificable en el programa donde es posible agregar una funcionalidad adicional. Se presenta al corte (*pointcut*) como un mecanismo para especificar y cuantificar puntos de unión. Por último, se define al aviso como código adicional que se agregará en el punto de unión que se identificó, dicho aviso se ejecuta antes, después o en lugar del punto de unión.

En [Lopes, Dourish, Lorenz, y Lieberherr, 2003], Videira Lopes et. al. abrieron una discusión sobre los elementos que se observan en los lenguajes naturales y que se agregan a la POA para describir abstracciones con un nivel de expresividad más cercano al inglés, al tiempo que se mantinene la formalidad de un lenguaje de programación tradicional. Los autores mencionaron que en las relaciones anafóricas se observan tres tipos de referencias: las referencias estructurales, las referencias temporales y las referencias reflexivas. Los lenguajes de programación tradicionales permiten un soporte limitado para emplear referencias estructurales y reflexivas, pero carecen de los mecanismos para trabajar con referencias temporales. Por otro lado, los autores mencionaron que *AspectJ* utiliza una forma rudimentaria de referencias temporales, pero carece de la capacidad para describir información que proviene del contexto, donde una referencia reflexiva y temporal como *the last operation* ayuda a definir mecanismos de expresividad más cercanos al lenguaje natural.

2.1.2. Programación naturalística

La programación naturalística es un concepto que Karl Lieberhherr define como el uso de elementos de los lenguajes naturales para el diseño de lenguajes de programación más expresivos y cercanos a un lenguaje natural [Lopes y cols., 2003]. En dicho artículo se abordan las ventajas y limitantes tanto de los lenguajes naturales, como de los lenguajes de

programación actuales si se busca emplearlos en el desarrollo de software. Como solución los autores proponen un término medio; es decir, un lenguaje de programación formal que posea elementos de los lenguajes naturales tales como las referencias anafóricas.

"Naturalistic" es un término que el Oxford English Dictionary [Press, 2016] define como:

> *"Copying the way things are in the natural world".*

De la definición anterior se infiere que algo naturalístico es artificial y se contrapone a lo natural, pero se diseña de forma que se asemeje a esto último. Los lenguajes de programación no se diseñaron pensando en asemejarse a los lenguajes naturales, sino a modelos formales que permitan indicar instrucciones a la computadora.

Otro detalle a tomarse en cuenta es que según [Clark, Murray, Harrison, y Thompson, 2010], un lenguaje de programación que se base en una representación formal de un lenguaje natural pertenece a una de las dos categorías siguientes: formalista o naturalista.

Un lenguaje naturalista es aquel que describe una forma más simple un lenguaje natural; mientras que un lenguaje formalista, es una representación determinista con una especificación más formal del lenguaje natural. En los lenguajes controlados de tipo naturalista la ambigüedad se mantiene en menor medida. Las herramientas que se basan en lenguajes naturalistas procesan el texto por medio de múltiples interpretaciones para devolver la mejor, pero bajo la premisa de que el lenguaje será un subconjunto de un lenguaje natural completo. Un lenguaje naturalista se enfoca a que una lengua sea más entendible por una computadora y permite una traducción heurística de la ambigüedad, de modo que se extienda para eventualmente dar un soporte a un lenguaje natural completo.

Por otro lado, los lenguajes controlados de tipo formalista se asemejan más a un lenguaje natural formal que se definió con claridad, es predecible y más fácil de utilizar que un lenguaje formal. Un lenguaje formalista se enfoca a que la lógica sea más entendible por los programadores, de modo que se evita el indeterminismo ya que se requiere que dichos lenguajes se traduzcan a un lenguaje formal de forma clara y predecible. Posee una representación única para las oraciones y un significado único para cada palabra. Un lenguaje naturalista es más natural y fluido para el usuario, pero menos controlable dado que el usuario no siempre predice la ambigüedad. Por otro lado, un lenguaje formalista es más fácil de controlar, pero su lectura será menos natural y se requerirá que el usuario entienda el dominio al que se enfoca el lenguaje.

Por último, las semánticas programáticas son las relaciones procedurales que se infieren de la estructura lingüística [Liu y Lieberman, 2005b]. Se menciona que los verbos se asocian a estructuras de datos, los verbos con funciones y los adjetivos con propiedades [Mihalcea, Liu, y Lieberman, 2006]. Además, los principales principios de las semánticas programáticas: características sintácticas, que consisten en la diferenciación entre sustantivos, verbos la forma en que la mezcla de ambas proporciona un semántica a una expresión; características procedurales, que consiste en la expresión de reglas condicionales, iteración o recursividad, a su vez se dividen en subjuntivos o condicionales, posibilidades (lógica modal) y "cuando", que sirve como base para crear expresiones condicionales; características relacionales y de teoría de conjuntos, que son construcciones gramaticales que permiten describir una serie de procesos y ciclos; y equivalencia representacional, que consisten en representar entidades por medio de una narrativa.

2.1.3. Programación orientada a aspectos

No obstante sus beneficios, la programación orientada a objetos dejó sin resolver diversos problemas, entre ellos el cómo encapsular asuntos que se encuentran dispersos en el sistema y que dificultan el mantenimiento y evolución del software. La solución a esto llegó con la programación orientada a aspectos, que permite encapsular dichos asuntos de modo que su mantenimiento se realice de forma eficiente y que además facilita el desarrollo de aplicaciones al separar la funcionalidad principal de las tareas complementarias [Kiczales y Hilsdale, 2001].

Pero los aspectos poseen una limitante que se deriva de la forma en que se expresan los cortes, que en el caso de AspectJ es por medio de un patrón de firma que es expresivo de acuerdo a la sintaxis de la firma de los métodos de Java, lo que da como resultado cortes poco tolerantes a cambios cuando se requiere agregar o modificar la funcionalidad de métodos particulares; ante dicho problema, se menciona que los aspectos son "frágiles". Como solución, diversos autores proponen mecanismos como los eventos de ECaesarJ [Nunez y Gasiunas, 2009] y Ptolemy [Dyer, Rajan, y Cai, 2012]; o el uso de lenguajes orientados a aspectos basados en la programación lógica como ALPHA [Ostermann, Mezini, y Bockisch, 2005] y el lenguaje de aspectos que se diseñó para Smalltalk que se plantea en [Gybels y Brichau, 2003].

Por otro lado, en [Chitchyan, Rashid, Rayson, y Waters, 2007] se busca simplificar las tareas de modelado de sistemas por medio de descripciones de requisitos. La ingeniería de requisitos orientada a aspectos (AORE), que se enfoca a abordar la composición y análisis de asuntos de corte durante la ingeniería de requisitos para revelar las interrelaciones entre aspectos antes de que se establezca la arquitectura. La AORE se basa en referencias sintácticas, lo que resulta en cuatro problemas básicos:

- La composición de aspectos se basa en la estructura de requisitos en vez de las semánticas de los mismos.
- El uso de la composición sintáctica fuerza a que se hagan definiciones manuales de cada punto de unión antes de que se utilice.
- El análisis de compensaciones (*trade-off analysis*[1]) se basa en las composiciones sintácticas a partir de las cuales se hace referencia a las composiciones semánticas e interferencias.
- La composición sintáctica resulta en modelos frágiles.

Los autores diseñaron un lenguaje de descripción de requisitos (*Requirements Description Language*, RDL) basado en XML para solventar los problemas que se mencionan incrementando la especificación de requisitos en lenguaje natural con información semántica que deriva del propio lenguaje natural, lo que incrementa la intencionalidad de las composiciones y simplifica el razonamiento sobre la influencia e interrelación de los aspectos. Además, en [Chitchyan y cols., 2009] presentan los resultados de su estudio, en el cual compararon lenguajes de especificación de requisitos con RDL, ya que la fragilidad de los lenguajes que se basan en sintaxis se extiende a los artefactos que se generan de acuerdo al lenguaje.

Una solución a los problemas de expresividad de los aspectos consiste en los lenguajes de aspectos de dominio específico (DSAL), que son lenguajes se enfocan a un tipo particular de asunto de corte, a diferencia de los lenguajes orientados a aspectos de propósito general (GPAL). Los DSAL generalmente se utilizan en conjunto con lenguajes de programación de propósito general (GPL). Los autores presentan el concepto de los DSAL que se aplican

[1]Trade-off se define como un balance entre características incomparables entre sí donde una se pierde a costa de ganar otra.

a los lenguajes de programación de dominio específico (DSL), de modo que se provea de la capacidad de encapsular asuntos de corte que afectan las gramáticas [Rebernak, Mernik, Wu, y Gray, 2009].

2.1.4. Programación intencional

En [Simonyi, Christerson, y Clifford, 2006] se describe la problemática de que los lenguajes de programación se expresan en términos cercanos a la computadora, lo que dificulta que los expertos del dominio intervengan directamente en el desarrollo de software. El autor propone el uso de diversos lenguajes de dominio específico que trabajen en conjunto para abarcar los dominios de un sistema, además del uso de editores Wysiwyg. Se reporta la herramienta Domain Workbench, que permite el desarrollo de software por medio de un conjunto de DSL, que a su vez consisten en editores Wysiwyg [Simonyi y cols., 2006].

Se mencionan las desventajas de los lenguajes de programación, que aunque son precisos y libres de ambigüedades, se enfocan a registrar el trabajo de los programadores y todo cambio que se requiera en un sistema, se debe hacer sobre el código fuente. Dicho código posee la desventaja de que no registra las intenciones del programador de forma clara.

La programación más tradicional permite trabajar con dos elementos: el conocimiento del dominio del problema, que proviene de los expertos del dominio; y la implementación, que provee los ingenieros de software. La forma de trabajar consiste en que el experto le describe el problema al programador, este se documenta en forma de especificaciones, casos de uso, por mencionar algunos; entonces el programador entrelaza dichas intenciones por medio de ingeniería de software para generar código fuente en algún lenguaje de programación. Como desventaja, el mantenimiento y la exactitud del sistema se definen tanto en la oración del problema como en la implementación, pero de forma separada, pero el mantenimiento se debe realizar garantizando la exactitud por medio de evaluar que el código fuente concuerde con la oración del problema.

Por último, la programación intencional utiliza una herramienta que se conoce como *Domain Workbench*. La forma de trabajar consiste en que los expertos del dominio definen un esquema con ayuda de los programadores que sirve de interfaz entre el *Domain Workbench* y el generador, el esquema contiene al menos una identidad distinta para cada concepto del dominio. El código del dominio lo mantienen los expertos del mismo, de este modo se elimina la necesidad de código fijo, ya que el *Domain Workbench* tiene la

capacidad para proyectar el contenido de diversos dominios que se entrelazan por medio de varias vistas. El generador se programa de modo que se indique la forma en el que el código del dominio se debe procesar para obtener la instancia que se desea.

El esquema del dominio difiere de las clases y objetos en el hecho de que carece de semánticas. Esto se debe a que el experto del dominio no distingue objetos de cualidades, acciones, metas u otra cosa. Sólo necesita establecer una identidad que se realizará de acuerdo a la definición del esquema. Además, el esquema difiere del lenguaje de modelado unificado (*Unified Modeling Language*, UML) en el hecho de que carece de restricciones sobre lo que se representa o no en los dominios que describen los esquemas. Por medio del *bootstrapping* (arranque), se permiten argumentos circulares en la ingeniería de software, de este modo se considera que el esquema del dominio es extensible con nuevas propiedades, se proyecta y edita con notaciones diversas ya que cada lenguaje en un *Domain Workbench* se extiende y edita. El Domain Workbench se emplea para definir, crear, editar, transformar e integrar múltiples dominios. Se describe al *Domain Workbench* como una instancia del *Language Workbench*[2].

2.2. Elementos lingüísticos

En esta sección se presentan los elementos lingüísticos que se involucran en el desarrollo de esta tesis.

2.2.1. Deixis

El *Oxford English Dictionary* define la deixis como:

> *The function or use of deictic words or expressions (= ones whose meaning depends on where, when or by whom they are used).*

Donde el término *deictic* indica que una palabra o frase depende del contexto en el que se utiliza. Por ejemplo, la frase *"the list is complete"* implica una lista, pero el tipo de lista depende del contexto en el que se emplea la oración, ya sea una lista de invitados, una estructura de datos o una lista de mercado.

[2]En [Fowler, 2005], un *Language Workbench* es una herramienta que permite desarrollar aplicaciones por medio de lenguajes de dominio específico.

La deixis se encuentra presente en prácticamente todos los lenguajes naturales del mundo ya que permite expresar ideas reduciendo la cantidad de palabras al mínimo necesario, de modo que las oraciones presentarán ambigüedad que los interlocutores resuelven por medio del proceso cognitivo. Lo anterior sucede de forma inconsciente ya que los interlocutores emplean su razonamiento para complementar ideas parciales [Liblit, Begel, y Sweetser, 2006]. La deixis se relaciona con la *elipsis*[3] para eliminar palabras redundantes que evitan la ambigüedad, de modo que una oración como *"I will go to the classroom and you will go to the classroom too"*, se reduce sin perder su significado semántico a *"I will go to the classroom and you too"*.

Cabe destacar el caso particular de Jaaa, lenguaje de programación que nació como dialecto de Java y que permite el uso de anáforas de forma muy concreta y particular; su uso se limita a parámetros que recibe un método y sólo a uno por método [Lohmeier, 2011]. El autor menciona las anáforas directas como la referencia a conceptos que se definieron con anterioridad, como ejemplos se mencionan los pronombres *"it"* o *"those"*, además de conceptos finitos como *"the hard disk"*. Por otro lado, menciona la anáfora indirecta como una relación que no es explícita en el texto, pero que se encuentra en el modelo mental del usuario *"the hard disk"* implica una parte de la computadora.

La deixis permite reemplazar un significante[4] por una palabra genérica cuyo significado varía de acuerdo al contexto, lo anterior permite realizar oraciones gramaticalmente más cortas, pero ambiguas y cuyo contexto depende de la asociación entre la referencia indirecta y su referente.

Otra característica de las referencias indirectas, es que permiten describir situaciones en función no sólo del espacio, sino del tiempo. El manejo de esta capacidad temporal se abordó como una extensión al paradigma de la programación lógica, donde en diversos lenguajes agregan capacidades temporales a su funcionalidad para permitir la verificación y especificación de programas en función del tiempo. Se reporta el uso de operadores temporales unarios tales como *"always"*, que indica una condición que se debe mantener siempre; *"next"*, que indica una condición que se cumplirá en el siguiente estado; *"eventually"*, que

[3]La elipsis consiste en omitir una o más palabras sin perder el contexto de una oración [Lobeck, 2007].

[4]En semiótica, un significante se define como un componente del signo lingüístico, que se conforma por un conjunto de tres elementos: referente, que es una "cosa" real o abstracta; significante, que es la representación simbólica del referente; y significado, que es la representación de una idea mental [Ogden, Richards, Ranulf, y Cassirer, 1923]. Por otro lado, Ferdinand de Saussure menciona que existen signos sin referente, de modo que sólo considera al significante y al significado[De Saussure, 1916].

indica una condición que se cumplirá en algún estado futuro; *"all"*, que indica una condición que siempre se debe cumplir. Además de operadores binarios tales como *"Until"*, que indica que una condición *"q"* ocurre en el estado actual o futuro y otra condición *"p"* se cumple hasta esa posición, a partir de la cual ya no es necesario que se siga cumpliendo; *"release"* que indica que una condición *"q"* se cumple hasta que *"p"* se cumple, entonces ya no se debe cumplir *"q"* [Gaintzarain y Lucio, 2009].

Además de que se reportan extensiones para tratar la parte modal que incluye la posibilidad y la necesidad, y que se emplea en las matemáticas y la inteligencia artificial[5].

Si las referencias indirectas se encuentran dentro del mismo texto, se denominan endóforas; o si por el contrario, se definen en otro texto, se denominan exóforas.

2.2.1.1. Endófora

Una *endófora* es una referencia indirecta que se definió en el mismo texto, se clasifica en anáfora y catáfora del momento en el que se define el referente.

- Anafóra. Referencia indirecta cuyo referente se encuentra antes en el mismo texto: *"Take the key, use **it** in the lock"*.
- Catáfora. Referencia indirecta cuyo referente se encuentra después en el mismo texto: *"After take **it**, use the key in the lock"*.

2.2.1.2. Exófora

Una exófora es una referencia indirecta a algo que se definió en otro texto, un tipo particular de exófora es la homófora, que consiste en una referencia que depende de un contexto particular. Por ejemplo el tipo de dato *Integer*, que conceptualmente consiste en un tipo de dato para manejar números enteros, pero sus características particulares dependen tanto del lenguaje de programación como de la máquina que ejecutará el programa.

2.2.2. Expresividad

Según el Oxford English Dictionary, la expresividad se define como

[5]En [Orgun y Ma, 1994] se presenta un análisis que los autores realizaron a lenguajes de programación lógica temporal, lógica modal y lógica multimodal.

Showing or able to show your thoughts and feelings.

Con base en la definición, la expresividad es una característica propia de los lenguajes naturales ya que permiten que se describan ideas de forma tan concreta como ellos deseen. En contraste, los lenguajes de programación de propósito general (*general-purpose language*, GPL) dado que son representaciones formales, carecen del nivel de expresividad de los lenguajes naturales debido a su orientación lógica. Los DSL permiten un nivel de expresividad más elevado que los GLP, pero su uso se restringe a dominios particulares.

Los DSL no se limitan a la programación, sino que permiten a los no programadores realizar diversas tareas de un dominio particular sin lidiar con un GPL y utilizando elementos propios del dominio particular con el que se trabaja. Se reporta una gran cantidad de DSL que se enfocan no sólo a programación, sino también al proceso de ingeniería de software, multimedia o telecomunicaciones, por mencionar algunos[6].

La barrera del idioma es la limitante más notoria que se encuentra en la comunicación humana, de modo que desde la década de 1930 se reportan especificaciones controladas del idioma inglés para áreas tan diversas como lo son manuales de artículos de construcción o aviación comercial. La característica de dichos lenguajes consiste en que al ser versiones controladas, liberan a los usuarios de la gran cantidad de elementos ambiguos y regionalismos que diversos hablantes empleen. Limitando su uso a un área particular, se emplea el argot del dominio sin peligro de que haya alguna interpretación errónea[7].

En [O'Brien, 2003] se realizó una revisión de ocho versiones controladas del idioma inglés para obtener un conjunto de reglas generales. Los autores mencionan que de los ocho lenguajes que analizaron, sólo uno se enfoca a los programadores (*human-oriented controlled language*, HOCL), mientras que el resto se diseñaron de forma que las computadoras los procesen sin problemas (*machine-oriented controlled language*, MOCL). Además, varios de los conjuntos de reglas las analizaron bajo cláusulas de confidencialidad por lo que sólo se reportaron los resultados. Se describen tres tipos de reglas:

- Léxicas. Se enfocan a elegir palabras y afectar sus significados.
- Gramáticas. Se enfocan a afectar la sintaxis.

[6]Se recomienda [van Deursen, Klint, y Visser, 2000] para una revisión de 75 publicaciones que se relacionan con los DSL.

[7]Se recomienda [Kuhn, 2014] para una revisión de 100 versiones controladas del idioma inglés para diversos dominios.

- Textuales. Se enfocan a afectar el diseño o la información de las palabras (estructurales); o afectan el propósito del texto y la respuesta del usuario al mismo (pragmáticas).

Como resultados, se menciona que sólo una regla de las 61 que se obtuvieron se comparte por los ocho lenguajes, y siete reglas son comunes entre por lo menos cuatro de los lenguajes.

Cabe destacar que el contexto de expresividad varía entre los lenguajes naturales y los lenguajes formales, esto se debe a que en los primeros se espera que sean ambiguos y que la expresividad permita transmitir la mayor cantidad de información en la menor cantidad de palabras. Por otro lado, los lenguajes formales tienen como meta transmitir información precisa y sin ambigüedad, dejando la longitud de las construcciones como algo secundario, tal es el caso de un compuesto químico como la sal, cuyo nombre científico es *cloruro de sodio*, y su fórmula es $NaCl$, nótese que mientras la palabra sal se emplea para describir las propiedades del compuesto desde el contexto del sabor, el nombre científico y la fórmula proveen información sobre el compuesto desde un contexto más especializado; pero mientras que el nombre científico es fácil de leer y entender para el público en general, es probable que la fórmula se necesite de una explicación previa para que se entienda como por ejemplo, el motivo por el cual el sodio es Na y no So, o porqué va primero el sodio y luego el cloro en la fórmula.

En el caso particular de los lenguajes de programación la expresividad se enfoca no sólo a transmitir información, sino a que esta no sea ambigua al tiempo que es concisa, esto se debe a que mientras más información se transmite (en forma de líneas de código), hay un mayor riesgo de obtener un comportamiento inesperado en el código resultante (*bugs*). Además, el número de ideas no es importante, ya que este depende no sólo del lenguaje, sino del programador y el paradigma, que influyen en el número de líneas que se emplean para expresar una idea. Como consecuencia de esto, se deja de lado la legibilidad del código y el tiempo necesario para escribir un programa. Este enfoque permite el uso de métricas tales como las *métricas de complejidad de Halstead* (*Halstead complexity measures*) [Halstead, 1977], que dependen directamente del número de instrucciones sin considerar la legibilidad del código. En [Berkholz, 2013] se presenta un análisis de expresividad de lenguajes de programación con base en la cantidad de código que se emplea para resolver un problema, donde el autor considera que los lenguajes más expresivos son los que emplean la menor

cantidad de código para realizar una tarea, sin importar lo legibles que sean desde la perspectiva del lenguaje natural o lo complicado que sea aprender a programar con ellos.

2.2.3. Sintagmas

Otro elemento a destacar es la complejidad de las oraciones, mientras que en la POO y la POA se descomponen para dejar verbos y sustantivos simples, en un lenguaje naturalístico se integran para realizar instrucciones más complejas que no requieran de definiciones frágiles o poco expresivas desde la perspectiva del programador. En este caso se consideran dos tipos de frases: sintagma nominal y sintagma verbal [Bolshakov y Gelbukh, 2004].

Los sintagmas nominales son un conjunto de palabras que componen al sujeto de una oración, dichos sintagmas permiten una descripción basada en las propiedades de una abstracción, de este modo se definen instrucciones en función de dichas propiedades, pero además dichas instrucciones permiten indagar sobre la jerarquía, de modo que *casa con jardín* es una abstracción de tipo *Casa* que posee una propiedad de tipo *Jardín*, o una abstracción de tipo *CasaConJadrín* es un tipo de *Casa*; o bien una mezcla de ambos, de modo que *casa con jardín y alberca* es una abstracción de tipo *CasaConJardín*, que es un tipo de *Casa* que además posee una propiedad de tipo *Alberca*.

A su vez, el sintagma nominal complementa al sintagma determinante, que se compone además de un determinante[8].

Nótese que en esta tesis se utiliza la palabra *frase* de forma distinta a la palabra *oración*, esto se debe a que por frase se entiende un sintagma, ya sea un sintagma nominal, un sintagma verbal, un sintagma preposicional o un sintagma adjetival, por mencionar algunos. Por otro lado, una oración es un conjunto de palabras o sintagmas que se conforma de un sujeto gramatical y predicado.

2.2.4. Relaciones anafóricas

En los lenguajes naturales, las abstracciones se clasifican no sólo de acuerdo a su tipo, sino que también se clasifican de acuerdo a sus propiedades, de modo que *la casa* y *la casa con jardín* indican dos abstracciones diferentes, pero también a la misma abstracción con

[8]En la lengua española, un determinante es un complemento del sustantivo y que incluye al artículo, a los posesivos, los ordinales, multiplicativos, indefinidos, demostrativos, predeterminantes y numerales

una descripción más detallada en la segunda frase. En los lenguajes naturalísticos por otro lado, se requiere de una tipificación fuerte, incluso más que en otros paradigmas ya que las anáforas estructurales se basan en los tipos para una expresividad eficiente, de modo que las referencias indirectas tales como *la casa que se describió en este párrafo* se consideran correctas. Un nivel de expresividad más elevado se logra por medio de un soporte completo para trabajar con el sintagma nominal, lo que implicaría el uso de modificadores, determinantes y extensiones que permitirían una clasificación que no sólo se base en jerarquía, sino también en las propiedades de una instancia [Knöll, Gasiunas, y Mezini, 2011].

El uso de pronombres y tipos permite trabajar de forma eficiente con anáforas, aunque lo ideal sería considerar la deixis completa para establecer relaciones entre abstracciones de una forma más legible para los usuarios del lenguaje, de modo que oraciones tales como *el siguiente párrafo trata sobre las propiedades de un tipo*, cuyo sintagma nominal es *siguiente párrafo* indica una referencia indirecta cuyo significado cambiará siempre que se emplee dicha frase. De modo que si se utiliza en el siguiente capítulo, la frase *siguiente párrafo* hará referencia a otro.

Los pronombres se dividen en personales (*I*, *you*, *me*, *us*, *it* o *this*, por mencionar algunos), reflexivos (terminan en *-self*), recíprocos (*each other*), posesivos (*my*, *your*, *its* o *their*), demostrativos (*this* y *that*), indefinidos (con los prefijos *some-*, *any-*, *every-* y *no-*, con los sufijos *-thing*, *-one* y *-body*; o las palabras *many*, *both*, *more* y *most*), relativos (*who*, *whose*, *which* y *that*) y por último, interrogativos (similares a los relativos).

Del mismo modo, un sintagma determinante funciona como referencia indirecta a grupos de elementos, a un elemento particular de dicho grupo, o permite discernir si un elemento pertenece o no a dicho grupo, lo anterior se logra por medio de determinantes tales como los ordinales y los indefinidos. Por tanto, frases tales como *la primera casa de la lista* o *todas las casas de la lista* son correctas y compatibles con el predicado *se vendieron sin problemas*.

2.2.5. Contexto

El contexto es el conjunto de circunstancias comprobadas que ocurren alrededor de un evento, de modo que en el desarrollo de software se tiene una relación muy estrecha con el dominio del problema que se intenta solucionar. Muchas herramientas se enfocan a un contexto particular, pero dejan de lado otros. Los lenguajes de programación de propósito

general deben cubrir todos los contextos, de modo que poseen sintaxis y abstracciones que los cubran todos, provocando que muchas veces se incremente el código cuando haya situaciones que requieren de más de un contexto para trabajar, por ejemplo la expresión *add 5 to N*, donde se desconoce si *N* es una variable numérica o una lista de números. Es posible tratar dicha problemática definiendo cuidadosamente las abstracciones que se requieren para evitar la pérdida del mismo, o utilizando únicamente los elementos que se requieren en un contexto. Una solución consiste en definir capas contextuales que se activen en momentos determinados para cada contexto [Hirschfeld, Costanza, y Nierstrasz, 2008], pero a la vez se presenta el inconveniente de que dichas capas se activan de forma explícita, para lo cual es conveniente un mecanismo que se base en eventos [Kamina, Aotani, y Masuhara, 2010].

Por otro lado, los DSL son incapaces de responder a los cambios de contexto de un sistema ya que expresan semánticas complejas por medio de soluciones simples. Para lo cual se proponen soluciones como el desarrollo de un modelo para la interpretación dinámica de lenguajes de dominio específico, ya que estos ofrecen un alto nivel de expresividad para el desarrollo de aplicaciones especializadas, pero carecen de la flexibilidad para responder a los cambios que se hagan cuando cambia el contexto del dominio [Laird y Barrett, 2009]. Como solución, los autores describen un método adaptable para evitar que los lenguajes de dominio específico posean los problemas de los lenguajes estáticos, donde la parte adaptable consiste en diseñar un lenguaje donde se desacople la implementación de la intención para obtener adaptabilidad dinámica posterior al despliegue de la aplicación[9]. Se menciona una solución basada en componentes que involucra la reconfiguración dinámica de las interacciones de los componentes que conforman al intérprete.

2.2.6. Ambigüedad

La ambigüedad es una característica de los lenguajes naturales que provoca que una palabra u oración se interprete de varias formas. Para resolver la ambigüedad, quienes hablan emplean su razonamiento, pero las computadoras no razonan, de modo que se requieren de mecanismos para tratarla sin perder expresividad ya que una computadora ejecuta instrucciones precisas, por tanto una descripción ambigua resulta en una conducta

[9]Esto se reportó por primera vez en [Lieberherr, 1995].

inesperada, se plantean diversos mecanismos que se requieren para formalizarla de modo que la computadora la interprete correctamente.

Una técnica que se investigó desde hace varios años es la *resolución de anáforas*, que es el proceso de identificar el referente de una anáfora por medio de un enlace conceptual [Qiu, yen Kan, y seng Chua, 2004].

En [Arnold y Lieberman, 2010a], se propone que la computadora trate las ideas ambiguas por medio de un diálogo con el usuario para refinar la salida, aunque no se propone un modelo, sólo un mecanismo para generar código en un lenguaje de programación existente a partir de lenguaje natural. En [Arnold y Lieberman, 2010b], se describe la herramienta *Zones* que emplea la documentación para encontrar código reutilizable por medio de oraciones imprecisas, además de que se permite que el usuario agregue anotaciones al código, si así lo desea. Por medio de su *back-end*, *ProcedureSpace*, *Zones* permite encontrar pares de código-oración que asocian a los fragmentos dependiendo de su contexto. Otras herramientas limitan la ambigüedad gracias a diccionarios de datos, como en el caso de *Pegasus* y *Macho*; o simplemente se limitan a establecer palabras reservadas para mantener un control más estricto y formalizado.

Por otro lado, en [Begel y Graham, 2006b] se propone un algoritmo basado en el análisis GLR (Generalized Left to Right) para tratar la ambigüedad de los lenguajes embebidos y que emplea programación por voz. Consiste en dos gramáticas, una que utiliza tokens únicos y otra que utiliza tokens múltiples. Dicho algoritmo se implementó en la herramienta Harmonia. Se menciona el uso del análisis GLR en vez de usar el análisis LR (Left to Right), esto se debe a que el primero permite el análisis de varias instancias del segundo de forma simultánea, de este modo GLR permite tratar la ambigüedad procesando diversas instancias de un análisis. Se incluye además el tratamiento de palabras homófonas, de ahí el nombre de XGLR (eXtended GLR).

Capítulo 3

Trabajos relacionados

En esta sección se presenta un análisis de las diversas tecnologías que se enfocan al desarrollo de software por medio de un subconjunto formalizado del idioma inglés. El término *tecnologías* se utiliza porque no sólo se incluyen lenguajes, sino también *frameworks* y entornos de desarrollo integrado (*Integrated Development Enviroment*, *IDE*). Entre estas tecnologías se reportan herramientas de generación de software tales como Pegasus, Natural Language Computer, Macho, Metafor, por mencionar algunas, lenguajes de programación como Natural Java, o lenguajes orientados a la documentación como Attempto Controlled English. El objetivo de cada tecnología se centra en resolver problemas particulares y por tanto, estas tecnologías tienen un enfoque similar a los DSLs o, para ser más precisos, a lenguajes de cuarta generación (4GL), que son lenguajes orientados al sector empresarial y que se basan en una sintaxis expresiva y similar al idioma inglés [Heering y Mernik, 2002] [Mernik y Žumer, 2001].

3.1. FLOW-MATIC

FLOW-MATIC (también conocido como B-0) es un lenguaje de programación que se desarrolló para la UNIVAC I en 1955, su sintaxis semejante al idioma inglés inspiró a otros lenguajes como COBOL [Wexelblat, 1981]. FLOW-MATIC divide las instrucciones en dos categorías: instrucciones semejantes al inglés; por último, una sección particular para definición de datos con una sintaxis particular.

FLOW-MATIC se pensó como una forma de cerrar la brecha entre la programación y el lenguaje natural ya que, hasta ese momento, los desarrolladores utilizaban notación matemática para escribir código. Cuando se diseñó FLOW-MATIC, Grace Hopper propuso el uso del inglés como mecanismo para describir instrucciones ejecutables y formularios impresos para la definición de datos, de forma que se trabajaran de forma separada.

3.2. COBOL

COBOL (Common Business Oriented Language) es un lenguaje de programación estructurado y orientado a los negocios, su principal característica es que sus instrucciones poseen un enfoque imperativo y procedural semejante al idioma inglés, con una extensión que se agregó durante la década de 1990 y que se estandarizó en 2002, que permite trabajar con objetos [Lämmel, 1998]. Además de estas características, muchas compañías realizaron diversas modificaciones a COBOL para adaptarlo a una especificación particular o cuando se requiere expresividad semejante al inglés para alguna situación en específico.

Muchas de las palabras reservadas en COBOL son sinónimos o permiten definir expresiones con diferentes cantidades. Otra característica de COBOL radica en que el programador opcionalmente omita algunos elementos redundantes, por ejemplo *a IS GREATER THAN b* se simplifica como *a GREATER THAN b* o inclusive como $a < b$. Otro ejemplo es la palabra *DISPLAY*, que antes de un mensaje indica que este se presente en pantalla. Una variable que se definiera con anterioridad se asigna por medio de la sintaxis *MOVE ORIGIN-VAR TO DESTINYVAR*, en este caso la acción la indica el verbo *MOVE* y, la relación entre las variables se establece por medio de la preposición *TO*. Mientras que las variables se describen con sustantivos, las acciones se describen por medio de verbos. COBOL utiliza clases para crear un mecanismo de instanciación para incorporar el paradigma orientado a objetos para de esta forma crear interfaces con lenguajes orientados a objetos que utilicen COBOL como *back-end*.

Aún con los beneficios y su extendiso uso en la industria, la sintaxis de COBOL es imperativa, procedural y orientada a objetos cuando se requiere. Mientras que la incorporación del paradigma orientado a aspectos se consideraba un gran avance para convertir a COBOL en un lenguaje naturalístico propiamente dicho, sólo se reporta una propuesta conceptual [Lämmel y De Schutter, 2005]. Por tanto, mientras COBOL carece de las carac-

terísticas que se requieren para un lenguaje naturalístico, tales como referencias indirectas y manejo de temporalidad, posee un nivel de expresividad significativo y la formalidad suficiente para el desarrollo de software sin importar el dominio del problema.

3.3. Lenguaje de diálogo de Heidorn

En [Heidorn, 1973], el autor analiza un sistema experimental que "conversa" en lenguaje natural con el usuario acerca de un problema para resolverlo. En este sistema, el usuario define problemas por medio de oraciones en lenguaje natural; entonces, si el problema no se resuelve, el sistema solicita al usuario más oraciones. Cuando hay oraciones suficientes, el sistema analiza la información para resolver el modelo a ejecutar, después de esto, el usuario solicita por medio de texto en inglés, un análisis en cuanto a si el sistema entendió las oraciones y si se procesaron de forma correcta; también se solicita una representación en GPSS[1]. Este sistema se basa en simular un diálogo en inglés con el usuario para producir la implementación de un DSL.

3.4. Structured Query Language (SQL)

Structured Query Language (SQL) es un lenguaje de dominio específico que se enfoca en la creación y manipulación de bases de datos [Kim, 1982]. Originalmente, SQL se creó bajo el nombre de *SEQUEL*, pero su nombre se cambió debido a problemas de derechos de autor [Chamberlin y Boyce, 1974]. SQL se pensó como un lenguaje declarativo con un nivel de expresividad que permite definir elementos de bases de datos de forma clara y simple. Estos elementos son: entidad, que se refleja en forma de una tabla; atributo, que se representa como el campo de una tabla; relación, que es una asociación entre dos o más tablas; y por último, el registro (o tupla), que es la información que se almacena en la base de datos. Una instrucción define una acción particular, ya sea la creación de una tabla (con llaves y relaciones con otras tablas), la inserción, eliminación, modificación o la consulta de información. El nivel de expresividad de SQL es mayor que el de los lenguajes de programación tradicionales, posee una sintaxis declarativa fácil y claramente definida, pero

[1] *Gordon's Programable Simulation System* (GPSS), lenguaje de programación de propósito general que se utiliza en simulaciones de tiempo discretas [Schriber, 1990].

una implementación pura de SQL carece de estructuras de control que poseen los lenguajes de propósito general. Mientras que varios programadores desarrollaron extensiones de SQL que proveen soporte para procedimientos, estos generalmente se restringen a un entorno particular y no son intercambiables. Un ejemplo simple de SQL es el siguiente:

```
SELECT id, name, age, wage
FROM workers_db
WHERE position = 'manager'
```

Como se observa, este ejemplo es muy expresivo y fácil de entender: se obtendrá una consulta con el *id*, *nombre*, *edad* y *salario* de todos los gerentes.

Es de destacar que en [I. Androutsopoulos y Thanisch, 1995], los autores revisaron el modo en que las interfaces de lenguaje natural permiten la generación de consultas de SQL a partir de una entrada en lenguaje natural controlado que nombraron *Natural Language Interfaces to Database* (NLIDB), donde identificaron ventajas como el uso de un lenguaje natural en lugar que utiliza consultas en forma de preguntas y soporta una forma limitada de oraciones elípticas. Los autores describen las desventajas siguientes: un subconjunto muy limitado del lenguaje natural; los sistemas no entienden elementos gramaticales tales como conjunciones o falsos positivos ya que carecen de habilidades lingüísticas y razonamiento; y, que el usuario da por hecho que está tratando con un sistema inteligente. En [Nihalani, Silakari, y Motwani, 2011], los autores resentan una revisión histórica de diversos NLIDBs.

3.5. Natural Language Computer (NLC)

En [Biermann y Ballard, 1980], los autores presentan Natural Language Computer (NLC), un lenguaje de programación que se enfoca a realizar operaciones sobre arreglos por medio de un limitado subconjunto del idioma inglés. Mientras NLC resuelve problemas de expresividad por medio de lenguaje natural, para ese entonces no se reportaba alguna tecnología que proporcionara la habilidad de trabajar con construcciones semánticas, así que los usuarios de NLC no generan fácilmente expresiones aceptables por el sistema porque el lenguaje natural posee un rango ámplio de reglas y conjuntos de palabras.

Los autores proponen dos reglas para el uso de lenguaje natural como lenguaje de programación:

- Sólo se hace referencia a datos en pantalla y estructuras de datos por medio de oraciones simples.
- Verbos imperativos para instrucciones.

Los autores describen el inglés como un mecanismo para programar que es tan preciso como el programador desee. NLC descompone oraciones y compara tokens con un diccionario de datos y entidades que el usuario definió, entonces realiza correcciones ortográficas y revisa abreviaturas y ordinales. NLC procesa sustantivos, verifica frases y palabras que alteren el significado y genera un conjunto de elementos que hacen referencia a las estructuras que se despliegan en pantalla. NLC analiza verbos imperativos sin procesarlos y los actualiza en pantalla. Si un verbo no se procesa, se analiza con un módulo semántico donde se procesan algunos verbos predefinidos con verbos que el usuario definió.

Un conjunto de características par consiste en un diccionario que establece si una palabra es un verbo, un pronombre, un adjetivo o un sustantivo. NLC realiza correcciones ortográficas y sugiere alternativas.

El usuario describe una acción ejecutable por medio de un verbo imperativo, un sintagma nominal (un sustnativo con su determinante) y un *verbicle* (que los autores definen como una preposición asociada a un verbo imperativo), por ejemplo: *multiply X by Y*, donde *by* es el *verbicle*. El parser procesa la oración *add X to Y* en el siguiente orden:

```
imperative, noun phrase, verbicle, noun phrase, end (period).
```

Por conjunción, NLC utiliza una expresión MIX A, como se muestra a continuación:

```
A, separator (comma), A, separator (comma), (optional) "and", A.
```

23 estudiantes probaron NLC después de recibir entrenamiento. De 1581 instrucciones, 81 % se procesaron inmediatamente, mientas que del 19 % restante, NLC rechazó la mitad debido a limitaciones del sistema, mientras que la otra mitad fue por error humano. En conclusión, se necesitan cerca de 300 palabras para resolver un problema, con sólo ocho palabras que se detectaron como faltantes. La definición de palabras incompletas se debió a que se involucraron elementos implícitos en formas más naturales. Se procesaron instrucciones de forma correcta entre un 70 % y 90 % de las veces. Los mensajes de error fueron incompletos. Por último, los autores mencionaron que se requería un refinamiento para procesar cuantificación.

3.6. Layered Domain Class (LDC)

En [Ballard y Lusth, 1983], los autores presentan Layered Domain Class (LDC), que es un lenguaje de tamaño medio que se enfoca en consultas de datos empresariales y que aprende de nuevos dominios en inglés. Un dominio se divide en capas si tiene una matriz NLC que consiste en una relación estructural donde las entidades conforman un nivel inferior (por ejemplo, un calendario tiene meses, días y semanas como entidades de nivel inferior). LDC tiene dos componentes: Prep (que se ejecuta por medio de un componente de adquisición de conocimiento) y la fase de usuario (que se ejecuta por medio del procesador *User-Phase*). Prep establece entidades del dominio, relaciones y sustantivos asociados que el usuario debe proveer, y estructuras de datos tales como entidades involucradas. Prep establece además si las entidades son o no compuestas, si las entidades son plurales, o si poseen ordinales. Posteriormente, Prep solicita sustantivos para verificar entidades y verbos, para trabajar con ellas. Durante la fase de usuario, LDC recibe palabras y las compara con otras que se encuentran almacenadas en el diccionario que se creó durante la ejecución de Prep, entonces crea una estructuras sintáctica de nombre *bubble*, que es la representación formal de las consultas del usuario.

LDC permite varios modificadores: ordinales (*the second floor*); superlativos (*the largest office*); adjetivos (*the large room*); modificadores de sustantivos (*conference rooms*); subtipos (*offices*); sustantivos que toman argumentos (*classmates of Jim*); sintagmas preposicionales (*students in CPS215*); sintagmas comparativos (*students better than Jim*); sintagmas verbales triviales (*instructors who teach AI*); sintagmas verbales con parámetros implícitos (*students who failed AI*); sintagmas verbales operacionales (*students who outscored Jim*); adjetivos que toman argumentos (*offices adjacent to X-238*), y negaciones (*the non-graduated students*). LDC posee además soporte para modificadores de sintaxis únicamente anafóricos tales como: comparativos anafóricos (*better students*); adjetivos anafóricos que toman argumentos (*adjacent offices*); verbos anafóricos con parámetros implícitos (*failing students*); y, sustantivos anafóricos que toman argumentos (*the best classmate*) [Ballard, 1984].

3.7. Transportable English-Language Interface (TELI)

En [Ballard y Stumberger, 1986], los autores presentan Transportable English-Language Interface (TELI), que es un analizador de lenguaje natural que permite la modificación de las palabras de un dominio específico conocido dentro de un sistema. TELI utiliza bases de datos en primera forma normal para analizar y aprender sobre los conceptos del dominio del problema, para de este modo realizar consultas en lenguaje natural. Los autores mencionan que la implementación falla porque el usuario no provee reglas gramaticales claras y palabras reservadas, o no se reciben las construcciones gramaticales que se requieren para tratar el dominio del problema. TELI depende del lenguaje natural porque su sintaxis se basa en LISP. TELI provee varias características tales como verbos pasivos, relativos preposicionales y adjetivos reducidos, cuantificadores, y números. Además, TELI proporciona procesamiento semántico donde el usuario provee las definiciones, el significado de los modificadores se infiere como extensiones de manejo de predicados y su análisis semántico es composicional. Las semánticas son definiciones semejantes a las de LISP que se almacenan y asocian con un índice que es una palabra que se definió con anterioridad. La desventaja de TELI es que no posee soporte para operaciones aritméticas.

3.8. HyperTalk

HyperTalk es un lenguaje de programación embebido dentro de Apple HyperCards que se utiliza para la manipulación de datos e hipermedia con una gramática similar al idioma inglés [Wheeler, 2004] que se basa en Macintosh Toolbox. HyperTalk posee una estructura lógica similar a la de Pascal que se organiza por medio de eventos, y se diseñó pensando en usuarios no programadores para cerrar la brecha entre ellos y la programación. HyperTalk describe transiciones de elementos de HyperCard que se conocen como *cards*, que son contenedores para objetos tales como botones o campos de texto. HyperTalk permite el uso de referencias indirectas tales como *it*, *the third*, *last* and *before*, por mencionar algunas.

En [Eisenstadt, 1993], el autor describe ocho problemas que presenta HyperTalk, un lenguaje fácil de aprender y usar, pero difícil de depurar porque HyperCards posee una estructura compleja y carece de herramientas para depurar código. El proceso de depuración requiere muchos objetivos secundarios y el intérprete es inaccesible durante la ejecución, de modo que el usuario no utiliza herramientas de apoyo, ni observar una vista de eje-

cución coherente. El usuario debe poseer un entendimiento profundo para asegurar una correcta interpretación del flujo de datos interrelacionado, y requiere de un entendimiento aún más profundo para cerciorarse del estado interno de los objetos, que no siempre es igual al estado aparente.

3.9. AppleScript

Como AppleScript deriva de HyperTalk, también es un lenguaje de scripting desarrollado por Apple que se utiliza para la comunicación entre aplicaciones por medio de *AppleEvents* [Apple Inc., 2016]. AppleScript trabaja de forma similar a una consola de UNIX. AppleScript se diseñó para su utilización con una biblioteca predefinida de objetos que Apple diseñó. AppleScript continúa la idea de HyperTalk de crear un lenguaje de programación similar al lenguaje natural y que sea fácil de entender. A continuación se presenta el ejemplo de una lista de elementos de la cual se selecciona uno:

```
set chosenListItem to choose from list {"A", "B", "3"}
  with title "List Title"
  with prompt "Prompt Text"
  default items "B"
  OK button name "Looks Good!"
  cancel button name "Nope, try again"
  multiple selections allowed false
  with empty selection allowed
```

La palabra reservada *if* define condicionales con un nivel de expresividad similar al de una oración en inglés:

```
if x is greater than 3 then
  -- commands
else
  -- other commands
end if
```

La palabra reservada *repeat* se utiliza para definir ciclos y se utiliza en el siguiente ejemplo para describir un ciclo que se repite diez veces:

```
repeat 10 times
     -- commands to be repeated
end repeat
```

Como se observa en los tres ejemplos, AppleScript se diseñó pensando en una sintaxis que se asemeje al inglés y donde la ambigüedad se resuelve por medio de limitar el dominio a los elementos que se definieron en la biblioteca.

3.10. Attempto Controlled English (ACE)

En [N. E. Fuchs y Schwitter, 1996], [N. Fuchs y cols., 1999] y [N. E. Fuchs, 2018], los autores presentan Attempto Controlled English (ACE), una versión formalizada del idioma inglés que es transformable en lógica de primer orden y se ve como lenguaje natural. ACE carece de ambigüedad y es entendible por personas que hablen inglés, pero además por computadoras. ACE consiste en varios elementos. Attempto Parsing English (APE) es un intérprete basado en Prolog que incorpora construcciones de ACE y reglas de interpretación, con texto que se traduce de forma no ambigua en estructuras de representación de discurso (Discourse Represetation Structures, DSR). DSRs son variantes de lógica de primer orden fáciles de traducir a cualquier lenguaje formal. Attempto Reasoner (RACE) es una herramienta basada en Prolog que permite razonamiento automático para utilizarlo como mecanismo de verificación de la influencia de un texto sobre otro. Además, los autores describen las extensiones Semantic Web ACE, así como la traducción entre estas y lenguajes tales como OWL, SWRL, RuleML, Protune y R2ML, mientras que ACE se utiliza como un lenguaje de consultas MIT Handbook. AceWiki es la combinación de ACE y la herramientas enfocada a wikis Semantic Web [N. E. Fuchs, Kaljurand, y Kuhn, 2008].

Las palabras reservadas (que se toman de funciones y algunas palabras semejantes a anáforas) y palabras de contenido constituyen el vocabulario de ACE, con definiciones gramaticales que restringen la forma y significado de las oraciones. El texto es una oración declarativa anafóricamente relacionada que es simple o compuesta, donde las oraciones compuestas utilizan expresiones *and*, *or* y *if-then* para conectar oraciones simples. Las oraciones de consulta utilizan un formato de pregunta con el caracter *?* y palabras tales como *what*, *which* o *how*, mientras que aquellas oraciones que finalizan con *!* indican comandos ejecutables. Para lidiar con la ambigüedad, las oraciones ambiguas se descomponen en dos o más frases para reemplazar palabras sin perder el significado. Para referencias anafóricas, ACE busca anáforas y las reemplaza con el último sustantivo que sea el más específico y consistente con el género y número. Las pruebas indicaron que los artefactos de software que se documentaron con ACE son claros y se definen con un inglés controlado [Kuhn y Bergel, 2014], con ACE dando mejores resultados que Prolog y SOUL en cuestionarios aplicados a 20 experimnentados ingenieros de software graduados y por graduarse, considerando el tiempo de ejecución y las preferencias del usuario.

3.11. MOOSE Crossing

En [A. S. Bruckman, 1997] y [A. Bruckman, 1998], los autores presentan MOOSE Crossing, un MUD[2] educacional basado en texto que se orienta a objetos y que posee un DSL de nombre MOOSE, que posee una sintaxis similar al inglés y se enfoca a niños con edades entre 9 y 19 años. Los niños utilizan MOOSE para describir acciones, criaturas o lugares por medio de procedimientos que se establecen con verbos, cadenas, números o referencias que se reciben utilizando otros verbos. MOOSE es un lenguaje fácil de aprender donde la simplicidad se prefiere por encima de la formalidad y en el cual se evitan caracteres no alfanuméricos tanto como sea posible.

3.12. NaturalJava

En [Price, Rilofff, Zachary, y Harvey, 2000], los autores presentan NaturalJava, que es una interfaz de usuario que se basa en tres sistemas: *Sundance*, un procesador de lenguaje natural; *Programming Instruction Synthesis Module* (PRISM), un intérprete de estructuras; and, *TreeFace*, un manejador de árboles de sintaxis abstracta de Java. Mientras que NaturalJava genera código Java a partir de expresiones de lenguaje natural, este no provee soporte a arreglos y clases internas.

NaturalJava limita la interferencia para reducir la ambigüedad, utiliza Sundance como un analizador sintáctico parcial de lenguaje natural que, se enfoca a la extracción de información y resuelve expresiones frágiles. A diferencia de los analizadores sintácticos completos que generan un árbol de sintaxis abstracta, NaturalJava genera fragmentos de sentencias en forma de representaciones sintácticas planas por medio del analizador sintáctico parcial.

La extracción de información es un proceso de lenguaje natural que extrae tipos predefinidos de un texto en lenguaje natural, se enfoca en detalles importantes y deja de lado información irrelevante. Los sistemas de extracción de información incluyen varios dominios que, en este caso, las definiciones de lenguaje natural proveen construcciones del dominio de Java que Sundance procesa para generar 27 casos. Un caso es una representación de la construcción del programa y que se usa para describir conceptos.

[2]Un Multi-User Dungeon (MUD) es usualmente un videojuego de rol basado en texto [Bartle, 2003].

PRISM provee una interfaz de usuario para NaturalJava donde se agregan sentencias, y que permite agregar, borrar, modificar y consultar la información de un AST por medio de comandos. Los casos se dividen en dos tareas: un usuario determina las acciones a ejecutar y PRISM obtiene la información necesaria para los casos de prueba, de este modo se completa la petición. TreeFace genera un AST, mientras que una instancia (TreeFace object) mantiene un registro de la edición del contexto con la información a realizarse y que se determina por medio del concepto que PRISM almacena. Además, TreeFace provee nuevas interfaces, clases, campos, métodos, variables locales, ciclos, condicionales y asignaciones, además de que permite la creación de valores de retorno.

3.13. HANDS

En [Pane, Myers, y Miller, 2002] se presenta la herramienta Human-centred Advances for the Novice Development of Software (HANDS), que se enfoca a la programación de propósito general y la usabilidad por parte de niños de quinto grado o superior. Los autores descartaron otras características tales como la eficiencia, la correctitud o la similitud con otros lenguajes para enfocarse en la usabilidad.

Los autores realizaron un estudio para analizar cómo los niños y adultos describen estructuras antes de aprender a programar [Pane, Ratanamahatana, y Myers, 2001], observaron que los no programadores tienden a utilizar descripciones basadas en eventos y reglas donde los sustantivos se combinan en conjuntos y se evitan las estructuras de datos. Los conceptos matemáticos se describen en términos de lenguaje natural, mientras que las "cosas" conocen su propio estado de forma implícita y sólo se hace explícito cuando este cambia.

La usabilidad se describe como algo a tener en cuenta cuando se diseñan nuevos lenguajes, esto se debe a que la usabilidad se incrementa o decrementa con base en el usuario objetivo del lenguaje.

HANDS trabaja con su entorno de programación, que consiste en un compilador, un depurador y otros elementos, además de que posee un lenguaje textual y visual que se complementan. Dado que HANDS es un lenguaje enfocado al usuario, evita conceptos matemáticos, donde el artículo *the* se utiliza en cualquier parte del código, además de que se permiten instrucciones tales como *end if*. HANDS utiliza una representación gráfica en

forma de cartas para las variables y que se manejan por medio de un agente de nombre *Handy*. A continuación se presenta un ejemplo:

```
set nectar of flower to 100
set flower's nectar to flower's nectar + 25
add 25 to flower's nectar
```

Donde *nectar* es una propiedad de la tarjeta *flower*, con las instrucciones posteriores representando el mismo evento, mientras que la segunda instrucción es más natural desde la perspectiva del usuario.

Además, HANDS utiliza mecanismos similares a las consultas para trabajar con iteradores:

```
all flowers evaluates to orchid, rose, tulip
all bees evaluates to bumble
all snakes evaluates to the empty list
all (flower and (nectar < 100)) evaluates to orchid
```

Donde la palabra *all* se emplea como una referencia indirecta para todas las cartas que se asocian con *flower*.

3.14. Aplicación basada en acción para interfaces de lenguaje natural

En [Chong y Pucella, 2004], los autores presentan un framework que permite la creación de interfaces en lenguaje natural para aplicaciones que se basan en acciones. Los autores describen una interfaz de aplicaciones como un conjunto de enlaces que facilitan la interacción entre el usuario y el sistema. La interfaz de usuario recibe la información que proporcione el usuario y representa las respuestas del sistema a esas entradas.

Para trabajar adecuadamente, se requiere una correcta separación la interfaz y la aplicación, de modo que muchos tipos de interfaz se utilicen de forma transparente o, por lo menos, que se emplee un adaptador para permitir que el framework trabaje de forma correcta.

El framework traduce lenguaje natural en lógica de orden superior por medio de gramáticas categoriales que devuelven una representación semántica para la entrada. Como resultado, el marco de trabajo genera expresiones de cálculo lambda simples que se ejecutan por medio de llamadas a procedimientos desde al interfaz de usuario.

3.15. Metafor

En [Liu y Lieberman, 2005a], los autores presentan Metafor, una herramientas que permite la abstracción de los elementos necesarios para ayudar a nuevos programadores a incrementar sus habilidades de abstracción. Además, programadores experimentados utilizan Metafor como un mecanismo de "lluvia de ideas" durante las etapas tempranas del desarrollo de software. El usuario escribe el texto en forma de una historia que Metafor analiza por medio de semánticas de programación que devuelve código Python, que no es ejecutable directamente, pero ayuda a consolidar las ideas del usuario desde una perspectiva programática. Esta técnica se presenta como una alternativa que permite que el usuario programe de forma interactiva, en lugar de utilizar libros de programación como medios de aprendizaje. Metafor no analiza toda la gramática del inglés y permite que el código fuente se genere de forma aproximada a su interpretación en lenguaje natural.

Metafor se utiliza para resolver problemas de programación orientada a objetos, que consisten en el hecho de que los objetos son ineficientes cuandos se trata de describir protocolos o abstracciones que involucren patrones, que se resuelven por medio de lenguajes de patrones. Además, la programación orientada a objetos carece de soporte para encapsular requisitos no funcionales, algo que se soluciona por medio de aspectos. Las diferencias entre la programación orientada a aspectos y los lenguajes de patrones radica en que estos últimos encapsulan abstracciones que cubren muchos objetos, mientras que los aspectos encapsulan abstracciones dispersas entre objetos. Aunque el concepto de abstracción consiste en tomar algunos elementos e ignorar otros, esto no es lo ideal cuando se trata de aplicaciones sensibles al contexto [Lieberman, 2006].

Metafor analiza pronombres como *it* y *he* vía preprocesamiento y, a diferencia de otras herramientas, utiliza la ambigüedad como ventaja porque en los lenguajes tradicionales, los programadores toman decisiones esenciales durante las primeras etapas de desarrollo y a partir de ahí estructuran todo el código, por tanto, se requiere reestructurar todo el código si más adelante se encuentra que las decisiones son contraproducentes. El diseño de objetos se simplifica por medio de representaciones figurativas, que son mecanismos eficientes para llevar a cabo correcciones. Mientras esta tarea generalmente se lleva a cabo por medio de lenguajes formales, los autores proponen resolver este desafío con una representación de objetos más neutral en forma de tuplas con formato *(nombre, argumentos, cuerpo)*. Un analizador de tipos examina de forma dinámica el cuerpo y argumentos. Metafor utiliza

normalización para convertir un adjetivo en un sustantivo para transformar expresiones como *the drink is sweet* en *the drink has sweetness*.

3.16. Spoken Java

En [Begel y Graham, 2005], los autores presentan Spoken Java, una herramienta para programación por voz en el lenguaje Java. Spoken Java permite codificar en tres formas: dictar palabra por palabra, por medio de lenguaje natural o parafraseando texto. Dictar es una tarea lenta y tediosa, mientras que parafrasear texto implica problemas de abstracción que impiden comprensión de texto por parte de programadores inexpertos. Los autores describen la pronunciación como una desventaja porque los lenguajes de programación se enfocan a una representación escrita y no se enfocan al dictado, lo que da como resultado estructuras impronunciables o difíciles de pronunciar. Más aún, los usuarios generalmente hablan acerca del código que desean escribir, en lugar de hablar el código en sí mismo. Spoken Java es un súperconjunto de Java con reglas adicionales y una ambigüedad semántica y sintáctica incrementadas.

En [Begel y Graham, 2006a], se presenta un estudio que se realizó a Spoken Java con diez programadorees que poseen las siguientes características: cinco saben Java y cinco no; cinco son angloparlantes nativos y cinco no; y, cinco son estudiantes de Estados Unidos y cinco no. El código que recitaron se tradujo a texto. Se observaron elementos con una compleja traducción a código, por ejemplo, la expresión *array[i]* se tradujo como *array sub i* por parte de algunos programadores, como *array of i* por parte de otros, y como *i from array* por otros más. Las palabras homófonas son otra desventaja porque sonidos como *for* se entienden como un ciclo (*loop*, en inglés), un número *4* (*four*, en inglés) o la variable *fore*. El uso de letras mayúsculas es además un problema porque la definición de clases se recitó de forma explícita como *that class with capital c* por parte de algunos participantes, de modo que el analizador necesitó deducir si la palabra significa *c* o *see*. Otro problema son los espacios en blanco, esto se debe a que el sistema no reconoce la diferencia entre *dropStack Process* y *drop stack process*. Los autores mencionan que la pronunciación de palabras parciales es un problema para personas que no hablan inglés de forma nativa porque pronunciaban la palabra o frase completa, u omitían secciones como en *printLn*. Quienes hablan inglés omiten puntuación cuando hablaron con los autores, quienes detectaron

una interpretación obstaculizada en sentencias tales como *ex.printStackTrace()*, esto se debe a que algunos participantes parafrasearon la instrucción como *ex printStackTrace*, mientras que otros la pronunciaron como *ex dot printStackTrace*. Los usuarios describieron construcciones de diversas formas y utilizaron expresiones como *no arguments* en forma de sinónimos para *()*. Por último, la prosodia, que es una variación en la voz que se emplea para expresar ideas ambiguas en su forma correcta, tiene un peso elevado porque los usuarios la emplean para recitar las expresiones, de modo que la expresión *array sub i plus plus* se interpreta como *array[i]++* o *array[i++]*. El ejemplo anterior se define como un *array sub i <pause> plus plus*, mientras que el segundo se define como *array sub <pause> i plus plus*. Algunos usuarios no angloparlantes nativos tuvieron dificultad con esto porque no están familiarizados con la prosodia del idioma inglés.

3.17. Computer-Processable Language (CPL and CPL-Lite)

En [Clark, Harrison, Jenkins, Thompson, y Wojcik, 2005] y [Clark y cols., 2010], los autores describen a Computer-Procesable Language (CPL), que es un lenguaje natural controlado que se desarrolló utilizando un enfoque naturalístico y el cual posee una variante formalística que denominaron CPL-Lite.

CPL emplea heurísticas para tomar decisiones cuando encuentra ambigüedad, de modo que produce la implementación que el usuario espera. CPL acepta tres tipos de sentencias – hechos, preguntas y reglas. Estos incluyen auxiliares y partículas gramaticales[3] con un verbo, y en el cual, un sustantivo se modifica con otros sustantivos, preposiciones y adjetivos.

Los autores además desarrollaron CPL-Lite, un mecanismo que define consultas de forma comprensible y controlable. A diferencia de CPL, CPL-Lite no utiliza heurísitcas, lo que compensa al utilizar un intérprete más restringido para lidiar con expresiones ambiguas, y así sólo emplea palabras que se traducen al concepto que se definió en la ontología.

[3]De acuerdo con el *Oxford Dictionary*, una partícula gramatical es cualquier clase de palabra, tales como *in, up, off, over*, que se utilizan con verbos para hacer verbos frasales. Una partícula es una palabra sin significado claro y que no se inflecciona.

Tanto CPL como CPL-Lite poseen el mismo nivel de expresividad, pero CPL requiere menos palabras porque contiene varios mecanismos que permiten el nivel de expresividad necesario para el significado correcto. En el siguiente ejemplo que se tomó de [Clark y cols., 2010], CPL se utilizó en el contexto de la herramienta AURA, un sistema basado en conocimiento para física, química y biología:

```
A man drives a car along a road for 1 hour.
The speed of the car is 30 km/h.
```

A continuación, el mismo ejemplo con CPL-Lite:

```
A person drives a vehicle.
The path of the driving is a road.
The duration of the driving is 1 hour.
The speed of the driving is 30 km/h.
```

CPL infiere el concepto y los elementos necesarios, mientras que en CPL-Lite, el usuario los describe de forma explícita. Aún con esto, ambos enfoques son aparentemente incompatibles, con CPL-Lite embebido en el núcleo determinístico de CPL, lo que significa que CPL incluye los 113 patrones de sentencia de CPL-Lite, y ambos lenguajes utilizan el mismo intérprete.

3.18. Prototipo traductor de lenguaje natural a Python

En [Vadas y Curran, 2005], los autores presentan un prototipo que recibe entradas en lenguaje natural y devuelve código Python. Los autores indican que las tareas de programación y análisis de código generalmente son complicadas si no se posee una explicación en lenguaje natural, y que los lenguajes de programación poseen un nivel incierto de detalle desde la perspectiva del lenguaje natural. En el siguiente ejemplo se describe la entrada deseable por parte del usuario:

```
read in a number
add 2 to the number
print out the number
```

Que se convierte en la siguiente representación en Python:

```
number = int(sys.stdin.readline().strip())
number += 2
print number
```

Como se observa, mientras la conversión a Python es transparente, la ambigüedad se mantiene cuando se lee un número, en este caso sumando 2 porque el prototipo no distingue entre una adición y una concatenación de cadenas. Otra variante es la siguiente.

```
read in 2 numbers
add them together
print out the result
```

Cuyo equivalente en Python es:

```
number1 = int(sys.stdin.readline().strip())
number2 = int(sys.stdin.readline().strip())
result = number1 + number2
print result
```

Como se muestra en el ejemplo anterior, el prototipo generó dos variables porque las expresiones *read in 2 numbers* y *add them together* implican una referencia indirecta a la creación e invocación de una variable, a diferencia del primer ejemplo donde *number* se describe de forma explícita en las tres instrucciones.

En el primer ejemplo, la ambigüedad se resuelve por medio se suposiciones: si la variable se llama *number*, es más probable que se utilice como un valor numérico que como un string. Por otro lado, el segundo ejemplo se resuelve cuando el prototipo analiza la expresión *2 numbers*, el alcance de *them* y la correcta asociación con *result*. El último ejemplo aún no tiene una solución adecuada.

Los autores realizaron un estudio de cómo los programadores describen tareas, con una muestra que consistió de 370 sentencias que proporcionaron 12 programadores. Para esta muestra, los autores encontraron que los programadores generalmente describen tareas en términos de programación, en lugar de utilizar una solución más simple y en lenguaje natural. Por ejemplo, los autores encontraron que algunos programadores describen construcciones procedurales, incluso agregando la expresión *end loop* que se corresponde parcialmente con la gramática categorial combinatoria (*Combinatory Categorial Grammar*), posteriormente, un parser analiza la muestra y los errores se corrigen de forma manual. Los autores describen además que se requirieron correcciones para casi todas las instrucciones.

Como trabajo a futuro, los autores describen que utilizarán un mejor prototipo que sea capaz de una correcta identificación para una muestra mayor, y entonces refinen el parser y el motor semántico, para utilizar un sistema de cuestionarios para obtener información adicional.

3.19. Pegasus

En [Knöll y Mezini, 2006], los autores presentan una herramienta que no sólo facilita el desarrollo de software por medio de instrucciones en lenguaje natural, sino que además provee soporte para múltiples idiomas y de este modo facilitar el trabajo en equipos con integrantes de diferentes países y que hablen distintos idiomas sin afectar el desarrollo del sistema. En el estado que se reporta en el artículo, Pegasus genera programas simples en inglés y alemán, mientras que en [Mefteh, Ben Hamadou, y Knöll, 2012] se describe una extensión para el idioma árabe que se denomina Ara_Pegasus.

Los autores describen cuatro problemas que identificaron como motivos por los cuales no se reduce la brecha entre las expectativas del programador y las técnicas de programación:

1 Problema mental. Una idea se debe adaptar al lenguaje de programación, descomponiéndola o reagrupandola con otras.

2 Problema del lenguaje de programación. Un algoritmo se debe traducir a diversos lenguajes que usan nuevas tecnologías y conceptos, lo que significa que los viejos lenguajes aún tienen una fuerte presencia.

3 Problema del lenguaje natural. En la época actual gente de todo el mundo trabaja en equipos internacionales, de modo que muchas veces los integrantes tienen idiomas diferentes o regionalismos propios.

4 Problema técnico. En el diseño de sistemas, la contribución del desarrollador y el tiempo de los programadores se invierten en pensar cómo adaptar mejor las ideas para producir una implementación eficiente.

Pegasus posee un modelo limitado del pensamiento humano donde el concepto de idea se describe como una percepción que el programador tiene del mundo externo. Como una idea es una percepción personal que varía entre dos programadores, los autores definen una percepción como un conjunto de ideas. Las ideas atómicas son aquellas que no se descomponen en ideas más simples; por lo tanto, una mesa es un conjunto de ideas que son atómicas, como el color o un pánel horizontal, o ideas compuestas, como la textura, que se define por el elemento del que se construyó la mesa (madera, por ejemplo), o un olor

particular. Los autores proponen la *notation idea* para formalizar ideas. A continuación se presenta un ejemplo de este concepto, que se tomó de [Knöll y Mezini, 2006]:

```
wood: [smell of wood, warmth, solidness, brown color]
table: [horizontal panel, vertical bars, wood, solidness]
```

Este ejemplo define dos ideas complejas, con ideas más simples que se especificaron entre corchetes. Además, la idea *wood* es parte de la idea *table*, de modo que esta idea se refine de la siguiente forma:

```
table: [horizontal panel, vertical bars, [smell of wood, warmth, solidness,
brown color], solidness]
```

Una idea compuesta se define como la composición de representaciones. En este caso, por ejemplo, las ideas *table* y *brown* se convierten en ideas compuestas, como se muestra a continuación:

```
(table, brown)
```

Que se extiende de la siguiente forma:

```
([horizontal panel, vertical bars, [smell of wood, warmth, solidness,
brown color], solidness], brown)
```

Siempre y cuando una idea compleja sea la asociación de ideas complejas o atómicas, las ideas compuestas asociarán de forma efectiva varias ideas complejas en algún momento particular.

Pegasus es una herramientas que permite texto en lenguaje natural, realiza un análisis para devolver código ejecutable. Pegasus utiliza tres características básicas para funcionar: lee lenguaje natural; genera código fuente; y, expresa lenguaje natural. Los autores describen que tales características conforman un *brain* que representa los conceptos en una notación de ideas.

El cerebro de Pegasus posee tres partes:

1 Mente: se enfoca en entender el significado de una oración.

2 Memoria a corto plazo: una cola de ocho posiciones, donde el último elemento agregado se elimina cuando se agrega uno nuevo.

3 Memoria a largo plazo: un diccionario de datos donde las ideas, sus variaciones gramaticales y semántica se definen.

Pegasus analiza la estructura gramatical de una oración para obtener sus ideas y relaciones, y entonces identifica el contexto de la oración. Una idea aparece como una entidad, acción o propiedad. Pegasus analiza una idea comparando el significado de la idea de la oración abstraída con ideas que se almacenan en el diccionario de entidades. Entonces, Pegasus descompone una idea en sub ideas para resolver el significado. Una vez que una idea se resuelve, Pegasus genera código Java, aunque esto no se limita a dicho lenguaje. Para generar texto en otro lenguaje natural, Pegasus compara la entrada con el diccionario de ideas y genera la salida.

3.20. Prototipo de Little y Miller

En [Little y Miller, 2006], los autores describen un mecanismo para la traducción de instrucciones simples en código ejecutable que se relacione con un dominio particular. Los autores presentan dos prototipos – un navegador basado en Web y otro basado en Microsoft Word.

Similar a la programación por demostración (programming-by-demonstration, PbD), la traducción se realiza sin que el usuario conozca la forma en que el script funciona. Este enfoque permite la implementación de mecanismos de descripción y llenado que se aplican a formularios para prácticamente cualquier sitio Web. El usuario además, opcionalmente lleva un registro de sus acciones en una lista de comandos que trabajen en el sitio Web, inclusive para tareas de llenado de formularios y que se compartan con otras personas.

Se describen las funciones expresivas como *left margin two inches*, que el sistema traduce a código Visual Basic como *InchesToPoints*, esto se debe a que la palabra *inches* se comparte en ambas descripciones. Como el sistema infiere que esa es la función más apropiada en el contexto de la oración, se devuelve como resultado *ActiveDocument.PageSetup.LeftMargin = InchesToPoints(2)*.

El sistema infiere secuencias de cadenas como *UIST 2006 into search textbox*, que se traduce a *UIST 2006* y que se ejecuta con un *enter* implícito en la caja de texto con el nombre *search*.

El sistema tolera funciones anidadas, de modo que expresiones como *pick the 4GB RAM option* se traducen de forma exitosa en *pick(findOption("4GB RAM"))*. Este anidamiento se limita a cadenas continuas, de modo que expresiones como *pick option 4GB RAM* se

infieren correctamente, pero no expresiones como *option pick 4GB RAM*. Esto se debe a que *findOption* recibe un string, mientras que pick no lo hace, lo que genera *findOption()* como resultado y por tanto, un error en la generación de código.

El prototipo utiliza varios formatos para valores tales como *2*, que se reemplaza por *2nd, two or second*, por mencionar algunos. Mientras que las comillas son opcionales para strings, su uso reduce la ambigüedad. El sistema trata los argumentos de varias formas, una de ellas es el nombrado explícito de los mismos. Por ejemplo, mientras que para la definición de la función *click(integer x, integer y)*, es válido usar la oración *click 300 y 200* x y sus argumentos son ordenados, la oración *200 300 click* también es válida.

3.21. MOOIDE

En [Lieberman y Ahmad, 2010], los autores presentan MOOIDE, que es una herramienta para la creación de MOOs por medio de lenguaje natural. MOOIDE se conforma de un analizador sintáctico que analiza texto en inglés y provee información a partir de una base de conocimiento. Con MOOIDE, los usuarios agregan nuevos elementos para que otros usuarios interactúen con ellos. MOOIDE utiliza resolución de anáforas y *Common-sense knowledge* por medio de *Open Mind Common Sense* para garantizar que los objetos y sus características sean texto coherente. A continuación se presenta un ejemplo:

```
There is a chicken in the kitchen.
There is a microwave oven.
You can only cook food in an oven.
When you cook food in the oven, if the food
is hot, say "The food is already hot."
Otherwise make it hot.
```

MOOIDE aprende acerca de situaciones, de modo que si se escribe esto:

```
Cook chicken in microwave oven
```

El sistema asocia la oración con las reglas que se definieron antes. En este caso, se devuelve un error porque no se estableció una relación entre *microwave oven* y *chicken*. Lo que se resuelve al escribir lo siguiente:

```
People eat chicken
```

Entonces, el sistema asocia esto con la instrucción anterior y entonces imprime *the chicken is now hot*.

Como MOOIDE se restringió a propiedades del mundo real, no se definen cosas que se basen en magia o elementos sobrenaturales dentro del contexto del juego, y se devuelve código Python como ejecutable.

3.22. Macho

En [Cozzie, Finnicum, y King, 2011], los autores presentan Macho, que es una herramienta que facilita la generación de código fuente a partir de instrucciones simples en lenguaje natural. Macho analiza texto y lo transforma en consultas ejecutables para obtener fragmentos de código a partir de una base de datos que se llenó con ejemplos de código fuente y que se integran para devolver programas funcionales.

Dadas sus características, Macho sólo permite comandos simples en formato UNIX sin opciones, y uno o dos argumentos. Con base en la generación automática de código de los requisitos definidos por el usuario, Macho se enfoca en el análisis de requisitos y generación de código Java. Para reducir la ambigüedad, Macho utiliza fragmentos combinados.

Macho analiza, procesa y transforma instrucciones en lenguaje natural sencillo en consultas para seleccionar los fragmentos de código fuente que sirvan para construir el programa. Para hacer esto, posee cuatro subsistemas.

- Analizador sintáctico de lenguaje natural. Analiza verbos, sustantivos, preposiciones, abreviaturas y conjugaciones que contengan información relevante para la implementación de código.
- Base de datos. Una biblioteca que almacena más de 1,200 fragmentos de código a partir del cual los fragmentos candidatos se seleccionan.
- Stitcher (engrapadora). Combina los fragmentos de código para generar programas candidatos.

Macho combina fragmentos de código al analizar dos expresiones por medio de una variable, si el tipo de la salida coincide con el tipo de la entrada, o por medio del subsistema *Stitcher* fragmentos de código para la conversión de tipos. Macho provee un limitado flujo de control, donde el analizador sintáctico de lenguaje natural genera expresiones *if* y el sintetizador infiere, generando ciclos si el sistema lo sugirió. El depurador analiza y clasifica código en los siguiente cinco casos:

1 Excepción (el código se insertó en un *if*).

2 Una salida correcta del superconjunto (*print* se inserta en un *if*).

3 Basura (se prueba el siguiente programa de prueba generado).

4 Salida del suconjunto correcta (algunas impresiones posteriores se prueban).

5 Salida correcta (termina el proceso).

Macho genera código fuente a partir de oraciones expresivas en lenguaje natural. Dado que las oraciones son ambiguas, el programador debe proporcionar a Macho fragmentos de código. Macho ayuda al programador sin reemplazarlo porque su objetivo no es resolver la ambigüedad. Como se reporta en [Cozzie y King, 2012], los autores realizaron un estudio en Macho, del cual se encontró que de 69 casos, Macho generó 55 salidas correctas en un tiempo que varía de dos a 20 minutos.

En [Cozzie y King, 2011], los autores describen un trabajo temprano en una actualización que denominaron Macho II, donde la computadora "aprende" a generar composiciones a partir de patrones que derivan en código Java. Macho II trabajará por medio de técnicas de procesamiento de lenguaje natural y modelos probabilísticos para código Java. Con esto en mente, los autores mencionan que es posible que un modelo aprenda a predecir con base en operaciones para generar código en casi cualquier lenguaje.

3.23. Aiaioo NLC

En [Carlos, 2011], el autor presenta Aiaioo NLC, un sistema de programación en lenguaje natural que utiliza reglas secuenciales de clase. El sistema se limita a operaciones con números reales, en un proceso que consiste en descomponer las instrucciones en sentencias y su posterior clasificación en categorías. Con base en esta clasificación, el sistema extrae entidades para su uso en la generación de código, mientras que para el reconocimiento de lenguaje natural, este facilita la creación de nuevas reglas y el reconocimiento de nuevas instrucciones que el sistema clasifica en categorías existentes y que incluso se traducen a otros lenguajes por medio de la habilidad de aprendizaje del sistema.

El autor describe la "proceduralización" como la construcción de procesos por medio de pasos, bloques, condicionales y ciclos. Las palabras imperativas son verbos que indican llamadas a funciones o comandos de cambio. El sistema utiliza preguntas para evaluar una variable o expresión. El autor solicitó entradas de prueba vía Internet para expresar instrucciones en lenguaje natural, entonces clasificó las respeustas de acuerdo a las reglas de

notación descritas: primitivas condicionales expresan condiciones o acciones; primitivas de iteración evalúan primitivas de forma repetida; operaciones con efecto lateral que cambian el estado de una variable; operaciones sin efecto lateral que no cambian el estado de una variable; y, operaciones relacionales que comparan valores. El autor define las reglas secuenciales de clase como una secuencia ordenada de tokens donde una instrucción es correcta sólo si su regla se evaluó correctamente.

3.24. Lenguaje hablado similar a C

En [Gordon y Luger, 2012], los autores presentan un lenguaje de programación imperativo y similar a C que es fácil de recitar a través de un procesador de voz en inglés y de forma simple. Este lenguaje posee una gramática típica, pero omite puntuación y símbolos impronunciables, con esta interfaz se libera al usuario de la tarea de declarar y compilar de forma tradicional. El lenguaje se probó como un plug-in de Eclipse por medio de CMU Sphinx como motor de reconocimiento de voz y ANTLR para la definición de la gramática. Los autores describen cómo la mayoría de los motores de reconocimiento de voz carecen de soporte para los lenguajes de programación tradicionales y presentan dos enfoques para resolver este problema. El primero describe modelos de programación con motores de voz integrados, mientras que en el segundo, la herramienta genera código a partir de construcciones en inglés.

3.25. Automatización de pruebas automáticas

En [Thummalapenta, Sinha, Singhania, y Chandra, 2012], los autores presentan una técnica para automatizar las pruebas automáticas de sistemas de software que realizan testers quienes, por lo general no son programadores. Los testers realizan pruebas con base en casos de pruebas sin perder informalidad, flexibilidad y la ambigüedad relacionada al lenguaje natural, mientras que, al mismo tiempo mantienen la exactitud de los lenguajes de scripting. Los testers generalmente describen casos de prueba en lenguaje natural que se capturan en los requisitos, y casos de uso que se traducen a scripts, que entonces validan la calidad del sistema. Los testers realizan estas tareas de forma manual.

Los autores proponen un mecanismo para extraer relaciones entre acciones y elementos de la interfaz de usuario sin utilizar lenguaje natural avanzado. En lugar de esto, los autores proponen que se empleen descripciones paso a paso donde el tester describa sus interacciones con la interfaz de usuario. El problema principal de esto es que dichas descripciones aún contienen ambigüedad, de modo que se requieren heurísticas y reglas de aplicación específicas.

Este enfoque trata con la ambigüedad al explorar alternativas para obtener un resultado o, cuando el sistema llega a un punto sin rolución, realizar un retroceso a la última decisión, donde el sistema tomará la otra alternativa. Así, el sistema analiza una prueba que se divide en pasos que tiene objetivos identificados, y donde, si se encuentra ambigüedad, se genera una bifurcación. Los últimos dos pasos consisten en la identificación de los segmentos que se hacen referencia a los elementos de la interfaz, y su ejecución en el navegador con las decisiones correspondientes para explorar diferentes flujos.

3.26. Programming by example-based and Natural Language Hybrid

En [Manshadi, Gildea, y Allen, 2013], los autores presentan un framework que utiliza descripciones en lenguaje natural acerca del curso de uno más ejemplos. Con esto en mente, los autores proponen que la "programación por ejemplo" (*Programming by Example*, PbE) se refuerce con el uso de dichas descripciones para obtener la salida que se espera.

Para alcanzar este objetivo, el framework utiliza descripciones en lenguaje natural ambiguas que se complementan con ejemplos que resuelven la ambigüedad, lo que genera pares de entrada-salida para todas las soluciones posibles, entonces, se selecciona la mejor opción con base en qué tan bien se adaptó a partir de las descripciones en lenguaje natural.

Para resolver un problema a partir de la PbE, el framework descompone el problema en problemas más sencillos que se resuelven de forma individual con las restricciones que se definen para cada uno. Mientras que el sistema utiliza Space Algebra para tratar esta descomposición, este devuelve una cantidad casi infinita de composiciones de expresiones regulares; por tanto, estas expresiones se intersectan para todos los pares. Además se utiliza un método PbE probabilístico (*probabilistic PbE*, PPbE) para reducir el número de soluciones posibles.

Para el componente de lenguaje natural, los autores utilizaron un analizador sintáctico de dependencias para cada descripción, de este modo no sólo se codifica la estructura sintáctica, sino también la estructura semántica.

3.27. SmartSynth

En [Le, Gulwani, y Su, 2013], los autores describen una herramienta para generar scripts que se ejecuten en un teléfono inteligente. SmartSynth produce scripts en SmartScript, un DSL que se creó para este propósito. Como SmartSynth procesa descripciones en lenguaje natural, sino que también funciona como un sintetizador de programas que sirve como traductor entre el usuario y el teléfono.

En SmartSynth, el usuario ingresa descripciones en lenguaje natural que se traducen con un algoritmo y obtiene elementos que se relacionan con la API a partir de dicha entrada. Después de esto, el sistema interactúa con el usuario para resolver elementos ambiguos o desconocidos. Entonces, SmartSynth relaciona las descripciones y devuelve un script ejecutable.

Once estudiantes probaron SmartSynth y produjeron 750 descripciones diferentes para 50 tareas, de estas descripciones 640 coincidieron con las especificaciones y se emplearon para validar SmartSynth. Con base en esta muestra, SmartSynth identificó 64.3 % de los componentes únicamente a partir del algoritmo base, mientras que la precisión fue de 77.1 % al combinar todas las características, y un 90.3 % con la interacción del usuario. Como desventaja, la resolución basada en reglas provee únicamente un 86 % de precisión. Al tomar esto en cuenta, se obtuvo un 58.7 % de precisión al utilizar únicamente técnicas que se relacionen con la NLP, mientras que al utilizar un sintetizador de programas dicha precisión se incrementó al 90 %.

3.28. Natural Language Command Interpreter (NLCI)

En [Landhäußer y cols., 2016], los autores presentan Natural Language Command Interpreter (NLCI), que se basa en una ontología que modela una API que traduce entradas en lenguaje natural a intenciones de API, es una herramientas que se enfoca en generar código ejecutable a partir de descripciones en inglés. Los autores utilizaron dos APIs para

probar NLCI, openHAB y Alice, con NLCI construyendo llamadas a la API con una exactitud del 67 y 78 % para 50 scripts. La ontología sirve como puente entre el lenguaje y el código que implementa su significado.

NLCI se enfoca a resolver problemas no secuenciales que ocurren cuando dos o más interlocutores expresan ideas de forma no secuencial, lo que significa que no es necesario expresar la descripción de un programa en su orden de ejecución. Este problema se observó cuando se indicó a niños que utilizaran Alice, que es una herramienta 3D que se diseñó para enseñar a los niños a programar animaciones[4]. Una limitante de NLCI es su incapacidad para trabajar con elementos temporales, de las cuales se describen tres tipos de expresiones en inglés: tiempo, preposiciones temporales y adverbios temporales [Landhäußer, Hey, y Tichy, 2014].

En una primera prueba de NLCI que se describe como "AliceNLP", la herramienta divide texto en pasos ordenados, los analiza y organiza de acuerdo a las expresiones temporales que encuentre. NLCI tiene una tasa de éxito de un 86 % a partir de 24 pruebas con la implementación de Alice, con una línea de tiempo que se estableció correctamente en 17 pruebas. Además, NLCI no se limita a trabajar con Alice porque su enfoque se basa en una ontología para la descripción de una biblioteca, de modo que si se cambia la ontología, es posible utilizar otra biblioteca [Landhäußer y Hug, 2015].

Posteriormente, los autores realizaron pruebas tanto para software basado tanto en openHAB, como en Alice. En openHAB, realizaron cinco scripts con 15 comandos que se basan en la ontología que se generó a partir de la configuración del sitio Web. Para alimentar la ontología basada en Alice, los autores programaron diez animaciones y dos escenarios estáticos que programadores y no programadores describieron por medio de lenguaje natural, después que esta información se procesó con NLCI, se devolvió una secuencia de llamadas a la API. Los autores crearon una muestra de 86 scripts, con sólo 50 utilizables para animaciones, mientras que 30 se ejecutaron para escenarios estáticos y seis se descartaron. El nivel general de precisión fue de 78 %, con un registro de 67 % para llamadas a la API. Dado que openHAB y Alice no proveen soporte para tratar con precondiciones y postcondiciones, NLCI no resolvió elementos temporales, pero los autores expresaron su interés en mejorar NLCI para integrar un procesador de voz.

[4]La universidad de Saarland desarrolló el lenguaje Alice ML como un dialecto de Standard ML [University, 2014], sin embargo, en este artículo se utiliza el término "Alice Language" únicamente para la herramienta que se desarrolló en la universidad Carnegie Mellon.

3.29. NLyze

En [Gulwani y Marron, 2014], los autores presentaron NLyze, una interfaz basada en lenguaje natural para hojas de cálculo. Esta interfaz se basa en un DSL que utiliza cálculos algebraicos y diversos mecanismos para manejo de tablas y que se enfoca a usuarios inexpertos. Se implementó como un plug-in para Microsoft Excel.

Los autores diseñaron un DSL que fuera suficientemente expresivo para resolver tareas mientras que fuera suficientemente restrictivo como para mantener una traducción eficiente. Con esto en mente, el DSL permite rastrear, filtrar y reducir operaciones. Un algoritmo de nombre Translate recibe una instrucción en lenguaje natural y una hoja de cálculo, entonces devuelve un conjunto clasificado de programas candidatos que se generaron con el DSL y que son interpretaciones posibles de la instrucción a ejecutar sobre la hoja de cálculo. Este algoritmo contiene a su vez otros dos algoritmos, *Synth* y *Rule*.

Synth se enfoca en composiciones basadas en tipo, las genera al aplicar una descomposición recursiva para obtener fragmentos más pequeños. *Rule* se enfoca en un conjunto de reglas de patrones comunes para devolver instrucciones basadas en palabras a partir de la instrucción. *Rule* se limita a la sintaxis y falla cuando recibe instrucciones con palabras fuera del patrón esperado, que *Synth* complementa para incrementar la eficiencia de traducción del algoritmo.

Mientras que SmartSynth no resuelve la ambigüedad, devuelve programas candidatos con base en la entrada del usuario y la descripción en lenguaje natural para todos los programas candidatos, de modo que el usuario los entienda de forma fácil.

3.30. Framework NL a DSL

En [Desai y cols., 2016], los autores presentan un framework que toma entradas en lenguaje natural y devuelve varios bloques de código en un DSL que la herramienta califica de acuerdo a qué tan cerca se encuentran del resultado que el usuario espera con base en su entrada. Los autores indican que en el 80 % de los casos, se mostró como primera opción el código correcto, mientras que este se encontró en las tres primeras opciones en el 90 % de las pruebas, que se llevaron a cabo a partir de una muestra de 1272 entradas en lenguaje natural.

Este framework es independiente del DSL y su especificación, además de que se requiere de entrenarlo por medio de pares texto-código. Este framework utiliza un algoritmo de síntesis que genera programas candidatos a partir de texto, se basa en la relación entre terminales y palabras en el texto. El diccionario de datos establece dichas relaciones durante la etapa de entrenamiento, además de que filtra errores ortográficos, resuelve homónimos y, finalmente, asigna una puntuación para cada programa con base en qué tan cerca se encuentra la entrada del resultado que espera el usuario. Además, es digno de mención que este framework utiliza las técnicas que se presentaron en SmartSynth y NLyze, de modo que se le considera como un refinamiento o extensión de ambas herramientas, ya que a diferencia de aquellas, este framework no se limita a un DSL.

Los autores lo implementaron con tres dominios: editores de texto con Microsoft Excel, donde un DSL manipula la hoja de cálculo; un autómata para problemas de tutoría inteligente en el cual además se diseñó un DSL; y, un sistema ATIS para tráfico aéreo, para el cual se diseñó un DSL similar a SQL.

3.31. ReDSeeDS

En [Mefteh, Bouassida, y Ben-Abdallah, 2018], se presenta ReDSeeDS, una herramienta que toma una especificación de requisitos en lenguaje natural sin importar la lengua por medio de una versión enriquecida del modelo semántico. Los autores mencionan que dicho enriquecimiento se logró por medio de patrones en XML que presentan la información de los escenarios de prueba, de modo que se eliminan las ambigüedades en el significado y de esta forma se genera código Java.

ReDSeeDS utiliza casos de uso RSL como tipo especial de requisitos para los modelos de requisitos, donde los actores se relacionan con los casos de uso pormedio de los estereotipos «use», «participate» e «invoke». Por otro lado, ReDSeeDS trabaja con precondiciones, oraciones en formato SVO (sujeto, verbo, objeto), operaciones de invocación, condición, finalización y reunión.

ReDSeeDS utiliza además un modelo del dominio RSL que en conjunto con el modelo de requisitos RSL genera como resultado código Java. Los elementos del modelo del dominio son los siguientes:

```
Phrase := NounPhrase | VerbPhrase
```

```
NounPhrase := [Modifier] + Noun
VerbPhrase := SimpleVerbPhrase | ComplexVerbPhrase
SimpleVerbPhrase := NounPhrase + Verb
ComplexVerbPhrase := NounPhrase + SimpleVerbPhrase
```

Entre sus resultados, los autores mencionan que el código generado tiene una precisión de entre el 80.49 % y el 89 % en términos de los modelos de dominio RSL y de requisitos RSL, con una correctitud para el modelo de dominio RSL del 91.74 % y una correctitud promedio del 95.55 %

3.32. Lenguajes de cuarta generación

El término lenguaje de cuarta generación (*fourth-generation languages*, 4GL) generalmente se asocia con lenguajes empresariales de manipulación de datos que se optimizaron para el acceso a base de datos, generación de reportes e interfaces gráficas [van Deursen y cols., 2000]. Debido a su enfoque empresarial, los desarrolladores de lenguajes crearon los 4GL con un mayor nivel de expresividad que los lenguajes tradicionales y abstracciones más restringidas; sin embargo, estos lenguajes no poseen un estándar de diseño, de modo que todo 4GL posee su propia gramática y propiedades. En esta sección se revisan varios 4GLs que poseen características de lenguajes naturales.

3.32.1. Advanced Business Application Programming Language

Advanced Business Application Programming Language (ABAP/4) es un lenguaje enfocado a los reportes que utiliza procedimientos para diseñar aplicaciones empresariales complejas. ABAP deriva de SAP R/2 y se encuentra embebido en SAP R/3 [Kemper, Kossmann, y Zeller, 1999]. ABAP/4 permite acceso a base de datos por medio de Native SQL y OpenSQL, además de que funciona con eventos. Soporta programación orientada a objetos por medio de ABAP objects, que posee características similares a C *"structs"* de C, y permite la definición de instrucciones similares a las de COBOL [SAP-AG, 2014]. ABAP provee mecanismos para describir instrucciones verbosas como *ADD 2 TO result*, que es el equivalente naturalístico para *result += 2*; cabe destacar que las instrucciones en ABAP equivalen a una oración en inglés con sujeto elíptico.

3.32.2. Informix-4GL

Informix-4GL es un lenguaje enfocado a los reportes con un soporte limitado para operaciones imperativas, que se diseñó para asemejarse a un lenguaje natural. Produce como salida tanto bytecode de Java como código C, Informix-4GL es un lenguaje muy popular debido a sus clones, que poseen funcionalidad extendida tales como interfaces gráficas o traductores de código Java, siendo el más destacable Aubit-4GL [Aubit-4gl team, 2001]. Los usuarios generan reportes y los conectan con diversos gestores de bases de datos. Informix 4-GL posee un robusto soporte para interfaces gráficas por medio de las cuales el programador describe tareas con una sintaxis similar al inglés, por ejemplo *MOVE WINDOW*, *NEXT FIELD "fieldname"* o *NEXT FORM NEXT OPTION "optname"*, por mencionar algunas [O'Gorman, 2010].

3.32.3. Visual FoxPro

Visual FoxPro (VFP) es un lenguaje de programación orientado a datos que Microsoft desarrolló a partir de FoxPro, pero que posee la capacidad para integrar interfaces gráficas de forma simple. Microsoft diseñó VFP como una herramienta para crear aplicaciones Windows dependientes de datos, VFP posee un soporte integrado en su gramática para bases de datos y despliega información con facilidad gracias a su sintaxis simple [Hennig, Schummer, Slater, Granor, y Feltman, 2005]. Debido a que VFP se enfoca a la construcción de interfaces de usuario, posee una gramática imperativa que permite trabajar con entornos gráficos y bases de datos con una formal, pero expresiva gramática donde la instrucción *DEFINE POPUP table MARGIN RELATIVE SHADOW COLOR SCHEME 4*, describe una tabla emergente de nombre *tables*, que posee un *margin relative* con un *shadow color scheme* con valor de *4* [Pinter, 2004].

3.32.4. Nomad Software

Nomad Software es un lenguaje relacional orientado a bases de datos que combina un procesamiento por lotes con una ejecución interactiva. Permite definir bases de datos, manejo de información y generación de reportes, además posee un lenguaje orientado a base de datos para manipular los reportes. A diferencia de otros lenguajes, Nomad se enfoca al usuario final, por esto sus usuarios son generalmente grandes corporaciones que

utilizan los lenguajes en ciclos de producción por lotes, aplicaciones Web y aplicaciones de escritorio para la generación de reportes. Es un lenguaje intuitivo con un entorno de desarrollo interactivo en el que una instrucción se ejecuta en el momento que se escribe. Además, Nomad genera bases de datos relacionales y normalizadas que son compatibles con diversos gestores de bases de datos, tales como VASM, IMS, IDMS, DB2, Oracle y SQL Server [McCraken, 1980].

3.32.5. SAS

SAS es un lenguaje PL/I basado en sintaxis desarrollado por el instituto SAS, se enfoca en el análisis de datos estadísticos que se encuentran en bases de datos o tablas. SAS muestra estos datos en forma de gráficas o documentos, analiza y modifica tablas de varias maneras para presentar los resultados en forma de reportes, y además analiza código SQL. Un detalle notable es que SAS no posee restricciones en el uso de palabras reservadas, de modo que se permite que el programador las emplee como identificadores. El uso sin restricción de palabras reservadas resulta en un lenguaje ambiguo que requiere de instrucciones que se definan con anterioridad y que dependan del contexto. SAS contiene de forma embebida al lenguaje R[5] desde la versión 9.2, que permite el uso de sus características [Li, 2013].

3.32.6. OpenEdge ABL

OpenEdge ABL es un lenguaje que depende de los datos y que posee una sintaxis similar al inglés con el cual los usuarios finales desarrollan aplicaciones sin saber un lenguaje de programación. Se basa en interfaces gráficas y consultas de datos, posee una sintaxis expresiva que permite la construcción de consultas, inserciones, modificaciones y el despliegue de tareas simples [Sadd, 2006].

[5]R es un lenguaje orientado a objetos que se enfoca en estadísticas, recibe influencia de S y Scheme. Se enfoca en realizar pruebas estadísticas, análisis de series de tiempo, clasificación y agrupación de algoritmos, y modelos lineales y no lineales [The R Foundation, 2016].

3.32.7. LiveCode

LiveCode es un lenguaje que se basa en HyperTalk que se presentó originalmente bajo el nombre de Revolution pero, en 2010 los autores cambiaron su nombre a LiveCode. LiveCode es un lenguaje fácil de aprender que se ejecuta sobre varias plataformas tales como Android, Apple o Windows. LiveCode posee una sintaxis similar al inglés con tipificación dinámica y código suficientemente expresivo como para que se le considere autodocumentado y adecuado para el aprendizaje de programación a partir de los 14 años [LiveCode Ltd, 2016].

3.33. Comparativas

En esta sección se presentan dos tablas con características similares, pero que se enfocan a conceptos diferentes. Mientras que Tabla 3.1 describe tecnologías que contienen además gramáticas, la Tabla 3.2 describe aquellas tecnologías que únicamente son lenguajes que forman parte de una herramienta.

3.33.1. Herramientas

Como se observa en Tabla 3.1, el objetivo principal de estas tecnologías es la resolución de actividades relativas a un dominio particular. Otras poseen un enfoque de propósito general, pero integran elementos naturalísticos para facilitar la lectura a los programadores. Cabe destacar que varias de estas tecnologías utilizan gramáticas predefinidas para evitar ambigüedad, además de palabras reservadas con o sin diccionarios de datos. Algunas de estas herramientas generan código en algún GPL a través de fragmentos de código en uno o más lenguajes. Otra característica común es que utilizan referencias indirectas de forma muy limitada y por medio de pronombres como *it*. Con sólo dos tecnologías que abordan el problema que aparece cuando un programador no es angloparlante nativo, los autores de *AppleEvents* y [Knöll y Mezini, 2006] proponen el desarrollo de software en lenguajes diferentes al inglés. Otro detalle a destacar es que varias de estas tecnologías se enfocan a la industria, por tanto sus características buscan favorecer la productividad. Por otro lado, algunas herramientas se enfocan al aprendizaje por medio del uso de características del lenguaje natural, que estas integran de forma limitada para utilizarlas como un mecanismo

Tabla 3.1: Propiedades de las herramientas que se revisaron

Herramientas	Generación automática de código	Palabras reservadas	Gramática predefinida	Diccionarios de datos	Fragmentos de código predefinidos	Soporte a múltiples PL	Soporte a múltiples idiomas	Referencias indirectas	Enfocado a la industria	Enfocado a la educación	Enfocado a la documentación
Heidorn	*	*	*								
NLC			*					*		*	
LDC		*	*	*				*			
TELI		*	*	*				*	*		
HyperTalk		*	*						*		
AppleScript		*	*	*			*	*	*		
ACE		*	*	*				*	*		*
NaturalJava	*	*		*	*			*	*		
HANDS		*	*					*		*	
Chong's framework	*		*			*					
Metafor	*		*		*	*		*		*	*
Spoken Java		*	*						*	*	
CPL/CPL-Lite	*	*	*	*				*			*
Vadas' prototype	*	*	*					*			
Pegasus	*		*	*	*	*	*	*	*		*
Little's prototype	*		*	*		*			*		
MOOIDE	*		*	*							
Macho	*			*	*			*	*		*
Aiaioo NLC		*	*	*					*	*	
Spoken C		*	*						*	*	
PbE NL hybrid	*				*	*					
SmartSynth	*							*			
NCLI	*	*		*	*	*		*			
Nlyze	*								*		
Desai's NL to DSL	*				*	*		*			
Total	15	14	18	12	7	7	2	15	12	6	5

eficiente para el desarrollo de software. Por ejemplo, la herramienta HANDS deja de lado la eficiencia para que su ejecución sea lenta comparada con otras herramientas, pero eficiente desde la perspectiva didáctica. Por último, algunas herramientas se enfocan en la creación de código autodocumentado o que se traduzca en lógica de orden superior.

De todas las tecnologías que se revisaron, se observó que la generación automática de código es el problema principal que los autores buscan solucionar. Pegasus es la única herramienta que los autores definieron como un generador automático de código a partir de lenguaje natural controlado que se restringe por la definición de conceptos en inglés y alemán por medio de un diccionario de entidades. Más aún, una línea de investigación a futuro indica que Pegasus sería capaz de soportar otros lenguajes tales como español, francés o árabe. El objetivo principal en [Knöll y Mezini, 2006] fue resolver problemas de desarrollo que se observaron por los autores cuando los equipos de desarrollo se conforman por personas de diversas partes del mundo y necesitan trabajar en equipo inclusive cuando tienen lenguas nativas diferentes.

Macho es una herramienta que genéra código Java a través de consultas en lenguaje natural, devuelve código corto y muy específico por medio de un proceso iterativo que toma varios minutos para completarse y devolver bloques de código almacenados en una base de datos. Mientras que Macho genera código, no permite la programación directa por medio de un lenguaje naturalístico, esto se debe a su enfoque a la automatización de código. Además, carece de un mecanismo eficiente para lidiar con la ambigüedad. Los autores indicaron que trabajaban en una reestructuración mayor de Macho que emplea aprendizaje automático y un lenguaje de flujo de datos gráfico.

Metafor es otra tecnología cuyo objetivo principal es ayudar a los programadores a describir problemas por medio de lluvias de ideas, en las que el programador refleja un problema en forma de bosquejos de código Python. Metafor se basa en la retroalimentación por parte del programador por medio de diálogos para fortalecer la descripción de código. De forma similar a Pegasus, Metafor se enfoca a ayudar a programadores, en lugar de ser una alternativa a un paradigma.

NLC es una herramienta cuya implementación se reportó hace varias décadas, de modo que su contexto es diferente al resto. Su objetivo principal se enfoca a trabajar únicamente con arreglos que se muestren en pantalla, restringiendo su uso a un dominio muy estrecho y, además, controlando la ambigüedad cuando se hace referencia a muchos elementos.

NLCI es una herramienta que se enfoca a la traducción de comandos en lenguaje natural a código ejecutable. Como se mostró de la sección 3.28, es una herramienta de dominio específico donde la temporalidad se reporta como un problema a resolver en implementaciones futuras. NLCI además implementa varios DSL con sólo cambiar la ontología.

LDC y TELI son herramientas para el manejo de base de datos que aprenden del inglés en lugar de ser herramientas de programación por sí mismas, por tanto carecen de las características que se esperan de una herramienta que se enfoque al desarrollo de software. Esto es relevante debido a que se utiliza lenguaje natural en consultas que se basan en adjetivos, lo que permite instrucciones de código complejas que se basan en frases si se utilizan desde un enfoque de lenguajes de programación.

Aiaioo NLC es una herramienta que se limita a trabajar con operaciones sobre números reales, donde se utiliza el reconocimiento de comandos y reglas de traducción de lenguajes (lenguajes naturales), lo que permite utilizar verbos imperativos como procedimientos y preguntas como condicionales. El dominio de esta herramienta es muy específico y no se hace mención de cómo lidia con la ambigüedad dentro del mismo.

Attempto Controlled English es un subconjunto formalizado del idioma inglés que se utiliza para la documentación de artefactos de software. Se desarrolló debido a la necesidad por un mecanismo libre de ambigüedades para el idioma inglés que sea traducible a una especificación formal. Mientras que la principal desventaja consiste en que es una herramienta que se enfoca a la documentación y que no se reportan sistemas de generación de software, es un buen ejemplo de la importancia de la correcta documentación durante el desarrollo de software.

3.33.2. Lenguajes

En Tabla 3.2 se presentan las características naturalísticas de los lenguajes que se revisaron. Como se esperaba, muchos de estos se enfocan a la industria y poseen características para la generación de reportes. Debido a su enfoque, estos lenguajes poseen expresividad similar al inglés y algunos elementos de los lenguajes naturales. Muy pocos se enfocan al aprendizaje y sólo uno se reporta explícitamente para este propósito. Algunos de estos lenguajes tienen la habilidad de generar código en uno o más GPL, pero requieren diccionarios de datos, con excepción del lenguaje SAS.

Tabla 3.2: Propiedades de los lenguajes que se revisaron

Languages	Generación de código	Diccionario de datos	Expresividad similar al inglés	Referencias indirectas	Enfocado a la industria	Enfocado al aprendizaje	Generación de reportes
FLOW-MATIC			*				
COBOL	*		*		*		*
CPL/CPL-Lite	*	*	*	*			
Spoken C				*			*
Spoken Java				*			*
HyperTalk			*	*	*		
SQL					*		
ABAP/4	*	*	*		*		*
Informix-4GL	*	*			*		*
Visual FoxPro	*				*	*	*
Nomad Software			*		*		*
SAS		*	*		*		*
OpenEdge ABL			*		*		
LiveCode			*	*	*	*	*
MOOSE Crossing	*	*	*	*		*	
Total	7	5	10	6	11	3	10

COBOL es un GPL que se diseñó pensando en el uso militar y empresarial por parte de compañías y sin un enfoque académico o científico, gracias a esto posee una sintaxis expresiva. Es un lenguaje imperativo que carece de soporte para referencias indirectas, además de que se deriva del lenguaje FLOW-MATIC, lo que significa que ambos comparten varias características.

CPL es un lenguaje naturalístico que resuelve la ambigüedad por medio de heurísticas y técnicas de inteligencia artificial. Como limitante, CPL depende de CPL-Lite, que es un lenguaje formalístico. La principal versión de CPL resuelve problemas de ambigüedad con CPL-Lite porque este último se integra dentro del núcleo del primero. Otro punto a destacar es que ambos lenguajes poseen un alto nivel de expresivdiad.

El lenguaje C hablado que se presenta en [Gordon y Luger, 2012] se diseñó con la idea de que los programadores lo reciten, haciendo de esta la única característica que toma de los lenguajes naturales debido a que es imperativo, como C. Otro ejemplo es Spoken Java, que es un superconjunto de Java que se diseñó pensando en la programación por voz. Ambos lenguajes son muy complejos de formalizar debido a la prosodia y al problema de que un hablante no nativo del idioma inglés aún tendrá problemas al momento de pronunciar el código; por tanto, su interpretación se lleva a cabo por medio de analizadores sintácticos.

HyperTalk and AppleScript son dos lenguajes de programación que se basan en el inglés y se enfocan al diseño de interfaces gráficas, de modo que programadores inexpertos sean capaces de desarrollar aplicaciones. Mientras que ambos lenguajes proveen un manejo básico para referencias indirectas, la principal desventaja es que ambos son DSLs. Como HyperTalk se encuentra embebido dentro de HyperCards, no presenta soporte para depuración de código.

Para los lenguajes de cuarta generación se observó una tendencia hacia la resolución de problemas particulares, lo que significa que los lenguajes 4GL carecen de una normalización y más aún, generalmente son lenguajes propietarios y por tanto es difícil obtener y modificar su código fuente para problemas particulares. Los lenguajes 4GL son un mecanismo inflexible para desarrollar aplicaciones que no sólo dependan de datos, sino que tambíen requieran de varios procesos que los diseñadores de los lenguajes no consideraron durante el diseño del 4GL. Mientras muchos lenguajes 4GL resuelven este problema al ofrecer o aceptar bibliotecas que los usuarios crean en otros lenguajes, esto significa que

los lenguajes 4GL requieren de otros lenguajes menos expresivos, mismos que se suponía reemplazarían.

Capítulo 4

Aplicación de la metodología

En este capítulo se describe el modelo que se propuso con base en la investigación que se llevó a cabo, se realizaron pruebas para observar las capacidades naturalísticas de Scala, además del modelo y sus elementos junto con la descripción de un prototipo que se generó a partir del mismo.

4.1. Metodología

En esta sección se describe la metodología que se empleó en el desarrollo de esta tesis.

4.1.1. Análisis

En esta etapa se analizaron diversos artículos científicos donde se presentan trabajos relacionados con la programación naturalística. Además, se revisaron los conceptos lingüísticos que se consideraron indispensables para el diseño de un lenguaje naturalístico. Esta etapa consistió en las siguientes actividades:

1. Revisión de conceptos lingüísticos y de programación.
2. Análisis y clasificación de las implementaciones naturalísticas.
3. Selección de los elementos lingüísticos con base en lo que se reporta en la bibliografía.

4.1.2. Selección

A partir de la bibliografía, se identificaron los elementos comunes más relevantes que se reportan en las implementaciones naturalísticas, las cuales permitieron establecer los primeros elementos del modelo conceptual. Además, se observó que dichos elementos no abordan el contexto, lo que es comprensible dado que las implementaciones se enfocan a resolver problemas de un dominio particular, o lo limitan por medio de técnicas de lingüística computacional. Las actividades que se realizaron son las siguientes:

1. Análisis de oraciones gramaticalmente correctas en el idioma inglés.
2. Identificación de los elementos mínimos que se requieren para conformar oraciones gramaticalmente correctas.
3. Selección de los elementos relevantes para el modelo.
4. Identificación de los elementos que se requieren para que el modelo sea de propósito general.
5. Análisis de la forma en que los elementos seleccionados se interrelacionan.
6. Definición de las restricciones del modelo.

4.1.3. Implementación del modelo

Con base en el modelo, se definió el lenguaje naturalístico SN, que consiste en un analizador sintáctico, un analizador semántico y un traductor. Las actividades que se realizaron son las siguientes:

1. Selección de las palabra reservadas.
2. Diseño del analizador sintáctico.
3. Diseño del analizador semántico.
4. Diseño del traductor a Scala y AspectJ.
5. Diseño de un editor.

4.1.4. Diseño de la API

En esta etapa se diseñó y depuró la API básica del lenguaje. En esta etapa se realizó sólo una actividad:

1. Diseño de la API.

4.1.5. Modelado del lenguaje

En esta etapa se realizó el modelado del lenguaje con el lenguaje funcional Haskell, en esta etapa se realizó la siguiente actividad:

1. Modelado del lenguaje con Haskell.

4.1.6. Depuración del lenguaje

En esta etapa se realizaron las primeras pruebas del lenguaje y se corrigieron errores en la gramática, en el generador de código tanto en Scala y AspectJ, en la API y en el editor. Las actividades de esta etapa fueron las siguientes.

1. Depuración de la gramática.
2. Depuración del generador de código.
3. Depuración de la API.
4. Depuración del editor.

Cabe destacar que las actividades de esta etapa se realizaron en paralelo al diseño del lenguaje, la API, lo que implica que se realizó durante la mayor parte del proceso de diseñor del lenguaje.

4.1.7. Realización de pruebas

En esta etapa se realizaron las pruebas pertinentes al lenguaje para garantizar su correcto funcionamiento, además de que se definieron y desarrollaron los escenarios de prueba que se presentan en la sección 4.6. Cabe destacar que este paso se realizó en

conjunto con una estudiante de licenciatura durante el transcurso de sus residencias, de modo que las actividades fueron las siguientes:

1. Capacitación de la residente.
2. Revisión de ejercicios.
3. Retroalimentación por parte de la residente.
4. Realización de escenarios de prueba.

4.2. Modelo

Con base en los trabajos que se describen en el Capítulo 3, se observó que los autores se enfocaron a resolver problemas particulares. En dichos trabajos, los autores utilizaron principalmente sustantivos y verbos. Aparte de los sustantivos y adjetivos, los interlocutores se expresan por medio de pronombres, que son palabras que reemplazan a un sustantivo o sintagma y cuyo uso depende del contexto, pero en los trabajos que se reportan se utilizaron por lo general los pronombres neutros *it* y *this* para singulares; mientras que para plurales se emplearon *these* o *those*. Se observó que pocos trabajos utilizaron sintagmas nominales (frase cuyo núcleo es el sustantivo) de forma limitada y que se complementaron con adjetivos cuando se requería que un sustantivo tuviera características más específicas. Con lo anterior se infiere que las entidades relevantes para el desarrollo de software se asemejan no sólo a sustantivos, sino a sintagmas nominales que resultan de la especialización que proveen los adjetivos y a las que idealmente se haga referencia por medio de pronombres o incluso por medio de los propios sintagmas nominales. El uso de sintagmas y pronombres permite hacer referencia de forma indirecta a entidades que se definieron con anterioridad, de este modo se expresa en función de relaciones anafóricas, algo común en los lenguajes naturales. Para trabajar de forma eficiente con sintagmas, se requiere del uso intensivo de tipos porque estos reemplazan a los identificadores, ya que en los lenguajes naturales las cosas se describen por medio de tipos, en lugar de identificadores para las cosas a las que hacen referencia, es decir; un programador no dice *sort the list A*, sino *sort the list of numbers*.

Por ejemplo, para la descripción de un sistema que permite la venta de casas se infiere de forma preliminar que las abstracciones más importantes son las casas y los compradores, ambos sustantivos y con características particulares, donde una casa tiene medidas, número de habitaciones, ubicación o precio, mientras que el comprador tiene nombre, teléfono y una lista de casas que le interesen. Si se analiza más a fondo, se encontrará que una casa posee características particulares que las distingan, por ejemplo que la casa sea lujosa o austera, grande o chica y que además tenga características opcionales como estacionamiento o planta, en cuyo caso se trata de un condominio. Además, el estado de las casas y los compradores respecto a la agencia de ventas cambia conforme la casa pasa de "en venta" a "vendida", lo que se describe por medio de verbos para conformar oraciones tales como *la agencia vende casa a cliente*, *la agencia tramita la escritura* o *el cliente programa cita* (si el posible comprador quiere ver la casa en persona). Con base en lo anterior, se obtienen las siguientes abstracciones: sustantivos *house* y *client*; adjetivos *luxurious*, *big* y *sold*; propiedades de la casa *rooms*, *pool*, *garage*, *floor* y *address*; y propiedades del cliente *name*, *telephone* y *address*.

Esta definición permite construir sintagmas como *the house*, *the luxurious house* o *the sold house*; sintagmas preposicionales como *with 3 rooms* o *with pool and garage*; combinaciones de ambos tales como *the luxurious sold house with pool, garage and owner*; además, con estos sintagmas se crean oraciones complejas como *the luxurious and big house is next to the big house with pool* o *the client bought the house with 2 rooms*.

Una circunstancia es una característica de los lenguajes naturales que consiste en la capacidad que tienen para describir cosas de forma no secuencial o que ocurren como consecuencia de algo. Una circunstancia define cosas que ocurren cuando "algo" sucede, de modo que la circunstancia permite asociar situaciones a contextos. En Java se tienen eventos, que son objetos que se asocian a otros para ejecutar un proceso que se derive de alguna acción, la desventaja de este mecanismo es que se requiere de un objeto que se asocie a otro, lo que conlleva a una definición compleja y poco expresiva desde la perspectiva del idioma inglés. Otro mecanismo semejante a las circunstancias es el modelo de puntos de unión de AspectJ, donde se define el contexto de un punto de unión por medio de avisos, pero su desventaja radica en que los puntos de unión se basan en la sintaxis de Java, de modo que el cambio en la firma de un método implica que se revise el código del aspecto,

además de que el mecanismo sigue enfocado a una sintaxis formal y poco expresiva desde la perspectiva del idioma inglés.

Por ejemplo, una circunstancia asocia de forma indirecta la venta de una casa con la condición de que la misma no esté vendida, de modo que la expresión *sold the house to the client when the house is available* condiciona la primera oración al resultado de la segunda, de modo que la venta depende de la disponibilidad. Lo anterior se expresa en Java de la siguiente forma:

```
 1 class House { boolean available = true; }
 2 class Client {
 3   House house = null;
 4   void sold(House house) {
 5     if(house.available) {
 6       house.available = false;
 7       house.client = this;
 8       this.house = house;
 9     } else {
10       println("House not available"); }}}
```

En AspectJ se expresa de la siguiente forma:

```
 1 aspect ValidateHouse {
 2   before(House h, Client c) : execution(void sold(House)) & args(h)
 3    & target(c) {
 4     if(house.available) {
 5         proceed(h);
 6       } else {
 7         println("House not available"); }}}
 8 class Client {
 9   House house = null;
10   void sold(House house) {
11     house.available = false;
12     house.client = this;
13     this.house = house; }}
```

Por último, un ejemplo naturalístico basado en el proceso de venta de casas se expresa de la siguiente forma:

```
1 noun House:
2   attribute availability is true.
3 noun Client:
4   attribute house as House.
5   verb sold h as House to itself:
6     the house of this is h.
7   circumstance: sold house to this
8    when the availability of house is true.
```

Como se observa, la definición de Java genera código disperso y fuertemente acoplado, lo que dificulta las tareas de mantenimiento dado que la validación de la disponibilidad de la casa se realiza dentro del código de *Client* (líneas 5 y 9). La definición con AspectJ

separa la validación de la asignación, pero su sintaxis se mantiene asociada a un lenguaje de programación y además agrega una abstracción extra que se encarga de controlar el contexto (líneas 1 a 7). Por último, el ejemplo naturalístico posee una sintaxis similar al inglés y permite hacer la validación dentro del sustantivo al que afecta (líneas 7 y 8), de modo que con dicha validación se acceda al verbo donde se le asignará la casa al cliente (línea 6), aunque sin acoplarse al verbo ya que asocia los procesos, pero sin "cortarlos" de forma directa.

En estos ejemplos se presenta la forma en la cual la venta de la casa depende de su disponibilidad, lo que implica una precondición para llevar a cabo la venta de forma correcta. Es decir, la venta de la casa es un requisito funcional, mientras que su disponibilidad es un requisito no funcional que interfiere con la venta. Dicho de otra forma, la venta de la casa está en función de su disponibilidad. Este tipo de "cortes" en la funcionalidad de un método comunes (y necesarios) en el paradigma orientado a objetos, donde los requisitos no funcionales se encuentran dispersos entre la funcionalidad principal y generalmente un mismo requisito no funcional se presenta en varios métodos. El paradigma orientado a aspectos permite encapsular dichos requisitos, de modo que se facilita su control y permite un mayor nivel de modularidad en el código.

Con base en los elementos que se describieron con anterioridad, un lenguaje naturalístico se compone de sustantivos, adjetivos, verbos, circunstancias y sintagmas, de modo que dichos elementos contribuyan a que el lenguaje tenga una sintaxis más cercana al idioma inglés, con un nivel de formalidad suficientemente elevado para evitar ambigüedades en el código. Una sintaxis naturalística requiere también de mecanismos que permitan indagar sobre las líneas de código que se encuentren antes o después de la ejecución de una instrucción. Con base en lo anterior, de la investigación que se presenta en el Capítulo 3 y además, de la retroalimentación que se recibió por parte de dos expertos por medio de correo electrónico, mismos que se encuentran en el Apéndice F, los elementos mínimos que se consideran para la definición de un modelo naturalístico para el diseño de lenguajes de programación de propósito general, son los siguientes.

1. Sustantivo como abstracción base que es singular o plural.

2. Adjetivo como complemento al sustantivo.

3. Verbo como acción que realiza un sustantivo.

4. Circunstancia como condicionante que responde a eventos.

5. Sintagma para definir instrucciones con una complejidad más allá del sujeto y el predicado.

6. Definición de instrucciones en términos de elementos que se describieron con anterioridad (anáforas).

7. Definición de los tipos de forma explícita y estática, de modo que se utilicen en la construcción de instrucciones que se conformen con sintagmas.

Cabe destacar que dichos elementos sólo se consideraron para un lenguaje naturalístico que se enfoque en el idioma inglés con un nivel de expresividad que se encuentre más cercano a la forma de cómo los programadores expresan sus ideas, pero con la formalidad suficiente como para que una computadora lo procese sin ambigüedad.

Además, como elementos deseables para un lenguaje naturalístico se considera el uso de la deixis completa, es decir, que no sólo se limite a las anáforas, sino que también permita trabajar con catáforas, de modo que se provea de un soporte más robusto para el manejo de referencias indirectas. Otro elemento deseable, pero que no es indispensable son los identificadores, esto se debe a que como se mencionó, en los lenguajes naturales la gente se expresa por medio de sintagmas, reservando los nombres a elementos que deseen destacar, por lo tanto un identificador permitiría este tipo de expresiones. Por último, en ocasiones se requiere que una instancia posea propiedades adicionales, pero no se justifica la creación un adjetivo o refinar el sustantivo; en los lenguajes naturales dichas propiedades se agregan en cualquier momento, pero en un lenguaje naturalístico esto se limita a la creación de la instancia. A continuación se presentan los elementos del modelo que se consideran complementarios:

1. Soporte completo para la deixis, lo que implica que las instrucciones se definan en función de elementos que se describan no sólo antes, sino también después en el mismo texto.

2. Soporte para identificadores que permitan trabajar con las abstracciones por medio de referencias directas.

3. Tipificación basada en propiedades, lo que implica clasificar un sustantivo con base en sus propiedades y por tanto, que se tenga la capacidad para agregar nuevas propiedades a una abstracción cuando se cree.

El manejo completo de la deixis provee a un lenguaje de una mayor capacidad para trabajar con referencias indirectas estructurales, debido a que el uso de catáforas permite indagar sobre las instrucciones subsecuentes. El uso de identificadores agrega un soporte gramatical para destacar instancias particulares en un conjunto, lo que facilita su manejo y permite describir instrucciones más concisas. Por último, la tipificación basada en propiedades permite agregar nuevos elementos a un sustantivo, lo que facilita la extensibilidad del mismo al momento de la creación de una instancia, de modo que se requiere conocer las propiedades del sustantivo para una identificación más exacta. Por ejemplo, en la frase *la casa con sótano* se agregó una propiedad que antes no se mencionó y por tanto, en una lista de casas dicha propiedad sirve como medio para distinguirla del resto.

4.2.1. Sustantivo

En los lenguajes naturales, el sustantivo es la palabra con la que se identifica una "cosa" concreta, o un conjunto de "cosas". El sustantivo es la parte principal del sujeto gramatical (*Number described before*), el complemento de un verbo (*the System prints Number*) o el objeto de una preposición (*add 5 to Number*). Un sustantivo posee características (que a su vez son sustantivos) y realiza acciones (que se definen como verbos), además de que el idioma inglés posee número gramatical, lo que permite que el sustantivo se conjugue ya sea en plural (*Numbers*) o en singular (*Number*). Una abstracción naturalística que permita utilizar descripciones en función del número y el uso de tipos, ofrece un mecanismo para definir instrucciones por medio de referencias indirectas. Con base en lo anterior, el sustantivo que se describe en el modelo naturalístico toma las propiedades del sustantivo del idioma inglés[1]. Un sustantivo es una palabra con un significado particular que designa el nombre común para una cosa o concepto, ya sea tangible o intangible y que realiza acciones que se reflejan en forma de verbos, o funciona como un objeto para complementar al verbo.

[1]En las lenguas romances, el sustantivo posee además el género, que sirve para indicar qué artículo se emplea para complementarlo. No se consideró el género para el diseño de esta propuesta de modelo ya que en inglés los sustantivos se complementan con un artículo neutro.

Con base en la definición de un sustantivo, la abstracción que lo representa requiere de la capacidad para describir plurales, lo que en los lenguajes naturales se logra por medio de numerales (ya sean números u ordinales), calificadores (*all*, *some* o *none*) o artículos infinitivos. Un inconveniente consiste en que en inglés, existen sustantivos cuya escritura difiere en plural como en el caso de *child* y *children*, de modo que el modelo considera que el nombre de la abstracción representa una forma plural o no. Esto se debe a que hay palabras que representan grupos de entidades tales como *list* (que indica una lista de entidades) o *block* (una cuadra, en el contexto de un conjunto de casas).

Además de lo anterior, un sustantivo requiere de diversas propiedades que a su vez son otros sustantivos que lo conforman y le dan identidad, dichas propiedades también se emplean para crear instancias particulares del mismo o para identificarlo de entre varias instancias del sustantivo. Cabe destacar que en el contexto del sustantivo, y por tanto de la programación naturalística, la instancia es el equivalente conceptual al término lingüístico significante, ya que una instancia es la representación de una idea que se abstrajo a partir de un referente, que es un concepto real.

Por ejemplo, *Number* es un sustantivo que describe una abstracción que se emplea para representar magnitudes, medidas u órdenes. Aun y cuando hay diversos tipos de *Numbers*, el sustantivo abarca todos sin la necesidad de entrar en detalles particulares.

4.2.2. Adjetivo

Se tomó al adjetivo como una abstracción ya que se emplea para complementar al sustantivo en los lenguajes naturales. Un adjetivo permite que el sustantivo posea propiedades que no son comunes a todas las entidades, sino sólo a algunas en particular. De acuerdo a su forma gramatical, se consideran tres formas de adjetivo, que son atributivo (indican una cualidad de un sustantivo: *the integer number*), predicativo (se enlaza al sustantivo con un verbo copulativo: *the number* ***is*** *integer*) o nominal (actúan como sustantivos: *show all numbers and print the integer one*).

Por ejemplo, *Natural* es un adjetivo que se emplea para complementar al sustantivo *Number*, de modo que al hablar de un *Natural Number*, se hace referencia a un concepto particular y que deja de lado a otros como *Integer Number* o *Complex Number*.

En inglés, un adjetivo posee también formas superlativa y comparativa. El superlativo describe una característica en su grado máximo. Por ejemplo, *the greatest Number*, que

implica que el sustantivo debe conocer cuál de sus instancias coincide con el adjetivo. Por otro lado, el adjetivo comparativo permite realizar comparaciones entre dos sustantivos del mismo tipo. Por ejemplo, *number A is* ***greater*** *than number B*. Como se observa, en este caso el verbo es la palabra *is* que se complementa con el adjetivo comparativo *greater*, lo que implica que también se requiere que el susantivo valide el mayor, pero en este caso sólo de las dos instancias que se comparan.

4.2.3. Verbo

Se espera que el sustantivo responda a acciones que se representan por medio de verbos, de modo que el verbo equivale a las funciones de otros paradigmas como la programación funcional o los métodos de la programación orientada a objetos. En inglés, un verbo es el núcleo del sintagma verbal y se divide en dos tipos: finito cuando el núcleo es un verbo finito y no finito cuando el núcleo es un verboide (infinitivo, participio y gerundio).

Para el modelo que se propone en esta tesis se considera únicamente el uso de verbos finitos de modo que su implmentación en una gramática se simplifique ya que un verbo no finito como *the number* ***has been added*** *to the list* es más complejo que el finito *the number* ***added*** *to the list*, además del uso de la elipsis[2] a menos que se requiera que el sujeto de la oración aparezca en la misma, por ejemplo: si se trata de una lista de números de la cual se desea conocer el tercer elemento, la expresión *get the third number* posee una referencia indirecta a quien ejecuta la instrucción que se denomina sujeto *elíptico*.

De acuerdo a la literatura, se observaron diversos trabajos en los cuales los autores utilizan el sujeto elíptico para indicarle al sistema qué instrucciones ejecuta, por ejemplo: *get the third number from the stack* implica que el programador espera que el sistema obtenga el tercer número de la pila. Un equivalente orientado a objetos de dicha instrucción es *theStack.get(2)*, donde theStack es un identificador que indica una lista que contiene números.

Por ejemplo, un *Integer Number* provee mecanismos para expresar oraciones que ejercen algún efecto sobre el mismo u otros enteros, de modo que la expresión *add 25 to the Integer*

[2]En el inglés formal, se requiere especificar un sustantivo o pronombre antes del verbo, pero informalmente las personas hablan entre sí (o con entidades no humanas) por medio de oraciones con sujeto elíptico tales como *take it*, cuya forma correcta es ***you*** *take it*.

Number implica la descripción de una operación matemática que se expresa en lenguaje natural.

4.2.4. Circunstancia

Una circunstancia se define como el conjunto de cosas que ocurren alrededor de un hecho, de modo que en los lenguajes naturales se le denomina *circunstancia* a los elementos que afectan de algún modo al significado de la oración. Por ejemplo, la oración *the house is for sale at low price* indica un hecho, pero se desconoce el motivo por el cual la casa se oferta a bajo costo; si se extiende la oración y se agrega una circunstancia, se obtiene lo siguiente: *the house is for sale at low price because its owners need the money urgently*, donde ya se menciona por qué la casa se vende a bajo costo, lo que a su vez descarta otras casas que se vendan a bajo precio porque el motivo sea otra circunstancia diferente. Por ejemplo, *the house is for sale at low price because it is old and deteriorated* implica otra circunstancia por la cuál se vende la casa a bajo precio.

Por medio de una circunstancia se establecen contextos que describen o condicionan la ocurrencia de algo, de manera que el conjunto de circunstancias conforman el contexto de una oración. Desde el enfoque del modelo, la circunstancia provee mecanismos para describir la forma en que el sustantivo y los elementos que lo conforman interactúan con otros sustantivos, de modo que el programador expresa ideas y restricciones para instancias de forma correcta y sin que se dispersen.

Una circunstancia actúa sobre una condición para formar una estructura condicional (*the house with green walls implies a green house*), sobre un sustantivo para describir el comportamiento del mismo ante algún evento (*the house must be complete to do something with it*), sobre un verbo para establecer precondiciones (*when house is sold, mark it as not available*), sobre una propiedad del sustantivo para definir rangos para la misma (*the price must be greater than 250,000*), o sobre un grupo de sustantivos de forma que se restringen ciertas propiedades o adjetivos (*the list only accepts luxurious houses*).

Dado que se propone un enfoque en el que un sustantivo se complementa con adjetivos para crear instancias, una circunstancia permite establecer restricciones con base en qué adjetivos se permiten para la composición, qué adjetivos se requieren o qué adjetivos no se permiten. De este modo, se tiene un mayor control de la composición para mantener modularidad y tratar problemas de diseño. Por ejemplo, definir una abstracción *Number*

que sólo acepte composición con los adjetivos *Integer* y *Real*, mientras que derive en un error cuando se realice composición con otros adjetivos e inclusive se defina exclusión mutua entre ambos tipos ya que ambos adjetivos poseen los mismos verbos dado que su funcionamiento sólo varía en que uno maneja números con enteros y el otro reales, y dependiendo del paradigma o la implementación, se derive en el problema del diamante.

Como se mencionó anteriormente, una circunstancia permite que se describan restricciones no sólo para un sustantivo, sino para los elementos que lo conforman y de este modo crear políticas particulares para cada propiedad de un sustantivo. Por ejemplo, si se tiene un sustantivo *Numbers* que representa a una lista de números con una propiedad *length* para indicar la cantidad de elementos, se espera que dicha propiedad nunca sea inferior a cero. Por medio de una circunstancia se describe una condición *validate it before length is assigned*, donde *validate it* indica una referencia indirecta a un procedimiento que se representa con un verbo y que actúa sobre *length* por medio de la referencia indirecta *it*.

Por último, las circunstancias también actúan sobre los verbos, de modo que se condiciona la ejecución de un verbo respecto a otro o, al valor que tenga una propiedad en un momento determinado. Por ejemplo, si se tiene un verbo que agrega un número a una lista, tal como *add the Number to the list of Numbers* y se requiere validar que dicho número es positivo, se describe la circunstancia *verify value when add value to something*, de modo que el verbo *verify value* indica que se validará *value* cuando se ejecute el verbo *add* del sustantivo que contiene la definición tanto de la circunstancia, como de los verbos *add* y *verify*.

Con base en lo anterior, una circunstancia es una oración que permite validar la instanciación de un sustantivo, una asignación al valor de un atributo del sustantivo, o que permite establecer la ejecución de un verbo en función de otro. De modo que una circunstancia expresa un contexto para un sustantivo, adjetivo o verbo. Por ejemplo, cuando se habla de *Natural Number*, se sabe que son números positivos, de modo que el adjetivo *Natural* provee un contexto que se describe como *value must be greater than 0*[3].

[3]En este caso se excluyó el cero, pero hay áreas donde se toma en cuenta, en cuyo caso la expresión sería *a Natural Number must be greater or equal than 0*

4.2.5. Sintagmas

El modelo que se propone permite que se definan mecanismos de composición que se basen en el uso de sintagmas nominales para definir entidades. En este caso se describen dos tipos principales de sintagmas: el sintagma nominal y el sintagma verbal que juntos conforman la oración, más el sintagma preposicional como auxiliar. Por ejemplo, *the Natural Number* y *the Luxurious House* como sintagmas nominales; *divided by 25* y *sold to the Buyer* como sintagmas verbales; y por último, *the Natural Number divided by 25* y *the Luxurious House sold to the buyer* como oraciones completas.

4.2.5.1. Sintagma nominal

Se conoce como sintagma nominal a una frase cuyo núcleo es el sustantivo y se asocia al sujeto gramatical, pero también se emplea como complemento del verbo o de una preposición.

El sintagma nominal permite describir a un sustantivo con propiedades particulares que se reflejan en un adjetivo, o con valores específicos para sus propiedades particulares en forma de sintagma preposicional. Por ejemplo, la frase *the luxurious house with green walls*, donde el sustantivo es *house*, el determinante es *the*, el adjetivo es *luxurious* y el sintagma preposicional es *with green walls*. Otro ejemplo de sintagma nominal es *the Real Number with 5 as value*, donde el sintagma nominal es *the Real Number* y el sintagma preposicional es *with 5 as value*.

4.2.5.2. Sintagma preposicional

El sintagma preposicional es aquella frase cuyo núcleo es una preposición, el modelo lo describe como un mecanismo para establecer las propiedades de una entidad particular, de modo que el sintagma *with green walls* describe la propiedad *walls* (sustantivo) como *green* (adjetivo).

Si se retoma el ejemplo de *Number*, expresar el sintagma nominal *a Rational Number* con el sintagma preposicional *with 2 as numerator and 6 as denominator* permite describir la idea de un número racional que matemáticamente se representa como *2/6*.

4.2.5.3. Sintagma verbal

La frase cuyo núcleo es el verbo se conoce como sintagma verbal, el modelo lo describe en función del tipo de verbo y quién lo define. El sustantivo que lo define actuará como sujeto si es el que dirige la acción, como en *the Number increases*, en este caso aparece como sujeto gramatical; en caso contrario, si el sustantivo que lo define aparece como objeto directo o indirecto, entonces conceptualmente se da por hecho que el sujeto gramatical que la desencadena es el sistema, que se comporta como sujeto elíptico. Por ejemplo, *increase the Number*, donde la acción *increase* se condiciona al sustantivo *Number*, pero a nivel gramatical se dice que el sistema es quien incrementa el número. Con base en lo anterior, se describen dos tipos de verbo, el verbo intransitivo y el verbo transitivo.

El verbo intransitivo consiste en que el verbo no se asocia con un objeto directo, sino con un adverbio. En este caso el modelo describe a los verbos que únicamente se vinculan con el sujeto, de modo que no es elíptico. Por ejemplo la oración *the Number increases*, donde el sujeto *Number* realiza una acción *increase* que no contiene otra construcción gramatical.

El verbo transitivo consiste en que el verbo posee uno o más objetos directos sin el uso de una preposición entre ellos, por ejemplo en *the list deletes three, four and seven*. Si se agrega una preposición, entonces se dice que el verbo posee uno o más objetos directos y un sintagma preposicional, por ejemplo en *add 6, 7 and 3 to the list*, donde *to the list* funciona como sintagma preposicional; en este caso se dice que el verbo es ditransitivo. En el ejemplo de *Number*, un verbo transitivo se describe como *the Number increases by 2*, donde a diferencia de un verbo intransitivo, el contexto se complementa con el objeto directo.

4.2.6. Oración

La combinación de un verbo y un sustantivo (o sintagmas que los contengan como núcleo) da como resultado una oración con un sentido particular que depende de las definiciones tanto del verbo como del sustantivo, además de la forma en que estos últimos modifiquen su comportamiento por medio de un adjetivo. El modelo describe instrucciones gramaticalmente imperativas donde se especifican instrucciones elípticas, lo que significa que son en segunda persona siendo el sistema a quién van dirigidas las "órdenes"; e indica-

tivas para las instrucciones en tercera persona, que son las que poseen sujeto gramatical. Aunque cabe destacar que estos modos sólo son evidentes para la gente, una computadora no distinguirá modos gramaticales dada su falta de razonamiento. Un ejemplo de una instrucción imperativa es *add 5 to result*, mientras que una instrucción indicativa es *the System prints the Number*.

Un caso especial son las instrucciones interrogativas porque si bien, en teoría no hay problema en trabajar con dichas instrucciones en una implementación, no se consideraron para el diseño del lenguaje SN.

4.2.7. Referencia indirecta

El modelo integra el uso de referencias indirectas para describir elementos que se definieron con anterioridad en el código, o se definieron en otro sitio, como en el caso son exófo-ras; dichas referencias funcionan como reemplazo de los identificadores, que se consideran elementos opcionales en las implementaciones del modelo. De este modo, los sustantivos actúan como tipos que sirven para crear instancias, pero a su vez el tipo se utiliza como un mecanismo para hacer referencia a dichas instancias gracias al uso de pronombres, numerales u ordinales. De igual modo, las propiedades del sustantivo sirven como mecanismo para seleccionar abstracciones particulares, de esta forma, la instrucción *the Number with 50 as value* funciona como una referencia indirecta a una un número que tenga valor igual a 50.

De modo que si se tiene un conjunto de *Numbers*, se hace referencia a cada uno con base en diversos criterios: *the first Number*, *the 3rd Number*, *the last Number* o inclusive *it* que en hace referencia al último número que se utilizó o que se derive de alguna operación.

4.2.8. Tipificación explícita

El modelo que se presenta en esta tesis requiere de tipificación explícita, esto se debe a que en los lenguajes naturales, las personas expresan sus ideas en función de los tipos, por tanto hacen referencia a cosas como *the House*, *my House*, *the first House* o *all Houses*; en lugar de *the House H*. De modo que toda entidad que se defina, es a su vez un sustantivo cuyo tipo se definió claramente y cuyas propiedades sirven para identificarlo, o se combina con adjetivos que representan propiedades particulares que le otorgan alguna característica

que distinga a dicha entidad de otras similares. Por ejemplo, *the House with "green" as color* se define como *the Green House*, donde la propiedad *color* pasa a ser directamente el adjetivo *Green*. Por último, se espera que un tipo conforme una jerarquía que implique una especialización, de modo que un sustantivo base sirve como elemento general del cual se derivarán tipos que aportarán características más concretas. Por ejemplo, *the House* se define como una especialización de *the Building*. Como se observa, el concepto se asemeja a la jerarquía del modelo de objetos, esto se debe a que dicho modelo toma la idea de la semántica y por tanto, es parte fundamental de los lenguajes naturales.

4.2.9. Expresividad

La expresividad es la capacidad para expresar una idea en términos de un problema y gracias al razonamiento de los interlocutores, los lenguajes naturales son tan expresivos como lo desee quien los emplee, mientras que los lenguajes de programación son tan expresivos como la gramática lo permita. El modelo naturalístico que se propone se enfoca al diseño e implementación de gramáticas formales que permitan crear lenguajes de programación con un nivel de expresividad más cercano al inglés, pero a la vez con la formalidad de un lenguaje de programación tradicional. De este modo, se definen instrucciones formales, pero con un nivel de naturalidad que haga a dichas expresiones más expresivas sin presentar ambigüedad. Es decir, un lenguaje que sea suficientemente expresivo para que sea legible y comprensible si se conoce el idioma inglés, al tiempo que es formal y sin ambigüedad para que una computadora lo procese; pero que a su vez posea las abstracciones que dicta el modelo.

4.2.10. Elementos deseables

En esta sección se revisan los elementos que se consideran deseables, más no necesarios para la implementación del modelo.

4.2.10.1. Deixis completa

El uso de anáforas se considera como un elemento básico del modelo, pero la deixis como tal se deja de lado ya que las referencias a elementos que se definen más adelante en un texto son poco comunes, pero se consideraron para el modelo porque se basa en

referencias indirectas que dependen del tiempo. Por ejemplo, *previous* y *next*, de modo que la definición de elementos tales como *set rooms to 3 for the next 6 created houses* se implementan de forma transparente, aunque es algo que al momento de la redacción de esta tesis no se soporta en el lenguaje SN.

Las referencias indirectas son muy útiles cuando se trata de hacer procesos repetitivos, de modo que hablar de *repeat the next two steps until you have the result* es una forma natural de expresar una catáfora, cuyo equivalente en anáfora es *repeat the last two steps until you have the result.*

4.2.10.2. Identificadores

Los identificadores son un elemento fundamental en otros paradigmas ya que sobre ellos recae el trabajo de nombrar a las instancias para que se manipulen sin trabajar directamente con direcciones de memoria. Esta característica no se observa en los lenguajes naturales ya que generalmente se reserva para seres o cosas que una persona desea destacar del resto de las entidades de ese tipo. Por ejemplo, *From all my friends, Peter is my best friend*, donde se observa que *Peter* pertenece al conjunto de amigos, pero quien dijo esa oración lo destacó como su mejor amigo. De modo que el identificador de ese amigo es *Peter*, siendo *my best friend* una referencia indirecta y por tanto ambas son intercambiables en el contexto de quien expresó tal comentario. Si bien un lenguaje naturalístico es capaz de trabajar con referencias indirectas de forma que no requiere de identificadores, son un mecanismo de apoyo para trabajar con grandes cantidades de valores, pero cabe destacar que su uso adecuado dependerá en gran medida del proceso cognitivo que emplee cada usuario al momento de nombrar las entidades.

Por ejemplo, la expresión *A is an Integer Number with 25 as value* indica la definición de una variable *A* que es de tipo *Integer Number* y que además posee un valor de *25*, donde *value* es un atributo propio de *Number*.

4.2.10.3. Tipificación basada en propiedades

Las propiedades de un sustantivo son un elemento básico en los lenguajes naturales y en la POO, pero además, en los lenguajes naturales una propiedad también se utiliza para clasificar tipos de sustantivos. Esta forma de clasificación no es común en la POO ya que en este paradigma se emplea únicamente el tipo para realizar la clasificación, mientras que

en los lenguajes naturales se tienen expresiones como *the house with green walls*, donde la propiedad *walls* se utiliza para describir de forma indirecta a una entidad que es una casa, pero que además tiene paredes verdes. Esta propiedad se observa de forma limitada en los lenguajes de programación ya que primero se requiere conocer el tipo para saber si posee o no la propiedad, lo que deriva en una tarea redundante e innecesaria. Por otro lado, es algo muy común en las bases de datos, sobre todo cuando se requiere consultar o hacer una modificación sobre datos, de forma que se solicitan las tuplas que coincidan con el criterio de búsqueda. La tipificación basada en propiedades se reporta en [Knöll y cols., 2011] donde se describen diversos mecanismos, aunque para efectos del modelo que se propone en este artículo, se utiliza como mecanismo de extensión que no requiere de la definición de un nuevo sustantivo o de la composición con un adjetivo ya existente, ya que dicho tipo se agrega durante la creación de la instancia.

Por ejemplo, la expresión *a Number with "A" as name property*, permite agregar un nombre a un número, propiedad que no todos los números poseerán. Esto ayuda a refinar aún más los mecanismos, de modo que una descripción como *get the Numbers with name* implica únicamente a los números que posean la propiedad *name*. La tipificación basada en propiedades no se tiene integrada en el lenguaje SN al momento de la redacción de esta tesis.

4.2.11. Representación gráfica

En esta sección se presenta una representación gráfica de los elementos del modelo y su interrelación. Como se observa en la Figura 4.1, el sustantivo se combina con adjetivos para crear sintagmas nominales, aunque un sintagma nominal no requiere forzosamente de un adjetivo. Los verbos se combinan con sintagmas nominales para crear sintagmas verbales donde los sintagmas nominales trabajan como complementos ya sea directos o indirectos, aunque un sintagma verbal no requiere de un verbo. Los sintagmas nominal y verbal se combinan para crear oraciones. Por otro lado, mientras que los sustantivos, adjetivos y verbos dan coherencia gramatical al texto, las circunstancias permiten establecer un contexto. Por último, las referencias indirectas permiten facilitar la redacción al reemplazar un sintagma con base en un contexto que se definió con anterioridad, lo que acorta las instrucciones al tiempo que permite transmitir el mensaje de forma eficiente y sin ambigüedad.

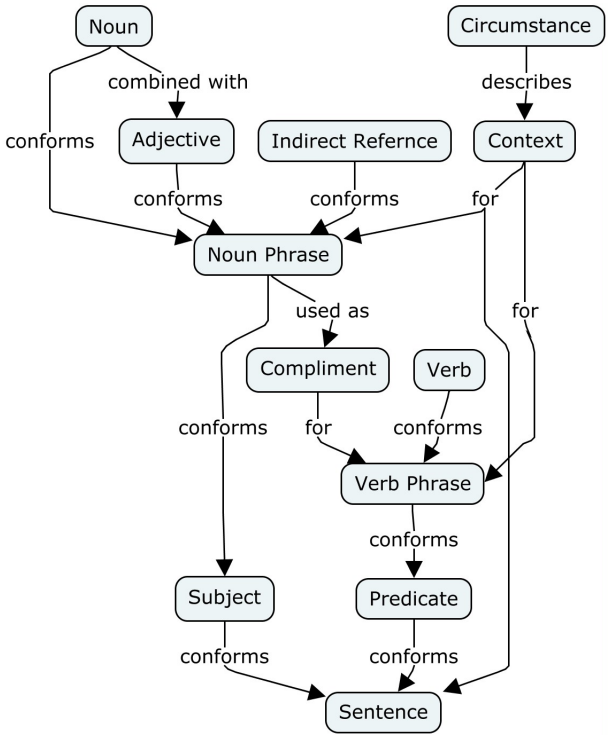

Figura 4.1: Representación gráfica del modelo

4.2.12. Validación del modelo

Un modelo conceptual es una representación de un sistema donde se emplea una composición de conceptos que sirven para entender o simular aquello que el modelo busca representar. Por tanto, es un conjunto de conceptos que representan entidades físicas o abstractas [Gregory, 1993]. Con base en lo anterior, el modelo que se propone en la tesis permite que otros desarrolladores de lenguajes identifiquen con facilidad qué elementos se requieren para diseñar nuevos lenguajes, de modo que funciona como punto de referencia y mecanismo común para la colaboración entre diversos desarrolladores [Kung y Sölvberg, 1986]. El objetivo de un modelo conceptual es presentar una imagen mental para el diseñador, lo que contrasta con los modelos formales que en el caso de la programación, se enfocan a describir los formalismos de los elementos de cada lenguaje, así como también de los sistemas que se desarrollan por medio de estos [Mylopoulos, 1992]. Por tanto, para validar un modelo de programación se requiere de un proceso de experimentación que permita garantizar la coherencia de los elementos del lenguaje respecto al modelo, esto se

debe a que un modelo de programación define características del lenguaje que no necesariamente son formales. Para dicha experimentación se diseñó el lenguaje de programación naturalística SN, mismo que se describe en las secciones restantes de este capítulo.

4.3. Lenguaje SN

El modelo conceptual que se propone en esta investigación sirve como base para definir lenguajes de programación naturalísticos de propósito general, en esta sección se presenta el lenguaje *SN*, un prototipo que se basa en el idioma inglés y que deriva de los elementos que se plantean en el modelo. El prototipo devuelve como resultado *bytecode* de Java, utiliza referencias indirectas, permite un mayor nivel de descripción al momento de definir un procedimiento al asemejarlo a un sintagma verbal (lo que conceptualmente lo asemeja a una función o método), define sustantivos que opcionalmente poseen una forma plural, adjetivos que se combinan con sustantivos ya sea durante la definición o instanciación y por último, el prototipo presenta una capacidad limitada para describir circunstancias para un sustantivo o un adjetivo. Dado que la idea principal es implementar el modelo, no se tomó en cuenta la optimización de código. Cabe destacar que dado lo extenso de la implementación, se describiran únicamente los elementos más relevantes de la gramática.

En el lenguaje SN, las palabras reservadas se escriben con minúscula, se recomienda utilizar la primera letra en mayúscula para el caso de sustantivos y adjetivos, mientras que para atributos y verbos se mantenga dicha letra en minúscula (aunque no hay problema en el uso de mayúsculas y minúsculas mientras no se utilicen palabras reservadas como sustantivos, adjetivos, verbos o identificadores). Se recomienda de este modo para crear una correcta distinción entre las abstracciones principales y el resto de las palabras.

Al basarse en la Máquina Virtual de Java (JVM), SN mantiene una estructura similar a la de las clases de dicho lenguaje, de modo que las abstracciones que hacen referencia a sustantivos y adjetivos se traducen como clases e interfaces respectivamente. Los atributos se traducen como campos de instancia con un método que permite encapsular su acceso cuando se requiere. Los verbos se traducen como métodos de instancia. Las circunstancias se traducen como cortes de AspectJ por medio del cual se implementa un mecanismo basado en eventos.

A continuación se muestra un ejemplo *Hello World*:

```
1 main Example:
2   System prints "Hello World".
```

Como se observa, en la línea 1 se define una abstracción *main* que funciona como punto de entrada al programa, mientras que en la línea 2, la oración *System prints "Hello World"* se compone del sustantivo *System*, que posee un verbo *prints* que a su vez acepta un objeto directo, que en este caso es el mensaje que se desea imprimir.

4.3.1. Sustantivo

Con base en el modelo, se emplea la palabra reservada *noun* seguida del nombre del sustantivo y por último la instrucción termina con punto. Opcionalmente se agrega la construcción "with plural" para indicar un sustantivo plural. La gramática formal para describir un sustantivo es la siguiente:

```
noun_def ::= noun_signature '.'
          |  noun_signature ':' '\n' abstraction_body


noun_signature ::= 'noun' ID
                |  'noun' ID plural_def


plural_def ::= 'with' 'plural' 'as' ID
```

Como se observa, la gramática permite definir abstracciones con una variante para trabajar con plurales, en el caso de SN se considera al plural como un conjunto no ordenado y con valores repetidos, esto es así porque se espera que esos detalles se consideren durante la creación de instancias plurales. A continuación se presenta un ejemplo:

```
noun House with plural as Houses.
```

El proceso de instanciación es simple, permite describir una instancia por medio de la palabra *a* o *an* para singulares, o *some* para plurales. A continuación se presenta un ejemplo:

```
1 a House.
2 some Houses.
```

En los lenguajes naturales, un sustantivo es una forma más especializada de otro sustantivo, lo que permite que los hablantes se expresen utilizando un nivel de "jerarquía" dado que todos los sutantivos son *cosas*. Con base en lo anterior, un sustantivo se refina para proveer un comportamiento más concreto, creando una jerarquía de tipos por medio de las palabras *is a*, como se muestra en la siguiente regla gramatical:

```
noun_signature ::= 'noun' ID
                |  'noun' ID plural_def
                |  'noun' ID 'is' a_an ID
                |  'noun' ID 'is' a_an ID plural_def
```

Con base en la regla anterior, el sustantivo *House* se define como una forma especializada de un *Building*, como se muestra a continuación:

```
noun Building.
noun House is a Building.
```

Donde, *Building* describe a un sustantivo base mientras el otro sustantivo *House* posee un comportamiento más concreto que extiende la funcionalidad de *Building*, de este modo se reutiliza código y se crea una jerarquía de tipos.

Por defecto, la raíz de los tipos del lenguaje SN es el sustantivo *Thing*, todos los sustantivos que se definan extenderán de forma implícita dicho tipo o alguno de sus subtipos.

4.3.2. Adjetivo

Para el caso particular del lenguaje SN se utiliza una forma de composición *mixin* similar a la que se observa en Scala y se emplea tanto en la definición del sustantivo, como en la instanciación del mismo, lo que permite definir adjetivos que complementen el comportamiento de los sustantivos durante la instanciación, a continuación se presenta la gramática de un adjetivo:

```
adj_def ::= adj_signature '.'
         |  adj_signature ':' '\n' abstraction_body


adj_signature ::= 'adjective' ID
               |  'adjective' ID 'is' adj_ex


adj_ex ::= ID
        |  adj_comma 'and' ID


adj_comma ::= ID
           |  adj_comma ',' ID
```

Con base en lo anterior, un adjetivo se define de la siguiente forma:

```
1 adjective Luxurious:
2   attribute price as Real Number.
```

Como se observa, el adjetivo permite describir una abstracción que posee característica que varias casas comparten (más no todas), pero que a su vez se aplica a otros sustantivos tales como automóviles o ropa. Para instanciar un adjetivo se requiere de un sustantivo al que complementa:

```
a Luxurious House with 250000.00 as price.
System prints it.
```

La notación indica que el nombre del adjetivo va primero que el nombre del sustantivo. De igual modo que en los paradigmas orientados a objetos, los adjetivos se emplean también de forma similar a un mecanismo de composición en tiempo de diseño. Con base en lo anterior, la gramática para definir un sustantivo que se combina con un adjetivo durante su definición, es la siguiente:

```
noun_signature ::= 'noun' ID
                |  'noun' ID plural_def
                |  'noun' ID 'is' a_an ID
                |  'noun' ID 'is' a_an ID plural_def
                |  'noun' ID 'is' a_an adj_ex ID
                |  'noun' ID 'is' a_an adj_ex ID plural_def
                |  'noun' ID 'is' adj_ex
                |  'noun' ID 'is' adj_ex plural_def
```

A continuación se presenta un ejemplo de un sustantivo que utiliza un adjetivo:

```
1 noun Mansion is a Luxurious House:
2   attribute price is 1000000.00.
```

Por defecto, la raíz de los adjetivos del lenguaje SN es *Adjective*, todos los adjetivos que se definan extenderán de forma implícita dicho adjetivo o alguno de sus subtipos.

4.3.3. Atributo

Un sustantivo se compone de atributos que a su vez son sustantivos. Para describir los elementos que componen a un sustantivo, se requiere utilizar dos puntos (*:*) y en la siguiente línea se comienza con tabulador. La palabra reservada *attribute* indica que se definirá un atributo donde se utiliza la palabra *as* para establecer el tipo. A continuación se presenta la gramática para trabajar con atributos:

```
attribute_def ::= '\t' attribute_signature '.'
```

```
attribute_signature ::= 'attribute' ID
                      |  'attribute' ID 'is' attr_val
                      |  'attribute' ID 'as' noun_phrase


attr_val ::= instance
           |  literal
```

A continuación se presenta un sustantivo que describe el contrato de venta de una casa para una inmobiliaria que posee dos atributos, uno de tipo *Client* y otro de tipo *House*:

```
1 noun Contract:
2   attribute buyer as Client.
3   attribute property as House.
```

La forma de acceder a los atributos de un sustantivo es la siguiente:

```
1 a Client with "John" as name.
2 a Luxurious House with 1000000.00 as price.
3 a Contract with the House as property and the Client as buyer.
4 System prints the buyer of the Contract.
5 System prints the property of the Contract.
```

Como se observa, el acceso a los atributos (líneas 4 y 5) permite crear instrucciones donde estos funcionan conceptualmente como complementos directos del verbo, mientras que el sustantivo que los contiene pasa a ser el complemento indirecto de una oración. Cabe destacar que la construcción de la línea 4 equivale en Java a *System.out.print(contract.buyer)*, donde *contract* es una instancia de la clase *Contract*, donde se definió el campo *buyer*.

4.3.4. Verbo

Dado que una oración se compone de sujeto y predicado, se requiere de un verbo que complemente al sustantivo indicando qué acciones realiza y si alteran o no su estado y el de otros sustantivos. Generalmente los programadores se expresan en función de construcciones donde se espera que quien ejecute la acción (desde una perspectiva gramatical) sea la computadora. Se propone un mecanismo que permite indicar en qué posición de la instrucción que llama al verbo se encuentra el sustantivo que contiene al verbo, en este caso por medio de la palabra *itself* se establece dicha posición. A continuación se presentan las reglas que se encargan de describir esta clase de instrucciones:

```
verb_def ::= '\t' verb_signature ':' instructions
           |  '\t' verb_signature '.' '\n'
```

```
verb_signature ::= 'verb' ID 'itself'
```

Cuya implementación en el lenguaje SN es la siguiente:

```
noun House:
  attribute availability is true.
  verb sell itself:
    available is false.
```

Como se observa, se emplea la palabra reservada *verb* que antecede al nombre del verbo y después del nombre, se define la posición que ocupa el sustantivo cuando se utiliza en una instrucción, en este caso se indica que el sustantivo actúa como *objeto directo* del verbo:

```
a House.
sell the House.
```

Donde la oración *sell the House* indica una instrucción cuyo verbo es *sell* y *the House* es quien invoca y que en la firma se define como *itself*.

Dado que la referencia al propio sustantivo se encuentra en la firma, la misma varía su posición dependiendo del tipo de oración que se requiere formar, esto se logra al cambiar de posición para crear firmas con una complejidad diferente. En este caso, se presenta una definición de verbo donde el sustantivo que lo define actúa como *objeto indirecto*, mientras que el argumento es un *objeto directo*. Con base en lo anterior, la regla *verb_ signature* se observa de la siguiente forma:

```
verb_signature ::= 'verb' ID 'itself'
                |  'verb' ID preposition 'itself'
                |  'verb' ID args preposition 'itself'


args ::= ID 'as' noun_phrase
      |  args_comma 'and' ID 'as' noun_phrase


args_comma ::= ID 'as' noun_phrase
            |  args_comma ',' ID 'as' noun_phrase
```

Cuya implementación es la siguiente:

```
1 noun House:
2   attribute owner as Client.
3   verb add buyer as Client to itself:
4     the owner of this is buyer.
```

A continuación se muestra la forma en que se invoca el sustantivo *House*:

```
a House.
a Client.
add the Client to the House.
```

Donde *add* es el verbo, *the Client* es el objeto directo y *the House* el objeto indirecto, que además es el que contiene la instrucción y por tanto, se representa en la firma con *itself*.

Por último, se provee el soporte para definir verbos donde el sustantivo que los contenga actúe como *sujeto gramatical* de forma semejante a las sintaxis de lenguajes como Java. A continuación se presenta la regla *verb_signature* integrando esta clase de construcciones:

```
verb_signature ::= 'verb' ID 'itself'
                |  'verb' ID preposition 'itself'
                |  'verb' ID args preposition 'itself'
                |  'verb' 'itself' ID args
                |  'verb' ID args
```

Cabe destacar el último caso, ya que la construcción *'verb' ID args* por defecto se considera como *'verb' 'itself' ID args*. A continuación se presenta un ejemplo de su utilización:

```
noun House:
  attribute availability is true.
  verb itself sold:
    availability is false.
```

Que se utiliza de la siguiente forma:

```
a House.
the House sold.
```

Donde, el verbo *sold* es el predicado de la oración, mientras que *the House* es el sujeto gramatical, que además contiene al verbo y en la firma se representa con *itself*.

4.3.5. Circunstancia

En el lenguaje SN, se utilizan las circunstancias para definir el contexto de ejecución de una oración, esto se logra al asociarla de forma indirecta por medio de una abstracción claramente definida que reacciona ante un evento. De este modo, una circunstancia se utiliza para establecer condiciones a las que reaccionarán los sustantivos, adjetivos y los verbos. Una característica de la circunstancia es que permite establecer qué adjetivos se requieren, cuáles no se permiten y cuáles son mutuamente excluyentes al momento de la instanciación. Por ejemplo, en el contexto de la inmobiliaria, se define la abstracción *Building* y se le asocian restricciones para el momento de trabajar con las instancias, como se muestra a continuación:

```
1 noun Building:
2   circumstance: this cannot be Integer.
3   circumstance: Office and Habitable are mutually excluded.
4   circumstance: this requires Habitable or Office.
```

Como se observa, se describieron tres circunstancias donde se indica que una instancia de *Building* no se combina con *Integer*, que forzosamente se debe combinar con *Office* o *Habitable*, pero estos a su vez no se combinan entre sí. Lo anterior permite describir mecanismos de exclusión mutua para adjetivos de forma no invasiva:

```
1 adjective Office:
2   circumstance: this cannot be Habitable.
3 adjective Habitable:
4   circumstance: this cannot be Office.
```

Cabe destacar que en el ejemplo sólo basta la definición de una circunstancia en cualquiera de los dos adjetivos, pero se colocaron ambas para efectos de demostración.

Por otro lado, una circunstancia permite condicionar la ejecución de un verbo antes o después de que se cree una instancia, de momento se restringió a que sólo se utilicen verbos que se encuentren en el mismo sustantivo que la circunstancia y el atributo, aunque se pretende proveer una funcionalidad extendida para dicho mecanismo por medio de atributos circunstanciales, lo que implica que su definición permita complementar el contexto de una circunstancia. Los atributos circunstanciales permitiran definir abstracciones cuya función principal sea establecer contextos para sustantivos y adjetivos que se definieran con anterioridad:

```
1 adjective Printable:
2   verb print itself:
3     System prints "A new house was created".
4   circumstance: print this after this is created.
```

Como se observa, se ejecutará el verbo *print* (línea 2) que provee el adjetivo *Printable* (línea 1) a la instancia con la que se combine después de creación. Por ejemplo:

```
a Printable House.
```

Imprimirá:

```
A new house was created
```

En el caso de los atributos, una circunstancia permite ejecutar un verbo cuando se intente asignar el valor para el atributo:

```
1 noun House:
2   attribute description as String.
3   verb itself prints desc as String:
4     System prints desc.
5   circumstance: it prints description when the description is assigned.
```

como se observa, el verbo *prints* se ejecuta cuando se asigna el valor del atributo *value*. Por ejemplo:

```
a House with "A beautiful house" as description.
the description of the House is "Another description".
System prints the description of the House.
```

Imprimirá:

```
A beautiful house
Another description
```

Esto se debe a que *when* (sustantivo *House*, línea 5) se consideró equivalente al aviso *before* de AspectJ.

Otro caso a considerar es cuando se requiere saber si el atributo tiene un valor particular, a continuación se presenta un ejemplo:

```
1 noun House:
2   attribute description is "".
3   verb itself prints desc as String:
4     System prints desc.
5   circumstance: it prints description when the length of description is 0.
```

Tomando el ejemplo anterior, la salida con esta circunstancia sería únicamente *Another description* debido a que se condicionó la ejecución de *prints* a que la longitud de *description* sea *0* (línea 5):

```
1 a House with "A beautiful house" as description.
2 the description of the House is "Another description".
3 System prints the description of the House.
```

Imprimirá:

```
Another description
```

Esta salida se debe a que durante la instanciación se asignó valor a *description* (línea 2), de modo que la circunstancia evalúa que la longitud no es *0*. Por el contrario, la siguiente instanciación:

```
1 a House.
2 the description of the House is "Another description".
3 System prints the description of the House.
```

Imprimirá:

```
Another description
Another description
```

Esto se debe a que no se asignó valor a *description* (línea 2), por tanto su longitud es *0*. Como se observa, la circunstancia del sustantivo *House* condiciona que se ejecute el verbo *prints* que se define en la línea 3 del mismo sustantivo cuando la longitud del valor de la cadena *description* de la línea 2 sea *0*.

Por último, dada una asignación de variable a un sustantivo *Mansion*:

```
1 noun Mansion:
2   attribute price as Integer Number.
3   verb assign p as Integer Number to itself:
4     the price of this is p.
5   derived attribute string as String:
6     a String with "Mansion valued as ".
7     add price to the String.
8     return it.
```

Se observa que no se cuenta con un mecanismo para validar si se le pasa un valor nulo o vacío al verbo *assign* en la línea 3, esto se soluciona por medio de una circunstancia y un verbo que se encargue de la validación:

```
1 adjective Validated:
2   verb itself validates val as Integer Number:
3      execute the next instruction when val is null.
4      the price of this is 0.
5      execute the next instruction when val is not null.
6      the price of this is val.
7   circumstance: this validates val instead assign val to something.
```

Como se observa, la ejecución de *assign* en la línea 3 de *Mansion* se reemplaza con la de *validates* entre las líneas 2 y 6 que se presenta en el adjetivo *Validated*, pero además la descripción condiciona a que esta circunstancia ocurra cuando se le asigne un *Integer Number* a "algo" (*something*), como se observa en la línea 7 de *Validated*. El compilador se encargará de validar que tanto la instrucción *validates*, como *assign* sean coherentes con el tipo que se requiere para el contexto particular de *Validated*. Por ejemplo: si *something* no corresponde con *Validated*, no se realizará la sustitución; lo mismo ocurre si el adjetivo no se combina con un sustantivo que posea al verbo *assign*. Por tanto, si se tienen dos instancias de *Mansion*:

```
a Mansion.
a Validated Mansion.
System prints the first Mansion.
System prints the last Mansion.
```

Cada uno imprimirá dos cosas distintas:

```
Mansion valued as null
Mansion valued as 0
```

Cabe destacar que de momento se tiene una implementación que limita esta circunstancia al uso de la referencia indirecta *something*, que toma el valor de cualquier sustantivo que ejecute un verbo que coincida con el de la firma, de modo que el uso de esta referencia crea cierto nivel de *fragilidad* (adjetivo *Validated*, línea 7) que se resuelve de forma interna, esto se debe a que la ejecución de ambos verbos que se involucran en la circunstancia se encuentran desacopladas entre sí, además de que la validación del contexto también se encuentra desacoplada.

4.3.6. Atributo derivado

En ocasiones se necesitan atributos cuyo valor depende de otros atributos o verbos, de modo que se propone la definición atributos derivados cuya implementación es similar a un verbo, pero su sintaxis e invocación es la de un atributo. Por ejemplo, si se tiene la fórmula del área de un cuadrado ($R = S*S$), esta se compone del resultado (*result*) y la longitud de los lados (*side*), la expresión naturalística para definirla es *the result of the Square*, donde *result* es un atributo cuyo valor depende de un cálculo que a su vez depende del valor de los lados. A continuación se describen las reglas que se involucran en su definición:

```
derived_attribute_def ::= '\t' derived_attribute_signature
                          ':' instructions


derived_attribute_signature ::= 'derived' 'attribute' ID
                                'as' noun_phrase
```

Esto se pensó así porque como se nota en las secciones sobre atributos y verbos, la invocación depende de la firma de un verbo, mientras que para el sustantivo se utiliza la preposition *of*. Por tanto, los atributos derivados son la combinación de la flexibilidad de un atributo con el potencial de un verbo. Esto permite definir instrucciones donde el valor de un atributo se deriva de otros, como en el caso de un sustantivo que describa el precio de una casa con un impuesto del 15 %:

```
noun House.
adjective Priced:
  attribute price as a Real Number.
  attribute tax as a Real Number.
  derived attribute total as a Real Number:
    divide tax by 100.
    add 1 to it.
    multiply it by price.
    return it.
```

Que se utiliza de la siguiente forma:

```
a Priced House with 200000.0 as price and 15 as tax.
System prints the total of it.
```

Y cuya impresión es la siguiente:

```
230000
```

La implementación es más extensa porque se desacopló el sustantivo y su implementación depende de un adjetivo, pero conceptualmente permite definir nuevos tipos de precios, como se muestra a continuación:

```
adjective Discount:
  attribute price as a Real Number.
  attribute tax as a Real Number.
  attribute discount as Real Number.
  derived attribute total as a Real Number:
    divide tax by 100.
    add 1 to it.
    multiply it by price.
    subtract discount from it.
    return it.
```

Que se utiliza de la siguiente forma:

```
a Discount House with 20000.0 as price, 5000.0 discount, and 15 as tax.
System prints the total of it.
```

Y cuya impresión es la siguiente:

```
18000
```

Comos se observa, al desacoplar la implementación se obtienen descripciones más extensas, pero que se leen de forma natural.

4.3.7. Plurales

Se espera que los plurales posean atributos y verbos, pero dado que su definición depende del sustantivo al que representan, sus elementos se definen en el mismo sustantivo anteponiendo la palabra reservada *plural*. A continuación se presentan las reglas para atributos, verbos y atributos derivados:

```
attribute_signature ::= 'plural' 'attribute' ID
                      |  'plural' 'attribute' ID 'is' attr_val
                      |  'plural' 'attribute' ID 'as' noun_phrase
```

```
verb_signature ::= 'plural' 'verb' ID 'itself'
                |  'plural' 'verb' ID preposition 'itself'
                |  'plural' 'verb' ID args preposition 'itself'
                |  'plural' 'verb' 'itself' ID args
                |  'plural' 'verb' ID args


derived_attribute_signature ::= 'plural' 'derived' 'attribute' ID
                                'as' noun_phrase
```

A continuación se presenta un ejemplo de su implementación:

```
noun House with plural as Houses:
  plural attribute sorted is false.
  plural attribute house_list as List
  plural verb add house as House to itself:
    house_list add house.
```

4.3.8. Sintagma nominal

Los sintagmas nominales son construcciones gramaticales cuyo núcleo es el sustantivo, para su implementación en el lenguaje SN se considera que los constituyentes del sintagma nominal son un sustantivo y opcionalmente uno o más adjetivos. La reglas para definir sintagmas nominales son las siguientes:

```
noun_phrase ::= ID
             |  adjective_phrase ID


adjective_phrase ::= ID
                  |  adj_comma 'and' ID


adj_comma ::= ID
           |  adj_comma ',' ID
```

Lo que se traduce a construcciones como las siguientes:

```
Luxurious House.
Luxurious and Big House.
```

Un sintagma nominal por sí mismo no es una instrucción ejecutable en el lenguaje SN ya que sirve para complementar dos procesos particulares: instanciación y selección de instancias. A continuación se presentan las reglas que permiten la instanciación de un sustantivo:

```
instance ::= singular_instance
          |  plural_instance


singular_instance ::= a_an noun_phrase


a_an ::= 'a'
       |  'an'


plural_instance :: 'some' noun_phrase
```

A continuación se presenta un ejemplo de instancia, que es el proceso de asociar la definición de un sustantivo a un espacio de memoria, de forma similar a como un objeto consiste en asociar un espacio de memoria a una abstracción que se define en una clase:

```
an Luxurious House.
a Small House.
a Building.
some Houses.
some Buildings.
```

Como se observa, Se crearon instancias naturalísticas simples, pero si se desea asignar los atributos, se requiere de un complemento preposicional, en este caso por medio de la preposición *with*. A continuación se presentan las reglas:

```
np_with ::= 'with' with_attrs


with_attrs ::= with_attr
             |  with_attrs_comma 'and' with_attr


with_attrs_comma ::= with_attr
                   |  with_attrs_comma ',' with_attr


with_attr ::= obj 'as' ID
```

Con base en lo anterior, la regla *singular_ instance* queda de la siguiente forma:

```
singular_instance ::= a_an noun_phrase
                    |  a_an noun_phrase np_with
```

Por tanto, la instanciación de los ejemplos anteriores queda de la siguiente forma, considerando que los sustantivos poseen un atributo de nombre *value*:

```
an Luxurious House with 2500000.0 as value.
a Formula with 2 as value.
a String with "Four" as value.
```

Cabe destacar que los ejemplos anteriores se almacenan en una pila de instancias, de modo que sea posible acceder a cada instancia con base en su tipo y posición, para lo cual el lenguaje SN utiliza los sintagmas nominales y la posición por medio de ordinales. A continuación se presentan las reglas que lo permiten:

```
seek_instance ::= 'the' noun_phrase
                |  'the' ordinal noun_phrase


ordinal ::= 'first'
            .
            .
            .
          |  'tenth'
          |  'last'
```

A continuación se presenta un ejemplo de su implementación:

```
the first House
the last Building
the second House
```

En ocasiones se requiere de un sustantivo con un atributo de valor particular sin importar su posición, para estos casos se utiliza la cláusula *where*, que trabaja de forma similar a como lo hace *SQL*. A continuación se presentan las reglas:

```
np_where ::= 'where' where_attrs


where_attrs ::= where_attr
              |  where_attrs_comma 'and' where_attr


where_attrs_comma ::= where_attr
                    |  where_attrs_comma ',' where_attr


where_attr ::= ID 'is' compared obj


compared ::= 'greater' 'than'
           |  'greater' 'or' 'equal' 'than'
           |  'lesser' 'than'
           |  'lesser' 'or' 'equal' 'than'
           |  'equal' 'to'
           |  'distinct' 'from'
```

Lo anterior integrado a la regla *seek_ instance*, queda de la siguiente manera:

```
seek_instance ::= 'the' noun_phrase
                | 'the' ordinal noun_phrase
                | 'the' noun_phrase np_where
                | 'the' ordinal noun_phrase np_where
```

A continuación se presenta un ejemplo de su implementación:

```
the first House where price is equal to 40000.00
the last House where price is lesser than 60000.00
```

Por último, para trabajar con conjuntos de elementos, la gramática es la siguiente, nótese que la cláusula *where* aplica también para plurales:

```
seek_instance ::= 'the' noun_phrase
                | 'the' ordinal noun_phrase
                | 'the' noun_phrase np_where
                | 'the' ordinal noun_phrase np_where
                | 'the' 'first' integer noun_phrase
                | 'the' 'last' integer noun_phrase
                | 'all' noun_phrase
                | 'the' 'first' integer noun_phrase np_where
                | 'the' 'last' integer noun_phrase np_where
                | 'all' noun_phrase np_where
```

A continuación se presenta un ejemplo de su implementación:

```
all Houses
the last 3 Houses
all Houses where price is greater than 25000
the last 3 Houses where price is equal to 500000
```

4.3.9. Oración

La instrucción básica es la oración, se conforma de un sujeto y un predicado; pero también hay oraciones donde el sujeto se infiere del contexto, en esta clase de oraciones se dice que el sujeto es *elíptico*. El lenguaje SN permite trabajar con esta clase de construcciones gramaticales con una restricción que consiste en que el invocador del verbo siempre estará al final de la instrucción justo después de una preposición, como en la oración *sell the House*, donde se infiere que el sujeto elíptico es el sistema. A continuación se presenta la gramática para la construcción de oraciones:

```
verb_call ::= ID subject
           |  ID preposition subject
           |  ID objs preposition subject
           |  subject ID objs


objs ::= obj
      |  objs_comma 'and' obj


objs_comma ::= obj
            |  objs_comma ',' obj


obj ::= 'it'
     |  'these'
     |  seek_instance
     |  literal
     |  access_attr
```

Gramaticalmente, el no terminal *subject* es equivalente al no terminal *obj*, pero se hace esta distinción para ejemplificar lo que a nivel de implementación el compilador traduce como invocador y complementos del verbo. Otro detalle a destacar son las palabras reservadas *it* y *these*, donde *it* hace referencia a la última instancia singular que se creó o que un verbo devolvió, mientras que *these* hace referencia al último plural que se utilizó o que un verbo devolvió.

4.3.10. Acceso a atributos

Para acceder a los atributos, se definen las siguientes reglas:

```
access_attr ::= nested_attr obj


nested_attr ::= 'the' ID 'of'
             |  nested_attr 'the' ID 'of'
```

A continuación se presenta un ejemplo que convierte un entero en una cadena y viceversa:

```
an Integer Number with 4 as value.
the string of the Number.
a String with "4" as value.
the integer of the String.
```

Para asignar un valor a un atributo, se utiliza la palabra reservada *is*, seguido de lo que se desea asignar al atributo. A continuación se presentan las reglas:

```
nat_assign ::= access_attr 'is' obj
```

A continuación se presenta un ejemplo de asignación del valor de un atributo:

```
a House with 40000.00 as price.
the price of the House is 25000.00.
```

4.3.11. Iterador naturalístico

En los lenguajes de programación tradicionales se acostumbra utilizar instrucciones asociadas a bloques, de esta forma se conoce el alcance de un iterador, pero en los lenguajes naturales las acciones repetitivas se describen haciendo reflexión sobre el texto que expresa las acciones a repetir. El lenguaje SN permite trabajar con instrucciones reflexivas para realizar iteraciones, de modo que un iterador funciona como referencia anafórica o catafórica, de acuerdo a lo que se indique en la propia instrucción. A continuación se presentan las reglas que componen al iterador naturalístico.

```
iterator ::= 'repeat' 'the' iterator_signature iterator_condition


iterator_signature ::= ord_one 'instruction'
                     |  ord_many numeric_literal 'instructions'


iterator_condition ::= obj 'times'
                     |  'until' condition
                     |  'for' 'each' list 'as' ID


list ::= these
       |  seek_instance
       |  access_attr


ord_one ::= 'previous'
      |  'last'
      |  'next'
```

```
ord_many ::= 'first'
           |  'previous'
           |  'next'
```

Lo anterior permite instrucciones donde se especifican únicamente qué instrucciones se repetirán y bajo qué condiciones sucederá dicha iteración, la instrucción por sí misma no tiene acceso al contexto de las instrucciones a las que afecta y ni siquiera si realmente hay una instruccón antes o después, esto debido a la naturaleza propia de la computadora, que desconoce el contexto de una instrucción.

A continuación se presenta un ejemplo:

```
i is 0.
repeat the next 2 instructions until i > 10.
System prints i.
add 1 to i.
```

Como se observa, no se definen bloques o marcas, el compilador se encarga de trabajar con las instrucciones que se le indican por medio de *the next 2*.

4.3.12. Condicional naturalístico

La bifurcación naturalística se define de forma semejante a como se expresan los iteradores naturalísticos, posee las mismas características y limitaciones, con la diferencia de que en vez de indicar que un conjunto de instrucciones se repetirán N veces, indica que el conjunto de instrucciones se ejecutará sólo si una condición particular se cumple. A continuación se presentan las reglas involucradas en la bifurcación naturalística.

```
decision ::= 'execute' 'the' decision_signature 'when' condition


decision_signature ::= ord_one 'instruction'
                     |  ord_many numeric_literal 'instructions'
```

A continuación se presenta un ejemplo:

```
flag is false.
execute the next 3 instructions when flag is equal to false.
System prints "Optional output".
System prints "Another output".
flag is true.
System prints "From the outside".
```

Donde se ejecutarán las tres instrucciones posteriores a la definición de la condición si la bandera tiene valor *false*, en caso contrario sólo se imprimirá *From the outside*.

4.3.13. Oraciones compuestas

En lenguajes como Java se acostumbra colocar instrucciones dentro de instrucciones para reducir la cantidad de líneas de código, tales como colocar la llamada a un método como argumento de otra llamada a método. El lenguaje SN no permite esa clase de construcciones debido a que la ambigüedad del código se incrementa a niveles intratables sin el uso de técnicas más complejas, algo que se encuentra fuera del alcance del modelo que se propone, aunque no se descarta que su uso aporte un mayor grado de naturalidad a un lenguaje naturalístico. Para resolver este problema, el lenguaje SN permite trabajar con varias instrucciones en una línea separadas por punto y coma (*;*) que el compilador de SN tratará como una sola instrucción. A continuación se presentan las reglas que dan vida a esta clase de instrucciones:

```
composite_instruction ::= semicolon_instructions 'and' nested_instruction


semicolon_instructions ::= nested_instruction ';'
                         |  semicolon_instructions nested_instruction ';'


nested_instruction ::= instance
                     | verb_call
                     | nat_assign
                     | nested_iterator
                     | nested_decision
```

Con base en dichas reglas, el equivalente a una instrucción en Java de tipo *A.b(C.d(E.f()))*, se expresa de esta forma en SN:

```
the E f; the C d it; and the A b it.
```

Obviando que se reemplazaron variables por referencias indirectas, el funcionamiento de este ejemplo se basa en la referencia indirecta *it*, que guarda el resultado de cualquier instrucción previa. En este caso, El resultado de la oración *the E f* se almacena en la referencia *it*; posteriormente se utiliza como objeto directo en la oración *the C d it*, el resultado de esta oración se sustituye el valor que anteriormente se asignó a *it*; por último, la oración *the A b it* de nuevo utiliza la referencia indirecta *it* como objeto directo, de esta forma se devuelve el resultado de la oración que también se almacena en la referencia indirecta *it*.

4.3.13.1. Iterador y condicional en oraciones compuestas

En los lenguajes naturales es común que al final de un párrafo se mencione que el contenido del texto se repetirá con base a una condición o por el contrario, que el texto descrito en el párrafo se condiciona por algo. Es por esto que SN tiene soporte para agregar condicionales e iteradores a las instrucciones compuestas, de modo que estas construcciones indiquen que lo que se escribió se repetirá o se ejecutará si se cubren los criterios especificados. A continuación se presenta la regla que describe la iteración dentro de una instrucción compuesta:

```
nested_iterator ::= 'repeat' iterator_condition
```

Para trabajar con condicionales dentro de una instrucción compuesta, se presenta la siguiente regla:

```
nested_decision ::= 'execute' 'when' condition
```

A continuación se presenta un ejemplo[4]:

```
a Number.
add 1 to the Number; repeat 10 times;
 and execute when the Number is equal to 0.
```

El equivalente aproximado en Java de la instrucción anterior se muestra a continuación (*add 1 to the Number; repeat 10 times; and execute when the Number is equal to 0.*):

```
int theNumber = 0;
if(theNumber == 0) {
  for(int i = 0; i < 10; i++) {
    theNumber += 1;}}
```

Como se observa, el condicional se encuentra en la posición más externa, de modo que englobará a todas las instrucciones que se encuentren antes de izquierda a derecha, al transformarse a código se analiza y se coloca al inicio del conjunto de instrucciones, lo mismo ocurre con el iterador. Por ejemplo, si se tiene la siguiente instrucción:

```
a Number.
add 1 to the Number; repeat 10 times; subtract 8 from the Number;
 and execute when the Number is equal to 0.
```

Su equivalente en Java es el siguiente:

```
int theNumber = 0;
if(theNumber == 0) {
  for(int i = 0; i < 10; i++) {
    theNumber += 1;}
  theNumber -= 8;}
```

Como se observa, el condicional incluye todas las instrucciones, pero el iterador sólo a la que le precede.

[4]El *;* indica que la siguiente instrucción continúa en la misma línea, pero por cuestiones de espacio se dividió en dos

4.3.14. Identificadores locales

En los lenguajes orientados a objetos, generalmente se tiene una sintaxis clara donde el identificador va primero, después el método y por último los argumentos, en caso de haberlos. Por otro lado, en un lenguaje natural se emplea el razonamiento para resolver las ambigüedades que deriven de descripciones donde se utilicen sustantivos y no sintagmas, esto conlleva a oraciones que un compilador naturalístico resuelva equivocadamente, de modo que oraciones como *print page* y *page read* causan una ambigüedad desde la perspectiva de un compilador que carece del proceso cognitivo para entender el significado de cada palabra. Diversos autores resuelven esto por medio de diccionarios, heurística o directamente con reducir la flexibilidad del lenguaje. Por ejemplo, un compilador que carece de esta capacidad malinterpreta la siguiente instrucción:

```
divide A by B.
```

Por razonamiento, alguien infiere la oración anterior como "toma A y divídelo entre B" y la preposición *by* se utiliza para indicar que quien ejecuta la acción es B, dicho funcionamiento dirigido por la preposición derivará en problemas de interpretación en casos como el siguiente:

```
A divided by B.
```

De nuevo, el proceso cognitivo de alguien que lee la instrucción le ayuda a entender el contexto, pero un compilador quizá lo traducirá como toma *divided* y ejecuta su verbo A con el parámetro B. Para resolver dicho problema de ambigüedad se tomó la decisión de limitar el formato del verbo, de modo que el sustantivo que lo ejecuta sea sujeto gramatical u objeto indirecto, dejando de lado al verbo ditransitivo.

Los dos ejemplos anteriores ejemplifican lo ambiguo que es trabajar con identificadores cuando se busca integrar un elemento lingüístico tal como el sujeto deíctico (*divide A by B*). Dicho problema se resolvió por medio de una validación en la tabla de símbolos, lo que da como resultado que no se utilicen como identificadores aquellas palabras que se anteriormente se definieron como verbos, ya que el compilador marcará un error semántico.

Con base en lo anterior, algunas de las reglas que se definieron en esta sección reciben una pequeña modificación para que soporten el uso de identificadores locales.

La asignación se modifica de la siguiente forma:

```
nat_assign ::= access_attr 'is' obj
            |  ID 'is' obj
```

Los argumentos del verbo (y por tanto, el sujeto) se modifican de la siguiente forma:

```
obj ::= 'it'
     |  'these'
     |  seek_instance
     |  literal
     |  access_attr
     |  ID
```

Por último, los valores que un atributo toma durante su instanciación, son los siguientes:

```
attr_val ::= instance
          |  literal
          |  ID
```

4.3.15. Gramáticas embebidas

Se observó que en la literatura científica, los conceptos de un dominio particular se describen por separado, esto se debe a la naturaleza ambigua de los lenguajes naturales que provoca que los formalismos sean suceptibles a una mala interpretación. Por ejemplo, en los lenguajes naturales los paréntesis se utilizan para hacer una aclaración dentro del texto, mientras que en matemáticas se utilizan para agrupar y priorizar operaciones. El lenguaje SN se enfoca a describir expresiones naturalísticas, de modo que se requiere de un mecanismo que permita definir formalismos de un dominio particular para funcionar de forma adecuada. Con base en lo anterior, se propone un mecanismo para describir instrucciones de un dominio particular por medio de gramáticas embebidas, para lo cual se utiliza la siguiente notación:

```
embedded_grammar ::= '(' fullName ')' '{:' any ':}'


fullName ::= ID
          |  fullName '.' ID
```

Esta notación permite cargar gramáticas embebidas que se compilen de forma separada a SN. Por ejemplo, a continuación se presenta una abstracción que realiza una operación aritmética con operadores y paréntesis:

```
noun Formula:
  attribute a is 5.
  attribute b is 2.
  attribute c is 6.
  attribute res is 0.
  verb itself function x as Number:
    (grammars.embedded.math.Math) {:
      res = a * x * (x + b) + c
    :}
    System prints res.
```

Donde *grammars.embedded.math.Math* es una gramática embebida que se provee con la API[5], para de esta forma trabajar con operaciones aritméticas.

Cabe destacar que SN provee una interfaz para extender la funcionalidad de las gramáticas embebidas, pero de momento se requiere de una gramática separada dado que los DSL no son parte de los objetivos de esta tesis. Dicha interfaz es la clase abstracta *grammars.EmbeddedGrammar*, que provee del método *analyzeCode(java.lang.String) : java.lang.String*. Aunque no se requiere que se utilice ANTLR para integrar gramáticas embebidas, de momento sólo se creó la gramática *grammars.embedded.math.Math* con dicho generador de gramáticas.

4.4. Modelado del lenguaje SN

Dada la complejidad del lenguaje y la cantidad de reglas que se obtuvieron, el modelado se mantuvo en constante cambio, en esta sección se presenta el modelado de las reglas necesarias para trabajar únicamente con oraciones naturalísticas, se tomó esta decisión debido a que el modelado de abstracciones no conlleva mayor trabajo de análisis. Se observó que dada la ambigüedad presente en la sintaxis de las oraciones al momento de integrar identificadores, el modelado en Haskell resulta con un nivel de complejidad mayor del que se esperaba al inicio de este trabajo, de modo se optó por modelar las reglas con dicha ambigüedad. A continuación se presentan las reglas que se involucran únicamente en la definición de las oraciones. En el Apéndice B se presentan las reglas que complementan la funcionalidad de estas.

```
naturalisticSentence :: Parser NatInstruction
naturalisticSentence
  =
   do
    s <- naturalisticSentence'
    return $ VerbCall s
```

[5]La descripción de la API se encuentra en el Apéndice D

```
naturalisticSentence' :: Parser NaturalisticSentence
naturalisticSentence'
 = naturalisticSentenceSP
 <|> naturalisticSentencePO
 <|> naturalisticSentenceSPOs
 <|> naturalisticSentenceSPpO
 <|> naturalisticSentencePOspO
 <|> naturalisticSentenceCGLT
 <|> naturalisticSentenceCGLEQT

naturalisticSentenceSP :: Parser NaturalisticSentence
naturalisticSentenceSP
 =
  do
   s <- subject
   v <- identifier
   return $ SubjPred s v

naturalisticSentencePO :: Parser NaturalisticSentence
naturalisticSentencePO
 =
  do
   v <- identifier
   s <- subject
   return $ PredObj v s

naturalisticSentenceSPOs :: Parser NaturalisticSentence
naturalisticSentenceSPOs
 =
  do
   s <- subject
   v <- identifier
   o <- compliments
   return $ SubjPredObjs s v o

naturalisticSentenceSPpO :: Parser NaturalisticSentence
naturalisticSentenceSPpO
 =
  do
   s <- subject
   v <- identifier
   p <- identifier
   o <- compliment
   return $ SubjPredPrepObj s v p o

naturalisticSentencePOspO :: Parser NaturalisticSentence
naturalisticSentencePOspO
 =
  do
   v <- identifier
   o <- compliments
   p <- identifier
   s <- subject
   return $ PredObjsPrepObj v o p s


naturalisticSentenceCGLT :: Parser NaturalisticSentence
naturalisticSentenceCGLT
 =
  do
   s <- subject
   reserved "is"
   glt <- greaterLesser
   o <- compliment
```

```
   return $ ComparatorsGLT s "is" glt o
naturalisticSentenceCGLEQT :: Parser NaturalisticSentence
naturalisticSentenceCGLEQT
 =
  do
   s <- subject
   reserved "is"
   gloet <- greaterLesserOrEqual
   o <- compliment
   return $ ComparatorsGLEQT s "is" gloet o
```

4.5. Compilador

En esta sección se detallarán las características más importantes del compilador del lenguaje SN. Cabe destacar que al ser el primer prototipo, se hizo especial énfasis en la funcionalidad y correctitud del resultado, dejando de lado la eficiencia y velocidad del bytecode. Por esto mismo, también se decidió utilizar dos lenguajes de programación existentes debido a que poseen características que permiten darle vida de forma sencilla a los elementos que se consideraron para el diseño del lenguaje SN.

El lenguaje principal que se utiliza como intermediario para la generación de bytecode es Scala, ya que posee composición de *mixin* durante la instanciación, una propiedad interesante que permite trabajar con abstracciones con un mayor nivel de modularidad, ya que si su descripción se deriva de adjetivos, es probable que no todos los sustantivos involucrados posean dicha característica. Por otro lado, Scala es un lenguaje funcional y esto permite trabajar con listas para realizar filtros con base en el tipo y valor de un atributo particular de una instancia. Esto permite utilizar una lista para almacenar todas las instancias sin considerar un ordenamiento distinto al momento de la instanciación.

Se utiliza AspectJ para trabajar con las circunstancias. AspectJ es un lenguaje orientado a aspectos basado en Java y que utiliza la firma de los métodos para identificar puntos de ejecución donde se inserta el código. Los cortes de AspectJ dependen de la sintaxis, de modo que se dice que dicho lenguaje es frágil, para el lenguaje SN se aprovechó esta limitante al desacoplar la ejecución de las circunstancias de los avisos, que únicamente se encargan de validar que el punto de unión y el verbo a ejecutar sean los indicados, en caso contrario el aviso no hace algo.

La gramática se implementó con ANTLR, de modo que las reglas complejas que se mostraron en la sección anterior se simplificaron al pasar de LR a LL, al momento de la redacción de esta tesis se cuenta con 68 reglas y 137 tokens, esto se debe a la complejidad que se dio a las instrucciones, donde se tienen oraciones con sintagmas nominales que se conforman de sustantivos y adjetivos, además de oraciones con sujeto deíctico y donde los complementos del verbo también son sintagmas nominales. La cantidad de tokens es elevada porque se utilizan preposiciones y auxiliares para la correcta generación de los verbos.

Otro detalle que se observó es que trabajar con elementos gramaticales de tan alto nivel conllevó a que las firmas de los métodos en el bytecode fueran ambiguas. Por ejemplo, la firma de un verbo *add num as Number to itself* en bytecode era simplemente *add*, lo que provocaba que el compilador tuviera problemas a la hora trabajar con ese verbo, porque era incapaz de decidir si la firma era *the Number add 5* o *add 5 to the Number*, de modo que para la traducción a Scala se optó por generar métodos con un nombre que incluyera no sólo el verbo, sino también la posición del sustantivo que lo invoca, el número de argumentos y la preposición, si es que los tiene. De modo que si se tiene el siguiente sustantivo:

```
noun System:
  verb itself prints arg as Thing.
  verb itself prints arg0 as Thing and arg1 as Thing.
  verb delete arg as Thing from itself.
```

La firma aproximada es la siguiente:

```
class System
  public Thing itself_prints_arg
  public Thing itself_prints_arg_arg
  public Thing delete_arg_from_itself
```

Para el lenguaje SN, las referencias indirectas existen como sustituto a las variables locales, de modo que cada verbo se traduce a un método de Scala posee un objeto que gestiona el manejo de dichas referencias. Cuando se crea una instancia, se almacena en una lista de la cual se extrae con base en la posición en la que se creó, su tipo y los adjetivos que complementan al sustantivo. Además, también se permite buscar dichas instancias por medio de los atributos que posea (sin importar que sean del sustantivo o de los adjetivos). Por ejemplo, dada una instancia de tipo *Integer Number* con un valor de *55*, la frase *the Integer Number where value is equal to 55* devolverá dicha instancia. Pero si se tienen

varias instancias del mismo tipo y el mismo valor, entonces se utiliza un ordinal para indicar qué instancia se necesita, en este caso la instrucción es *the first Integer Number where value is equal to 55*, lo que indica que se obtendrá la primera instancia.

Para trabajar con iteradores y condicionales, SN realiza un proceso de análisis semántico donde se revisa que haya las instrucciones suficientes (una o más) después de la definición del iterador o el condicional. En caso de no haberlas se presenta un error semántico y, en caso de haber instrucciones suficientes, las instrucciones se acomodan de forma que se generen bloques *while* o *for* para los iteradores; o bloques *if* para los condicionales. Dado que los iteradores y condicionales no describen bloques, el lenguaje SN sólo presenta un ámbito para las variables locales, de modo que si se define una variable en una instrucción que se relacione con un condicional o iterador; entonces su declaración se moverá justo antes de los bloques, mientras que el punto en el que se utilice se empleará para darle una asignación.

El proceso de compilación del lenguaje SN se observa en la figura 4.2. Como primer paso, se recibe código fuente que se generó en SN, el cual se procesa durante el análisis sintáctico; una vez que se analizó y se verificó que el código no posee errores, se procede a resolver las ambigüedades lingüísticas que existan en la gramática, lo que da como resultado una estructura abstracta del código, a partir de esta estructura se acomodan las instrucciones para generar los bloques *for*, *while* o *if* en caso de existir; a partir de la estructura abstracta se genera el código fuente en los lenguajes Scala y AspectJ; el código fuente se compila con sus respectivos compiladores para obtener bytecode que se entrelaza en tiempo de carga de clases (*load-time weaving*, LTW por sus siglas en inglés).

Dado que el lenguaje SN trabaja con relaciones elípticas para los condicionales e iteradores, se optó por realizar un paso intermedio entre el análisis sintáctico y la generación de código en Scala y AspectJ, este paso consiste en generar una estructura abstracta del código SN en forma de una estructura de clases de Java donde se colocan todos los elementos del programa. Esto se trabaja de este modo porque cuando se trata de referencias elípticas (en partícular con referencias catafóricas para iteradores y condicionales), el compilador desconoce qué instrucciones siguen a la referencia indirecta, de modo que es incapaz de obtener en ese momento el bloque indicado, de modo que primero se requiere de analizar todo el código SN, para posteriormente acomodarlo para que coincida con el bloque que se necesita durante la traducción.

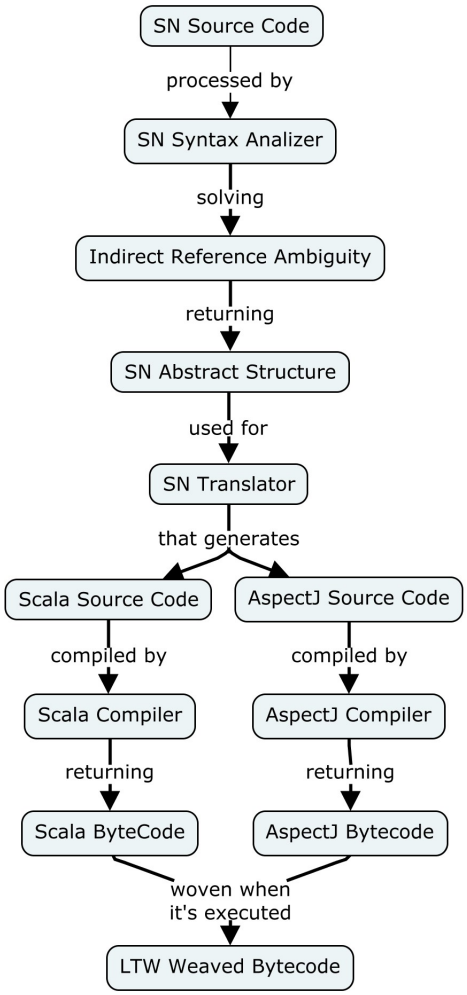

Figura 4.2: Proceso de compilación de SN

Cabe destacar que para la generación de código Scala y AspectJ se utiliza *Reflection*, esto se debe a que la traducción por medio de referencias indirectas a código ejecutable no siempre devuelve un tipo que se infiere directamente. Por ejemplo, que la referencia indirecta *it* contiene el resultado de un verbo que realice una evaluación de una consulta de SQL y por tanto, no se sabe si el verbo devolverá un entero, un real, una cadena o un plural. Para referencias indirectas se utiliza un objeto auxiliar dentro de todos los verbos que almacena una lista de todas las instancias que se crean y que permite realizar una búsqueda con base en el tipo de la instancia, el valor de alguno de sus atributos o su posición; esta búsqueda utiliza *Reflection* para filtrar los tipos (por defecto *Thing*,

posteriormente verificar si se busca desde el principio o desde el final y por último la posición del elemento que se busca. Se optó por dejar invocaciones a métodos por medio de *Reflection* que se validan en tiempo de ejecución, esto dio como resultado que la validación de las firmas en las circunstancias se hagan dentro de un aviso de AspectJ y no por medio de la firma desde el corte; que al momento de validarse resuelve la fragilidad de la firma y en el lenguaje SN se observa como un mecanismo basado en eventos. Como consecuencia, se imposibilitó el uso de cortes por medio del aprimitiva de corte *call*, de modo que su utilización se limita a la primitiva de corte *execution*.

4.5.1. Editor

En esta sección se presenta un editor simple que se diseñó para trabajar con el prototipo de lenguaje SN, este editor tiene la capacidad para detectar palabras reservadas, compilar y ejecutar archivos escritos en el lenguaje SN, como se muestra en la Figura 4.3. Este editor permite trabajar con varios archivos y además, detecta todas las abstracciones *main* que se definan en una unidad de compilación, para de esta forma permitir al programador que seleccione la que desea utilizar. Dado que el lenguaje SN genera código Scala y código AspectJ que utiliza reflection, la gran mayoría de los errores de compilación se detectarán a tiempo de ejecución, esto se debe a que el uso de referencias indirectas dificulta en gran medida verificar errores de sintaxis, lo que se consideró innecesario en esta etapa de desarrollo del lenguaje ya que el objetivo del mismo es verificar que el modelo cubra con los requisitos mínimos para implementar lenguajes naturalísticos.

4.6. Escenarios de prueba

A continuación se presentan ocho escenarios de prueba, en el primero se utilizan referencias indirectas para trabajar con las operaciones básicas de números enteros, en el segundo se presenta la implementación del algoritmo de ordenación quicksort, en el tercero se describe un equivalente aproximado del ejemplo de archivos que se presenta en [Lopes y cols., 2003], el cuarto escenario consiste en un generador de nodos de XML, el quinto escenario permite describir fórmulas matemáticas de forma naturalística, el sexto escenario presenta un analizador de expresiones, el séptimo escenario presenta una conexión natu-

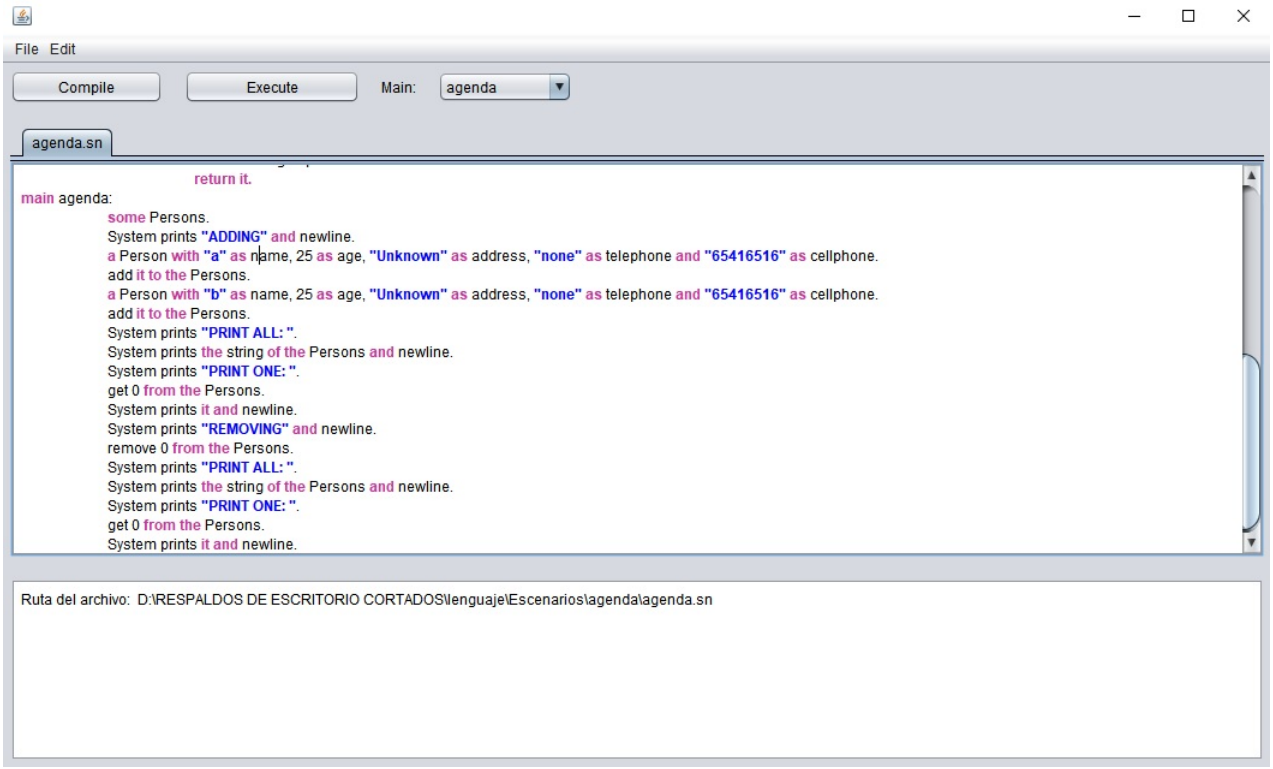

Figura 4.3: Editor básico de SN

ralística a base de datos y por último, el octavo escenario presenta una implementación del autómata celular unidimensional *Rule 110*.

Estos escenarios se eligieron porque representan de forma aislada, controlada y simple diversas características del lenguaje SN.

- El escenario 1 permite describir la forma de definir una abstracción, su jerarquía y cómo la composición con adjetivos permite trabajar con modularidad. Se eligió debido a que permite ejemplificar de forma simple la descripción naturalística de operaciones matemáticas, que contrasta con la sintaxis del dominio.

- El escenario 2 permite describir el manejo de abstracciones plurales, condicionales e iteradores naturalísticos. Se eligió debido a que se emplea el algoritmo *quicksort* para ordenar un plural de números, que es el equivalente naturalístico a las listas del paradigma funcional y los arreglos de los paradigmas procedural y orientado a objetos.

- El escenario 3 se basa en el ejemplo que se reporta en [Lopes y cols., 2003], sirve principalmente para poner en perspectiva el lenguaje SN respecto a lo que se reporta en dicho artículo.

- El escenario 4 profundiza en la descripción naturalística de la jerarquía y composición de elementos abstractos, se seleccionó XML porque es un lenguaje de marcado que permite describir abstracciones y procesos sin importar su dominio.

- El escenario 5 se presenta como un ejemplo de cómo el proceso cognitivo influye en la descripción de abstracciones, en este caso particular se observa cómo un razonamiento orientado a objetos produce una implementación funcional, pero poco naturalística.

- El escenario 6 muestra cómo las circunstancias agregan un contexto a las instrucciones. Este escenario se tomó de [Siobhan y Elisa, 2005] y permite ejemplificar de qué forma se implementa una descripción de requisitos para diseñar un analizador de expresiones, donde las circunstancias se emplean para establecer bajo qué condiciones se realizan los procesos de evaluación.

- El escenario 7 presenta un prototipo de conexión a bases de datos que se basa en el lenguaje SN y que permite desacoplar por completo la conexión de la abstracción, lo que incrementa la reutilización de código y la modularidad.

- El escenario 8 consiste en una implementación del autómata celular conocido como *Rule 110*, que es una *máquina universal de Turing*. Este escenario se utiliza para comprobar si el lenguaje SN es o no una máquina universal de Turing[6].

4.6.1. Escenario 1: operaciones con números

En este escenario se presenta el sustantivo *Number* que describe las operaciones aritméticas básicas de suma, resta, multiplicación y división. A continuación se presenta su equivalente en el lenguaje SN, cabe destacar que este sustantivo es parte de la API base del lenguaje SN, pero se utilizará una versión simplificada para demostrar las propiedades del lenguaje.

[6]También llamado "Turing completo".

```
 1 noun Number is a Thing with plural as Numbers:
 2   verb add number as Number to itself.
 3   verb itself plus number as Number.
 4   verb subtract number as Number from itself.
 5   verb itself minus number as Number.
 6   verb multiply number as Number by itself.
 7   verb itself times number as Number.
 8   verb itself divided by number as Number.
 9   verb itself mod number as Number.
10   verb leftover number as Number by itself.
11   attribute value as Number.
12 adjective Integer.
```

En la línea 1 comienza la definición del sustantivo propiamente dicho donde se establece de forma explícita que es un tipo de *Thing*, en caso de que no se especifique lo establecerá por defecto, quedando como *noun Number with plural as Numbers*. En la parte de los verbos, se definieron nueve verbos que representan las cuatro instrucciones básicas y dos más para el módulo. En la línea 2 se presenta la operación *add*, nótese que la sintaxis indica que el sustantivo se llama al final, lo que permite instrucciones con la firma *add 5 to the Number*. En la línea 3 se describe la instrucción *plus*, que también realiza una suma pero a diferencia de la anterior, esta regresa una nueva instancia y su firma es *the Number plus 5*. En las líneas 4 y 5 se definen la instrucciones *substract* y *minus*, que funcionan de forma semejante a *add* y *plus* respectivamente. En las líneas 6 y 7 se presentan las instrucciones *multiply* y *times*, que realizan una multiplicación con un comportamiento semejante a *plus* y *add*. En la línea 8 se presenta la instrucción *divided by* que realiza una división. Por último, en las líneas 9 y 10 se presentan las instrucciones *mod* y *leftover by*, que se utilizan para calcular el módulo. En la línea 11, se presenta el atributo *value*, que permite asignar un valor al sustantivo. Por último, en la línea 12 se presenta el adjetivo *Integer* que complementa al sustantivo.

Su equivalente en Scala es el siguiente.

```
 1 abstract class Number extends Thing {
 2   def add(number : Number) : Number
 3   def plus(number : Number) : Number
 4   def subtract(number : Number) : Number
 5   def minus(number : Number) : Number
 6   def multiply(number : Number) : Number
 7   def times(number : Number) : Number
 8   def divided(number : Number) : Number
 9   def mod(number : Number) : Number
10   def leftover(number : Number) : Number
11   var value : Number = null }
12 trait Integer
```

Como se observa, se tiene una abstracta clase y un *trait* con el que se realiza la composición. La parte de los plurales se maneja fácilmente con una lista, como se muestra a continuación.

```
var numbers = List(new Number with Integer)
```

En el siguiente código se presenta la forma en la que se invocan los verbos.

```
 1 main Scenario1:
 2   an Integer Number with 5 as value.
 3   add 2 to the Number.
 4   the Number plus 3.
 5   the second Number minus the first Number.
 6   an Integer Number with the value of the third Number as value.
 7   subtract 1 from the fourth Number.
 8   divide the fourth Number by 2.
 9   170 mod 9; and an Integer Number with it as value.
10   leftover it by 3.
```

La línea 1 consiste en el punto de entrada al programa; en la línea 2 se crea una instancia de Number y se le asigna el valor de 5, cabe destacar que se propone que las instancias se creen y que sus atributos se asignen según lo necesite el programador ya sea en la definición del sustantivo para darle valores predeterminados o durante la instanciación como parte de la cláusula *with*; en la línea 3 se le suman 2 a la instancia, en este caso la expresión *the Number* hace referencia a la primera (y hasta el momento, única) instancia; en la línea 4 se crea otra instancia que deriva del verbo *plus*[7] y que resulta de sumar 3 a la primera instancia, en cuyo caso la instancia tendrá valor de 10; en la línea 5 se crea una tercera instancia que se deriva del verbo *minus* y que resulta de restar el valor de la primera instancia (7) a la segunda (10), lo que significa que la tercera instancia tendrá un valor de 3; en la línea 6 se crea una cuarta instancia cuyo valor es el de la tercera instancia (3); en la línea 7 se resta 1 a la cuarta instancia (3), que termina con un valor de 2; en la línea 8 se divide la cuarta instancia (2) entre 2, por tanto su valor es de 1; en la línea 9 se crea una instancia cuyo valor será el resto de la división de 170 entre 9, resultado es 8; por último en la línea 10, se crea una última instancia cuyo valor es igual al resto de la instancia anterior (8) al dividirse entre 3, cuyo resultado es 2.

Como se observa, en este caso se prescindió del uso de identificadores y en su lugar se utilizaron referencias indirectas a instancias de acuerdo al momento en el que se instanciaron. Además, se presenta el mecanismo de instanciación y asignación que simplifica la

[7]La diferencia conceptual entre los verbos *add* y *plus* es que el primero suma a la instancia, mientras que el segundo devuelve una nueva instancia de dicha suma, sin alterar a la instancia previa.

definición de constructores, lo que permite describir sustantivos con un nivel de detalle que dependerá de las necesidades del programador.

El equivalente en Scala de dicho código es el siguiente.

```
 1 class Scenario1 extends App {
 2   val num1 = new Number with Integer
 3   num1.value = 5
 3   num1.add(2)
 4   val num2 = num1.plus(3)
 5   val num3 = num2 minus num1
 6   val num4 = new Number with Integer
 7   num4.value = num3.value
 8   num4.subtract(1)
 9   val num5 = num4.divide(2)
10   val aux = 170.mod(9)
11   val num6 = new Number with Integer
12   num6.value = aux
13   num6.leftover(3)}
```

Como se observa, se utilizaron identificadores para trabajar las instancias de forma correcta (líneas 2, 4, 5, 6, 9, 10 y 11), en este caso no se utilizaron constructores con argumentos y la asignación del atributo *value* se hizo en una instrucción aparte (líneas 3, 7 y 12). Además, nótese que la sintaxis resulta en una mayor cantidad de código respecto al uso directo de funciones matemáticas, algo que se implifica por medio del uso de gramáticas embebidas que permiten el uso de paréntesis para agrupar instrucciones, en lugar de utilizar varias oraciones para resolver una fórmula compuesta, como se observa en el Escenario 5.

4.6.2. Escenario 2: ordenamiento de una lista de números

El escenario consiste en un sustantivo *Sorter* que recibe un grupo de números y los ordena de forma ascendente por medio del verbo *sorts*. En este caso de estudio se revisarán iteradores e instrucciones de decisión. Para ejemplificar este escenario, cabe destacar que el sustantivo *Number* y su plural poseen los siguientes elementos:

```
1 noun Number with plural as Numbers:
2   plural verb add num as Number to itself.
3   plural verb add nums as Numbers to this.
4   plural verb itself sorted.
5   plural attribute size is 0.
6   plural derived attribute pivot as Number.
7   plural derived attribute head as Number.
8   plural verb remove itself.
```

Cabe destacar que todos los verbos y atributos que se definieron se agregarán al plural y no al singular, de modo que en la línea 2 se define un verbo que agrega un *Number* al

plural, en la línea 3 se define un verbo que agrega un conjunto de tipo *Numbers* al plural, en la línea 4 se define un verbo que devuelve verdadero si el conjunto está ordenado, en la línea 5 se define un atributo que almacena el número de elementos en el conjunto, en las líneas 6 y 7 se definen atributos derivados que obtienen el pivote y el elemento actual del conjunto y en la línea 8 se define un verbo para remover un elemento del conjunto.

A continuación se muestra la utilización del código que se definió en el bloque anterior.

```
 1 noun Sorter:
 2   verb itself sorts numbers as Numbers and pivot as Integer Number:
 3     some Numbers.
 4     repeat the next two instructions until the Numbers sorted.
 5     this leftSide numbers and pivot; and add these to the first Numbers.
 6     this rightSide numbers and pivot; and add these to the first Numbers.
 7     return the first Numbers.
 8   verb itself leftSide numbers as Numbers and pivot as Integer Number:
 9     execute the next instruction when the size of numbers is equal to 1.
10     return numbers.
11     N is divide the size of numbers by two.
12     some Numbers.
13     repeat the next three instructions N times.
14     execute the next instruction when the head of numbers is lesser or equal
        to pivot.
15     add the head of numbers to the first Numbers.
16     remove numbers.
17     itself sorts these and the pivot of the Numbers; and return these.
18   verb itself rightSide numbers as Numbers and pivot as Number:
19     execute the next instruction when the size of numbers is equal to 1.
20     return numbers.
21     N is divide the size of numbers by two.
22     subtract the size of numbers from N.
23     some Numbers.
24     repeat the next three instructions N times.
25     execute the next instruction when the head of numbers is greater
        than pivot.
26     add the head of numbers to the first Numbers.
27     remove numbers.
28     this sorts these and the pivot of the first Numbers; and return these.
```

Como se observa en la línea 2, el verbo *sorts* recibe el tipo *Numbers*, que es el plural de *Number*. En la línea 3 se crea una instancia de *Numbers*, nótese que la sintaxis entre un singular y un plural cambia para que sea gramaticalmente correcto. En la línea 4 se define uns instrucción *repeat* que actúa como una referencia indirecta estructural para repetir las siguientes dos instrucciones, al repetir instrucciones que se definen más adelante, se tiene el uso de *catáforas*, dichas instrucciones se repetirán hasta que la instancia que se creó esté ordenada; cabe destacar que los argumentos no forman parte de la pila de instancias que se crean. En las líneas 5 y 6 llama a los verbos *leftSide* y *rightSide*, que reciben como parámetros los argumentos *numbers* y *pivot* y agregan el resultado a la instancia que se creó. Por último, la línea 7 devuelve la instancia con los números ordenados.

Los verbos *leftSide* y *rightSide* funcionan de forma semejante, de modo que sólo se describirá *leftSide* y para *rightSide* se mencionarán las diferencias. En la línea 9 se observa una instrucción *when* que al igual que *repeat*, actúa como catáfora para las siguientes instrucciones, sólo que en este caso indica instrucciones que ejecutará sólo si la condición *when* es verdadera y qué sólo consiste en devolver el conjunto *numbers*, como se observa en la línea 10. En la línea 11 se crea una nueva instancia que equivale a la mitad de la longitud del argumento *numbers*. En la línea 12 se crea una nueva instancia de *Numbers*. En la línea 13 se presenta otro iterador que repetirá las siguientes tres instrucciones el número de veces que se asoció al identificador N cantidad de veces, que equivale a la mitad de la longitud de *numbers*. La línea 14 indica una condición que ejecutará la siguiente instrucción sólo si el número actual de *numbers* es menor o igual al pivote. La línea 15 indica que se agregará a la variable local de tipo *Numbers* el resultado de remover el primer elemento de *numbers*, lo que se indica con la frase *head of numbers*. La línea 16 indica una frase para remover el primer elemento de *numbers*. Por último, la línea 17 hace una llamada a *sorts* con el conjunto de números que se llenó y su pivote y devuelve el resultado.

Para el verbo *rightSide*, sólo cambia la línea 25, donde sólo moverá los números que sean mayores al pivote.

Para ejecutarlo, se requiere de una instancia de *Sorter*, una instancia de *Numbers* y los valores para la misma:

```
 1 main Example:
 2   a Sorter.
 3   some Numbers.
 4   add 10 to Numbers.
 5   add 5 to these.
 6   add 25 to these.
 7   add 40 to these.
 8   add 1 to these.
 9   add 0 to these.
10   add 36 to these.
11   add 50 to these.
12   add 100 to these.
13   add 80 to these.
14   add 12 to these.
15   add 99 to these.
16   the Sorter sorts the Numbers and the pivot of the Numbers;
      and System prints these.
```

En este caso, en la línea 2 se crea la instancia de *Sorter*; en la línea 3 se crea la instancia de *Numbers*; de la línea 4 a la línea 15 se llena *Numbers*; por último, en la línea 16 se invoca al verbo *sorts* que recibe como parámetros el conjunto de números y el pivote para después imprimir el conjunto ordenado.

Cabe destacar que los plurales no se pensaron para que se encuentren ordenados, además de que aceptan valores repetidos, esto es porque el tratamiento para esos casos se manejará a nivel de API, de momento no se cuenta con una implementación que permita crear plurales ordenados y que no acepten valores repetidos.

El equivalente en Scala no requiere de una abstracción plural dado que el concepto se aborda con listas de datos, de modo que el ejemplo resulta de la siguiente forma.

```
class Scenario2 extends App {
  val sorter = new Sorter
  var numbers : List[Number] = List()
  numbers = numbers :: List(10)
  numbers = numbers :: List(5)
  numbers = numbers :: List(25)
  numbers = numbers :: List(40)
  numbers = numbers :: List(1)
  numbers = numbers :: List(0)
  numbers = numbers :: List(36)
  numbers = numbers :: List(50)
  numbers = numbers :: List(100)
  numbers = numbers :: List(80)
  numbers = numbers :: List(12)
  numbers = numbers :: List(99)
  sorter.sorts(numbers, (numbers.size/2))}
```

Como se observa, el uso de la lista de Scala provoca que los procesos de validación del conjunto se hagan dentro de *Sorter*, lo que conlleva a tener validaciones dispersas dentro de la clase que se encarga de la ordenación.

4.6.3. Escenario 3: manejo de flujos de datos

Este escenario se basa en el ejemplo que se propone en [Lopes y cols., 2003], dado que es una representación de código escrito en Java, sólo se describe una representación aproximada la cual no se garantiza que funcione sin realizar un análisis profundo de la API para traducirla al lenguaje SN y de este modo presentar un ejemplo funcional. Primero se describen tres sustantivos que poseen verbos que se encargan de leer, escribir o limpiar los datos, posteriormente se presenta el ejemplo principal.

```
1 noun Encoder:
2   verb encodeDuration bytes as Byte Numbers in itself.
```

```
1 noun InputStream:
2   verb read bytes as Byte Numbers from itself.
3   verb empty itself.
```

```
1 noun OutputStream:
2   verb write bytes as Byte Numbers into itself.
3   verb empty itself.
```

```
 1 noun Example:
 2   verb itself encodeStream input as InputStream and
      output as OutputStream:
 3     an Encoder.
 4     some Byte Numbers.
 5     repeat the next 3 instructions until empty input.
 6     some Byte Numbers; read the Byte Numbers from input;
        and add it to the Byte Numbers.
 7     encodeDuration these in the Encoder.
 8     write it into output.
 9     execute the next instruction when read the Byte Numbers from
         input are lesser than the first Byte Numbers.
10     this patch the last Byte Numbers and 0.
11   verb itself patch bytes as Bytes and number as Byte Number.
```

La siguiente descripción se enfoca únicamente al sustantivo *Example*. Como se observa, en la línea 3 se crea una instancia de *Encoder*; en la línea 4 una instancia de *Numbers* que además se delimita por medio del adjetivo *Byte*; en la línea 5 se describe un iterador que repetirá las siguientes tres instrucciones hasta que *input* esté vacío; en la línea 6 se crea otra instancia de *Byte Numbers* que recibe el valor del resultado del verbo *read* de *input*, que recibe como parámetro la primera instancia de *Byte Numbers*; en la línea 7 se ejecute *encodeDuration* que recibe la instancia de *Byte Numbers* que se creó en la línea anterior y que se invoca por medio de la referencia indirecta *these*; en la línea 8 se escribe el resultado de la línea 7 en *output*; la línea 9 describe un condicional que indica que se ejecutará la siguiente instrucción si los bytes que se leyeron de *input* son menores que el valor de la primera insta de *Byte Numbers*; la línea 10 llama al verbo *patch* que recibe la última instancia de *Byte Numbers* que se leyó y la línea 11 es la definición sin implementar de dicho verbo.

4.6.4. Escenario 4: generación de nodos de XML

Este escenario describe la generación de nodos de XML por medio de una descripción naturalística. A continuación se presenta la abstracción *Attribute*, que se encarga de generar cadenas de texto que dan formato a los atributos de un nodo de XML:

```
1 noun Attribute with plural as Attributes:
2   attribute name as a String.
```

```
 3   attribute val as a String.
 4   overrided derived attribute string as a String:
 5     value is "".
 6     add name to it; add "=\"" to it; add val to it; add "\"" to it;
        and return it.
 7   overrided plural derived attribute string as a String:
 8     value is "".
 9     repeat the next 3 instructions for each element of this as attr.
10     execute the next instruction when value is distinct to "".
11     add ", " to it.
12     add string of attr to it.
13     return it.
```

Un atributo de XML consiste de un nombre y el valor que se le asocia, en este caso representados por los atributos *name* y *value* respectivamente en las líneas 2 y 3. Además, la abstracción posee también el atributo derivado *string* que devuelve una cadena con formato *name = "value"*. Además, como se observa en la línea 1, la abstracción posee la forma plural*Attributes*, que posee un atributo derivado también llamado *string* y que devuelve el conjunto de atributos que conforman un nodo, con el formato *name_ 1 = "value1_ ", name_ 2 = "value_ 2", ... name_ n = "value_ n"*.

La abstracción *Attribute* se utiliza como parte de la abstracción *Node*, que es la que define los nodos de XML. A continuación se presenta su descripción:

```
 1 noun Node with plural as Nodes:
 2   attribute nodes as some Nodes.
 3   attribute attributes as some Attributes.
 4   attribute name as a String.
 5   overrided derived attribute string as a String:
 6     value is "<".
 7     add name to value.
 8     execute the next instruction when size of attributes is greater
        than 0.
 9     add " " to value; and add string of attributes to value.
10     execute the next 2 instructions when size of nodes is greater than 0.
11     add ">\n" to value; and add string of nodes to value.
12     add "<" to value; add name to value; and add "\\>\n" to value.
13     execute the next instruction when size of nodes is equal to 0.
14     add "\\>\n" to value.
15     execute the next instruction when length of value is equal to 0.
16     value is "<"; add name to it; and add "\\>" to it.
17    return value.
```

Como se observa, esta abstracción sólo posee tres atributos simples en las líneas 2, 3 y 4, y un atributo derivado a partir de la línea 5, donde el atributo *nodes* se utiliza para definir los nodos que se encuentren dentro del nodo en cuestión, el atributo *attributes* describe los atributos que posee el nodo, y el atributo *name* indica el nombre del nodo. El atributo derivado *string* se encarga de dar el formato para los nodos, donde en la línea 10 se condiciona la ejecución de las siguientes dos instrucciones a que el nodo contenga dos

nodos anidados, mientras que la línea 13 condiciona la ejecución de la siguiente instrucción a que el nodo no posea nodos anidados. Ambos condicionales permiten distinguir entre nodos con nodos anidados y nodos sin nodos anidados.

Por último, se presenta un ejemplo de su utilización:

```
1 main Main:
2   attrs are some Attributes.
2   an Attribute with "name" as name and "001" as val.
3   add it to attrs.
4   an Attribute with "age" as name and "25" as val.
5   add it to attrs.
6   System prints string of attrs and newline.
7   node is a Node with attrs as attributes and "Node0" as name.
8   System prints string of node.
```

Donde, entre las líneas 2 y 5 se crean atributos y se agregan a un plural *attrs*, que se definió en la lína 2 se imprime en la línea 6. En la línea 7 se crea un nodo *node* que contiene a *attrs*, que posteriormente se imprime en la línea 8. El resultado de la ejecución de este código es el siguiente:

```
name="001", age="25"
<Node0 name="001", age="25"/>
```

4.6.5. Escenario 5: descripción naturalística de fórmulas

En este escenario se presenta el concepto de fórmula en la abstracción *Formula* que se combina con los *Percentage* y *Speed*, que a su vez son mutuamente excluyentes:

```
noun Formula:
  circumstance: this must be Percentaje or Speed.
  circumstance: Percentage and Speed are mutually excluded.
```

A continuación se presenta la fórmula de porcentaje, que se representa por el adjetivo *Percentage*:

```
adjective Percentage:
  attribute quantity as a Real Number.
  attribute percent as an Integer Number.
  derived attribute result as a Real Number:
    aux is the real of percent; aux / 100; quantity * it; and return it.
```

A continuación se presenta la fórmula de velocidad, que se representa por el adjetivo *Speed*:

```
adjective Speed:
  attribute distance as a Real Number.
  attribute time as an Real Number.
  derived attribute result as a Real Number:
    distance / time; and return it.
```

Como se observa, ambos adjetivos poseen atributos que indican las partes de cada fórmula, pero coinciden únicamente en el atributo *result*, que es el que provee el resultado. Para su utilización, se presenta el siguiente ejemplo:

```
1 main Main:
2   an Speed Formula with 10 as distance and 5 as time.
3   System prints the result of it and newline.
4   spf is an Speed Formula with 10 as distance and 5 as time.
5   System prints the result of spf and newline.
6   System prints the result of an Percentage Formula
     with 10 as percent and 500 as quantity and newline.
7   a Percentage Formula with 10 as percent and 500 as quantity and newline.
8   System prints the result of it.
```

Donde en la línea 2 se creó una instancia de *Speed Formula* con los valores de distancia y tiempo, cuyo resultado se imprime en la línea 3. En la línea 4 se creó una instancia de la misma fórmula, sólo que esta se asoció a la variable local *spf*, cuyo resultado se imprime en la línea 5. En la línea 6 se imprime directamente el resultado del atributo *result* de la instancia de *Percentage Formula*, que a su vez recibe los valores de cantidad y porcentaje. Por último, en la línea 7 se crea otra instancia de *Percentage Formula*, cuyo resultado se imprime en la línea 8.

Cabe destacar que este ejemplo desarrolló una alumna que realizó las primeras pruebas para el lenguaje SN, dicha alumna conoce objetos y por tanto su razonamiento era orientado a objetos. Como resultado, su ejercicio que funcionaba, pero no era naturalísticamente expresivo. A continuación se presenta dicho ejemplo, donde se observa que el proceso cognitivo que se emplea para programar con objetos difiere del que se emplea para un lenguaje naturalístico:

```
noun SpeedFormula:
  attribute speed is 0.
  verb itself calculateSpeed distance as Number and time as Number:
    (grammars.embedded.math.Math) {:
      speed = distance / time
    :}
    System prints speed.
```

Como se observa, el ejemplo funciona correctamente, pero el nombre de la abstracción es una composición de un adjetivo con un sustantivo. Esto no tiene mayor problema, pero

con base en el proceso cognitivo, se trata de una descripción no naturalística. Lo mismo ocurre con el verbo, que es la unión de un verbo con un adjetivo. A continuación se presenta un ejemplo de su utilización:

```
main CalcularV:
  System prints "Write the distance: " and newline.
  System reads.
  dst is integer of it.
  System prints "Write the time: " and newline.
  System reads.
  time is integer of it.
  System prints dst and "/".
  System prints time and newline.
  fv is a SpeedFormula.
  fv calculateSpeed dst and time.
```

Nótese que el estilo es muy similar al que se emplea en lenguajes orientados a objetos.

4.6.6. Escenario 6: analizador de expresiones

El siguiente escenario consiste en un analizador de cadenas que se encarga de devolver abstracciones a partir de las mismas, o un error en caso contrario. Por ejemplo, si se introduce la cadena "2+4", devolverá una abstracción *Binary Expression* que a su vez contiene dos abstracciones de tipo *Numeric Expression*. A continuación se presenta el sustantivo *Expression*:

```
abstract noun Expression with plural as Expressions:
  circumstance: Numeric, Variable, Binary and Unary are mutually excluded.
  circumstance: this requires Numeric, Variable, Binary or Unary.
  abstract verb itself calculates.
  abstract verb print itself.
```

Dada la complejidad del ejemplo, los adjetivos *Unary*, *Binary*, *Numeric* y *Variable* se colocaron en el Apéndice C, lo mismo sucede con el sustantivo *Parser*, que se encarga del análisis de código.

El proceso de *Parser* es el siguiente: se recibe una cadena, se separan los caracteres buscando patrones, en caso de no encontrarlos devuelve un mensaje de error. Si se encuentran patrones, el programa verifica que tipo de expresión se encuentra y crea las instancias correspondientes.

El código se utiliza de la siguiente forma:

```
main TParser:
  a Parser.
  System prints "Write the expression: ".
```

```
  System reads.
  the Parser parses the String.
  System prints it.
```

Donde se pide una expresión y se valida por medio de *parses*, para posteriormente imprimirla.

4.6.7. Escenario 7: conexión a base de datos

En este escenario se presenta un ejemplo de conexión a base de datos por medio de consultas naturalísticas. SQL es un lenguaje de consulta con un alto nivel de expresividad, esta característica facilita que se implemente en el lenguaje SN sin mayores problemas. A continuación se presentan dos abstracciones muy simples: *Car* y *Person*:

```
noun Car:
  attribute model as a String.
  attribute maxSpeed as a Integer Number.



noun Person:
  attribute name as a String.
  attribute age as a Integer Number.
```

Con base en estas abstracciones, en el siguiente ejemplo se presenta un mecanismo que permite convertir a estas tres abstracciones en entidades de base de datos con la capacidad para realizar consultas, insertar, eliminar y modificar información:

```
 1 import naturalistic.sql.DBEntity as Entity.
 2 import naturalistic.sql.DBPersistent as Persistent.
 3 main example:
 4   a Persistent String with "BMW" as value.
 5   System prints it and newline.
 6   an Entity Car with the first String as model.
 7   the driver of the Car is "org.postgresql.Driver".
 8   the jar of the Car is "C:\\SN_examples\\lenguaje\\Escenarios
      \\database\\postgresql-42.2.1.jar".
 9   the user of the Car is "admin".
10   the password of the Car is "admin1".
11   the URL of the Car is "jdbc:postgresql://localhost:5432/Agenda".
12   some Strings.
13   add "model" to these.
14   select the Strings from the Car.
15   System prints these and newline.
```

Considerando que exista una tabla *Car* con atributos *model* y *maxSpeed*, en la línea 1 y 2 se cargan dos adjetivos que se utilizan para proveer trabajar con base de datos. Donde

naturalistic.sql.DBEntity convierte a las instancias de cualquier abstracción en entidades de base de datos, mientras que *naturalistic.sql.DBPersistent* convierte a los atributos de dichas instancias en elementos que se relacionan con las tuplas de dichas entidades. En la línea 4 se crea un *Persistent String* y se le asigna un valor, en la línea 6 se asigna dicho valor a una instanica de *Car* que se combina con el adjetivo *Entity*. De las líneas 7 a 11 se carga información relacionada con el gestor que se pretende utilizar, en este caso el *driver* de PostgreSQL (línea 7), la ruta donde se encuentra el jar para la conexión (línea 8), el usuario de la base de datos (línea 9), la contraseña de la base de datos (línea 10) y por último, la URL de conexión (línea 11). En la línea 12 se crea un plural de cadenas al cual se le agregarán los nombres de las columnas de la tabla *Car* en la base de datos, en la línea 13 se indica que se desea buscar *model*. En la línea 14 se ejecuta la consulta, nótese que la instrucción es una oración muy similar a una consulta de SQL, pero también es una instrucción válida en SN. En la línea 15 se imprime el valor.

En el Apéndice C se muestran los procesos de relacionados con la inserción, eliminación, consulta y actualización. Cabe destacar que si bien se espera que este mecanismo sea independiente del gestor de bases de datos, todavía se requiere de pruebas para corroborarlo.

4.6.8. Escenario 8: rule 110

Rule 110 es un autómata celular de una dimensión, se eligió este autómata porque posee la particularidad de que es una implmentación de una *máquina universal de Turing* [Cook, 2004], lo que significa que en teoría, permite resolver cualquier cálculo que se ingrese con la única restricción de que el concepto original propuesto por Alan Turing implica memoria infinita [Turing, 1937].

El autómata consiste en ocho reglas que aplican sobre una línea de valores "encendidos" (1) y "apagados" (0), posee ocho reglas que indican el estado de una celda de la siguiente línea a partir del estado de tres celdas superiores (la que está en la misma posición y sus adyacentes) [Wolfram, 2002]. Las reglas se presentan a continuación:

```
| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|  0  |  1  |  1  |  0  |  1  |  1  |  1  |  0  |
```

Lo que resulta en patrones como el que se describe a continuación dando como entrada el valor *1* y un límite de diez ciclos:

```
00000000000001
00000000000011
00000000000111
00000000001101
00000000011111
00000000110001
00000001110011
00000011010111
00000111111101
00001100000111
```

Si se reemplazan los valores por "X" para "encendido" y espacio en blanco para "apagado", el patrón se observa de la siguiente manera:

```
          X
         XX
        XXX
       XX X
      XXXXX
     XX   X
    XXX  XX
   XX X XXX
  XXXXXXX X
XX     XXX
```

Se eligió este escenario porque de este modo se demuestra que SN es una máquina de Turing completa, esto se debe a que para demostrar la completitud de la máquina, se requiere de implementar otra máquina de Turing completa en el lenguaje. Además, el lenguaje se diseñó con un generador de gramáticas y es modelable con BNF, que por definición son máquinas de Turing completas.

Dada la complejidad del programa, se optó por únicamente describir las reglas y un ejemplo de su salida, mientras que en el Apéndice C se presentan dos ejemplos de implementación, uno con Java y otro con SN.

4.7. Evaluación del modelo

Como se observa en los escenarios que se presentan en la sección anterior, la gramática de SN permite trabajar con referencias indirectas, además de reducir el uso de identificadores como mencionan los autores en [Lopes y cols., 2003], al tiempo que se mantiene una gramática similar al idioma inglés.

Para evaluar la legibilidad lingüística del código se utilizaron métricas enfocadas a evaluar el idioma inglés, ya que las métricas enfocadas al código fuente toman en cuenta elementos como el uso de caracteres especiales, comentarios y que el código sea lo más corto posible, SN carece de caracteres especiales tales como llaves, paréntesis u operadores

(que se definen como nombres de verbos, más no forman parte de la gramática), no permite el uso de comentarios ya que se espera que el código sea autodocumentado y como consecuencia, se espera un código más extenso. Se empleó la facilidad de lectura de Flesch (*Flesch Reading Ease score*) [Flesch, 1948], que deriva en un puntaje entre el 1 y el 100, donde a mayor puntaje, mayor facilidad de lectura. Por ejemplo, un texto puntuado con *40.0*, es un texto cuya dificultad se enfoca a estudiantes universitarios, mientras que un texto puntuado con *88.0* es un texto que refleja un inglés informal y fácil de leer. Otra métrica que se utilizó es el nivel de grado Flesch-Kincaid (*Flesch-Kincaid Grade Level*) [Kincaid, Fishburne Jr, Rogers, y Chissom, 1975], que consiste en una escala del 0 al 18, donde valores entre 0 y 6 se considera para gente que aprende inglés, un valor de *8* es el nivel de complejidad esperado para que el 80 % de los estadounidenses entiendan un texto, y un valor entre 15 y 18 consiste en artículos científicos. Por último, se utilizó el índice niebla de Gunning (*Gunning Fog index*) [Gunning, 1969], que consiste en una tabla de valores entre el 0 y el 20, donde un texto de nivel *6* es un texto fácilmente entendible por personas con un nivel educativo equivalente al sexto grado en Estados Unidos, un texto de nivel *17* se enfoca a graduados universitarios y la media para los estadounidenses se encuentra en *8*.

Cabe destacar que evaluar la expresividad desde la perspectiva del lenguaje natural es algo prácticamente imposible ya que como se menciona en [Cozzie y cols., 2011], no se reporta un mecanismo estandarizado para evaluar la traducción entre *man pages* (documentación de software para sistemas basados en *UNIX*), es decir: no se reporta un mecanismo que permita evaluar de forma objetiva un programa y lo que se espera que realice (sus intenciones descritas en lenguaje natural). Además, los mecanismos que se reportan en otros trabajos revisados se enfocan al código que resultó de la traducción (si es el caso) o en su defecto, a pruebas experimentales por parte de personas con diverso nivel de conocimiento en programación (desde niños, hasta arquitectos de software y programadores). Por este motivo se seleccionaron instrucciones y fragmentos de código que abarcaran la mayor cantidad de construcciones válidas de SN, para que posteriormente alumnos con diversos niveles de experiencia tanto en programación como en inglés evaluaran su expresividad desde la perspectiva del lenguaje natural. Se descartó el uso de métricas tradicionales como la de la complejidad de Halstead [Halstead, 1977] debido a que sólo toman en cuenta el número de instrucciones, pero dejan de lado la legibilidad

lingüística del código, algo que es central en la programación naturalística y que además, dada la complejidad de la gramática de SN, es de esperarse que los resultados sean poco satisfactorios dada la gran cantidad de construcciones.

Se eligió una evaluación por medio de análisis en lugar de una evaluación práctica para no sesgar las respuestas con un antecedente basado en experiencia con el paradigma, ya que durante las pruebas iniciales del prototipo, se observó que quienes trabajan con SN sin tener experiencia en la parte naturalística, buscan plasmar ideas conforme a paradigmas que dominen. Por ejemplo, SN permite trabajar con una sintaxis más cercana al paradigma orientado a objetos, pero esto se logra a costa de sacrificar la expresividad naturalística que se espera del prototipo. Con base en lo anterior, se observa que se requiere de un proceso de "des-aprendizaje" del paradigma que se domine o, en su defecto, enfocarse a gente sin conocimientos de programación para reducir el sesgo en la evaluación de la expresividad de SN. Por otro lado, si se le enseña a programar en SN a los participantes, se obtendría otro sesgo que implica el ya conocer el paradigma y la sintaxis y por tanto, consideren expresivo el código no por su naturalidad, sino por la experiencia previa. Por último, se descartó permitir que los usuarios programaran ejercicios ya que como se reporta en [A. S. Bruckman, 1997] y en [Landhäußer y cols., 2014], las personas tratarán de construir oraciones en lenguaje natural sin considerar las restricciones del lenguaje (en este caso, de SN), en ambos trabajos se reportan problemas tanto en el proceso de traducción como en la interpretación del texto porque los participantes emplearon oraciones ambiguas o con elementos que las herramientas que reportan los autores no permiten.

Las pruebas se realizaron por medio de una escala de Likert [Likert, 1932] para una evaluación subjetiva de la expresividad, ya que la correctitud de una oración no depende únicamente de la gramática, sino también de su contexto. Por ejemplo, la oración *Colorless green ideas sleep furiously* descrita en [Chomsky, 1956] por Noam Chomsky es gramaticalmente correcta, pero carece de sentido semántico por las contradicciones que se generan entre *colorless* y *green*, y entre *sleep* y *furiosuly*, ya que definen ideas mutuamente excluyentes. Aunque SN permite establecer un contexto cualquiera para la oración de Chomsky sólo con una modificación menor (*the Colorless and Green Ideas sleep furiously*), las contradicciones en el significado de las palabras se mantienen cuando las personas leen la oración.

4.7.1. Evaluación de métricas

Las pruebas consistieron en evaluar por medio de la facilidad de lectura de Flesch, el nivel de grado Flesch-Kincaid y el índice niebla de Gunning para medir la legibilidad lingüística de los ocho escenarios de prueba que se presentan en esta tesis, en la tabla 4.1. En la sección 4.8 se presentan un análisis de los resultados.

Tabla 4.1: Métricas

Escenario	**Flesch**	**Flesch-Kincaid**	**Gunning**
1	67.4	5.3	4.2
2	79.2	3.8	4.2
3	67.9	5.3	4.4
4	81.9	3.3	6.5
5A	64.8	5.8	6.9
5B	71.0	4.3	4.7
6	58.5	6.6	8.6
7	64.7	5.3	4.4
8	75.4	4.6	7.3

4.7.2. Cuestionario

Para las pruebas se solicitó el apoyo de alumnos de la carrera de ingeniería en sistemas computacionales del Tecnológico Nacional de México, campus Orizaba, donde 49 alumnos revisaron ejemplos de programas hechos con SN. Los alumnos se encuentran en un rango de edad de entre 18 y 32 años. El 46.9 % de los alumnos consideran tener un nivel de programación medio, el 2 % consideran tener un nivel alto mientras que el 51.1 % consideró tener un nivel bajo o nulo. En el dominio del inglés, se observa que el 8.1 % de los alumnos consideran tener un nivel alto, el 53.1 % consideran tener un nivel medio, mientras que el 38.7 % consideraron tener un nivel bajo o nulo.

La escala se estableció con valores entre el 1 y el 4, donde 1 es *nada expresivo* y 4 es *totalmente expresivo*. Las pruebas consistieron en analizar desde instrucciones simples para entender su significado y contexto, hasta el análisis de programas completos que realizan tareas como operaciones aritméticas por medio de oraciones en inglés; programas que definen sustantivos, adjetivos y sus combinaciones; o el manejo de bases de datos. Por

otro lado, las preguntas 4 y 5 consistieron en dos preguntas cuyas opciones eran verdadero o falso.

El criterio para establecer las preguntas consistió evaluar la expresividad respecto al lenguaje natural de las construcciones más comunes de oraciones que SN permite y que son: oraciones con formato *sujeto-predicado*, *sujeto-verbo-objeto*, *sujeto-verbo-preposición-objeto*, *verbo-objeto* y *verbo-objeto-preposición-objeto*. Además, se evaluó la correctitud de algunos fragmentos de código, de modo que se mostraron diversas impresiones a los participantes para que a partir del código eligieran la salida correcta. Dado que un cuestionario de este tipo evalúa información subjetiva, su validez depende de qué tan entendible sea el mismo y de la coherencia de los resultados obtenidos respecto a los resultados esperados. Para evaluar el cuestionario, antes de su aplicación se le mostró a ocho personas a quienes se les solicitó que lo respondieran como si fueran encuestados, pero se les permitió hacer preguntas sobre lo mismo. Se observó que las preguntas se enfocaban a la clarificación de detalles del código, más no sobre la estructura del cuestionario, de modo que se consideró como válido, ya que este mecanismo de validación se reporta en [Walonick, 2013] y [Coombe y Davidson, 2015]. Además, al ser un cuestionario que se aplicó por medio de Google Forms, se siguieron los criterios que se reportan en [Naithani y cols., 2011].

4.7.2.1. Pregunta 1

Para instrucciones de asignación tales como *A is 25*, el 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 40.8 % mencionó que es *medianamente expresivo*, el 36.7 % lo calificó como *poco expresivo*, mientras que el 10.2 % lo describió como *nada expresivo*.

4.7.2.2. Pregunta 2

Para instrucciones de instanciación tales como *an Speed Formula with 10 as distance and 5 as time*, el 8.2 % mencionó que el lenguaje es *altamente expresivo*, el 51 % mencionó que es *medianamente expresivo*, el 38.8 % lo calificó como *poco expresivo*, mientras que el 2 % lo describió como *nada expresivo*.

4.7.2.3. Pregunta 3

Para instrucciones de invocación tales como *System prints the first Number where its value is 62 and newline*, el 20.4 % mencionó que el lenguaje es *altamente expresivo*, el 36.7 % mencionó que es *medianamente expresivo*, el 34.7 % lo calificó como *poco expresivo*, mientras que el 8.2 % lo describió como *nada expresivo*.

4.7.2.4. Pregunta 4

Con base en las siguientes instrucciones de asignación:

```
a is 25.    b is 54.    c is 62.    d is 86.    e is 35.
f is 28.    g is 55.    h is 33.    i is 24.    j is 5.
k is 2.
```

Para instrucciones de invocación con referencias indirectas tales como *System prints the last 2 Number and newline*, se observó que para la pregunta "¿qué variable (o variables) almacena el resultado de la instrucción?", el 55.1 % de los alumnos contestó correctamente que las variables J y K son las que se almacenan en la referencia indirecta.

4.7.2.5. Pregunta 5

Para instrucciones de invocación tales como *System prints the first Number and newline*, se observó que para la pregunta "¿qué valor se imprime a partir de la instrucción?", el 51 % de los alumnos contestó correctamente *25*.

4.7.2.6. Pregunta 6

Para instrucciones de definición de abstracciones tales como *noun Formula*, *adjective Percentage* y *adjective Speed*, el 6.1 % mencionó que el lenguaje es *altamente expresivo*, el 49 % mencionó que es *medianamente expresivo*, el 34.7 % lo calificó como *poco expresivo*, mientras que el 10.2 % lo describió como *nada expresivo*.

4.7.2.7. Pregunta 7

Para instrucciones de instanciación tales como *a Speed Formula* y *a Percentage Formula*, el 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 46.9 % mencionó que es *medianamente expresivo*, el 38.8 % lo calificó como *poco expresivo*, mientras que el 2 % lo describió como *nada expresivo*.

4.7.2.8. Pregunta 8

Dada la siguiente instrucción:

```
adjective Speed:
  attribute distance as a Real Number.
  attribute time as an Real Number.
  derived attribute result
   as a Real Number:
    distance / time; and return it.
```

Cuyo equivalente en Java es:

```
class SpeedFormula {
  float distance; float time;
  float result() { return distance / time; }}
```

El 22.4 % mencionó que el lenguaje es *altamente expresivo*, el 38.8 % mencionó que es *medianamente expresivo*, el 34.7 % lo calificó como *poco expresivo*, mientras que el 4.1 % lo describió como *nada expresivo*.

4.7.2.9. Pregunta 9

Para instrucciones de refinamiento de adjetivos, el 14.3 % mencionó que el lenguaje es *altamente expresivo*, el 57.1 % mencionó que es *medianamente expresivo*, el 26.5 % lo calificó como *poco expresivo*, mientras que el 2 % lo describió como *nada expresivo*.

4.7.2.10. Pregunta 10

Dadas las siguientes instrucciones:

```
an Speed Formula
 with 10 as distance
  and 5 as time;
 and System prints
  the result of it
  and newline.
spf is an Speed Formula
 with 10 as distance
 and 5 as time;
 and System prints
  the result of spf
  and newline.
System prints
 the result of an Speed Formula
  with 10 as distance
  and 5 as time and newline
System prints
 the result of a Percentage Formula
  with 10 as percent
  and 500 as quantity and newline.
```

El 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 46.9 % mencionó que es *medianamente expresivo*, el 36.7 % lo calificó como *poco expresivo*, mientras que el 4.1 % lo describió como *nada expresivo*.

4.7.2.11. Pregunta 11

Dado el siguiente ejemplo:

```
noun Person:
  circumstance:
   Male and Female are mutually excluded.
  circumstance:
   this requires Male or Female.
adjective Male.
adjective Female.
```

El 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 51 % mencionó que es *medianamente expresivo*, el 32.7 % lo calificó como *poco expresivo*, mientras que el 4.1 % lo describió como *nada expresivo*.

4.7.2.12. Pregunta 12

Dado el siguiente ejemplo:

```
noun Person:
  circumstance:
   this cannot be Sick or Dead.
  circumstance:
   this cannot be Wounded and Dead.
adjective Dead.
adjective Wounded.
adjective Sick.
```

El 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 49 % mencionó que es *medianamente expresivo*, el 28.6 % lo calificó como *poco expresivo*, mientras que el 10.2 % lo describió como *nada expresivo*.

4.7.2.13. Pregunta 13

Dado el siguiente ejemplo de código:

```
noun Test:
  verb test:
    System prints the first Number
     where its value is 62 and newline.
    numbers are the first
     five Numbers in this verb.
    System prints numbers and newline.
```

```
    System prints the first
     Number and newline.
    System prints the last
     2 Number and newline.
```

Cuyos resultados son los siguientes:

```
Valor asociado a it  : Variable
62                   : c
[25, 54, 62, 86, 35] : numbers
[25, 54, 62, 86, 35] : numbers
25                   : A
[5, 2]               : "those"
```

El 22.4 % mencionó que el lenguaje es *altamente expresivo*, el 38.8 % mencionó que es *medianamente expresivo*, el 32.7 % lo calificó como *poco expresivo*, mientras que el 6.1 % lo describió como *nada expresivo*.

4.7.2.14. Pregunta 14

Dado el siguiente ejemplo:

```
noun Person:
  attribute age is an Integer Number
   with 0 as value.
  attribute i is the real of "0".
  verb add aa as Integer Number to itself:
  add aa to age.
  verb itself print:
  System prints "Printing" and newline.
  verb itself print2:
  System prints "Printing2" and newline.
```

Para el manejo de circunstancias tales como *circumstance: add the i of this to the age of this after the age of this is assigned*, el 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 40.8 % mencionó que es *medianamente expresivo*, el 32.7 % lo calificó como *poco expresivo*, mientras que el 14.3 % lo describió como *nada expresivo*.

4.7.2.15. Pregunta 15

Con base en el ejemplo anterior, y dada la circunstancia *circumstance: this print after the age of something is assigned*, el 8.2 % mencionó que el lenguaje es *altamente expresivo*, el 57.1 % mencionó que es *medianamente expresivo*, el 20.4 % lo calificó como *poco expresivo*, mientras que el 14.3 % lo describió como *nada expresivo*.

4.7.2.16. Pregunta 16

Por último, dada la circunstancia *circumstance: it print2 instead something print*, el 22.4 % mencionó que el lenguaje es *altamente expresivo*, el 40.8 % mencionó que es *medianamente expresivo*, el 30.6 % lo calificó como *poco expresivo*, mientras que el 6.1 % lo describió como *nada expresivo*.

4.7.2.17. Pregunta 17

Dado el siguiente ejemplo:

```
a Persistent String with "BMW" as value.
System prints it and newline.
an Entity Car with the first String as model.
some Strings.
add "model" to these.
select the Strings from the Car.
System prints these and newline.
```

El 10.2 % mencionó que el lenguaje es *altamente expresivo*, el 53.1 % mencionó que es *medianamente expresivo*, el 28.6 % lo calificó como *poco expresivo*, mientras que el 8.2 % lo describió como *nada expresivo*.

4.7.2.18. Pregunta 18

Para gramáticas embebidas, el 32.7 % mencionó que el lenguaje es *muy útil*, el 59.2 % mencionó que es *medianamente útil*, el 8.2 % lo calificó como *poco útil*, mientras que nadie lo describió como *nada útil*.

4.7.2.19. Pregunta 19

Dado el siguiente ejemplo:

```
an Integer Number with 1 as value.
add 1 to the first Number.
System prints it and newline.
subtract 1 from the first Number.
System prints it and newline.
an String with "Hi" as value.
System prints it and newline.
52 + 4.
System prints it and newline.
```

El 12.2 % mencionó que el lenguaje es *altamente expresivo*, el 40.8 % mencionó que es *medianamente expresivo*, el 42.9 % lo calificó como *poco expresivo*, mientras que el 4.1 % lo describió como *nada expresivo*.

4.7.2.20. Pregunta 20

Dado el siguiente código:

```
execute the next 5 instructions when
 left is a Numeric.
| execute the next 4 instructions when
 right is a Numeric.
| | execute the next instruction when
 operator is equal to "+".
| | | the value of left plus the value
 of right; and return it.
| | execute the next instruction when
 operator is equal to "-".
| | | the value of left minus the value
 of right; and return it.
execute the next 4 instructions when
 left is a Variable.
| res is the string of the value of left.
| concat operator with it.
| concat the value of right with it.
| return res.
return "Error".
```

NOTA: los "|" no son parte del código, se colocaron para representar cómo se estructura la lógica de las instrucciones.

El 8.2 % mencionó que el lenguaje es *altamente expresivo*, el 59.2 % mencionó que es *medianamente expresivo*, el 22.4 % lo calificó como *poco expresivo*, mientras que el 10.2 % lo describió como *nada expresivo*. Por otro lado, 71.4 % de los encuestados mencionó que dicha estructuración es más compleja que la que se presenta en otros paradigmas de programación, mientras que el 28.6 % mencionó que no es más compleja.

4.7.2.21. Pregunta 21

Con base en el siguiente código:

```
numbers are some Numbers.
add 5 to numbers.
add 6 to numbers.
add 9 to numbers.
add 1 to numbers.
add 0 to numbers.
add 3 to numbers.
repeat the next instruction for each
 element of numbers as number.
| System prints number and newline.
System prints "Write a message: ".
System reads.
concat it with "The message is: ".
System prints it and newline.
```

NOTA: el "|" no es parte del código, se colocó para representar cómo se estructura la lógica de las instrucciones.

Respecto al uso de plurales, el 16.3 % mencionó que el lenguaje es *altamente expresivo*, el 55.1 % mencionó que es *medianamente expresivo*, el 22.4 % lo calificó como *poco expresivo*, mientras que el 6.1 % lo describió como *nada expresivo*.

4.7.2.22. Pregunta 22

Por último, los encuestados se refirieron a la utilidad de un lenguaje naturalístico de la siguiente forma: el 20.4 % mencionó que el lenguaje es *muy útil*, el 59.2 % mencionó que es *medianamente útil*, el 20.4 % lo calificó como *poco útil*, mientras que nadie describió como *nada útil*.

Tabla 4.2: Resultados de preguntas

Pregunta	**1**	**2**	**3**	**4**
1	10.2 %	36.7 %	40.8 %	12.2 %
2	2 %	38.8 %	51 %	8.2 %
3	8.2 %	34.7 %	36.7 %	20.4 %
6	10.2 %	34.7 %	49 %	6.1 %
7	2 %	38.8 %	46.9 %	12.2 %
8	4.1 %	34.7 %	38.8 %	22.4 %
9	2 %	26.5 %	57.1 %	14.3 %
10	4.1 %	36.7 %	46.9 %	12.2 %
11	4.1 %	32.7 %	51 %	12.2 %
12	10.2 %	28.6 %	49 %	12.2 %
13	6.1 %	32.7 %	38.8 %	22.4 %
14	14.3 %	32.7 %	40.8 %	12.2 %
15	14.3 %	20.4 %	57.1 %	8.2 %
16	6.1 %	30.6 %	40.8 %	22.4 %
17	8.2 %	28.6 %	53.1 %	10.2 %
18	0	8.2 %	59.2 %	32.7 %
19	4.1 %	42.9 %	40.8 %	12.2 %
20	10.2 %	22.4 %	59.2 %	8.2 %
21	6.1 %	22.4 %	55.1 %	16.3 %
22	0	20.4 %	59.2 %	20.4 %

Tabla 4.3: Preguntas verdadero/falso

Pregunta	**Sí**	**No**
4	55.1 %	44.9
5	51 %	49

En la Tabla 4.2, se presentan un resumen de los resultados para las preguntas respecto a los niveles de expresividad. Por otro lado, en la Tabla 4.3 se presentan los porcentajes de respuestas dadas a las dos preguntas cuyas opciones fueron verdadero o falso.

4.8. Discusión

Con base en lo que se observó durante el proceso de creación, depuración y pruebas de SN, el código resultante posee un mayor nivel de legibilidad lingüística, pero el código es más extenso comparado con sus contrapartes orientadas a objetos y aspectos. También se observa que aunque el uso de referencias indirectas es algo común en los lenguajes naturales, en programación es difícil de asimilar por parte de los programadores. Por otro lado, cuanta más experiencia tiene un programador con un paradigma, es más probable que intente crear abstracciones con base en ese paradigma, lo que da como resultado código SN ejecutable y correcto, pero sin la expresividad que se espera que dicho lenguaje provea.

A partir de la revisión de los trabajos relacionados, las pruebas que se realizaron con SN y los comentarios que se recibieron por parte de los expertos (ver Apéndice F), se concluye que el modelo posee los elementos mínimos para servir como base para el diseño e implementación de lenguajes naturalísticos de propósito general. Se llegó a dicha conclusión debido a que SN permite crear instrucciones con forma de oraciones simples con un nombre (variables) o pronombre (it o these), además de un predicado; oraciones donde el sujeto es un sintagma nominal (con o sin adjetivos), mientras que el predicado es un verbo con o sin complementos (que a su vez son sintagmas nominales); y además, permite el uso de oraciones gramaticales donde el sujeto va implícito y por tanto se omite (sujeto elíptico), mientras que el verbo posee complementos directo e indirecto. Además de lo anterior, una oración posee un contexto, mismo que se define por medio de las circunstancias que funcionan como un mecanismo de eventos para desacoplar el texto (oración) de su contexto.

También se observó una barrera que se esperaba desde que se inició esta investigación, la resistencia al cambio generó dificultades al momento de trabajar con SN. Dicha resistencia dio como resultado desde una dificultad o apatía para entender SN, hasta comentarios tales como "esto se hace con Java sin escribir tanto". El código extenso es otra dificultad, ya que al tender a la autodescripción se espera que las abstracciones posean nombres descriptivos y los verbos una sintaxis que los haga legibles de forma transparente. Por otro lado, para limitar la ambigüedad se restringió el uso de oraciones complejas donde una oración contuviera a otras oraciones, como en el caso de Java donde la expresión *Integer.ParseInt(JOptionPane.showInputDialog("Write a number"))* pide un número que se ingrese desde el teclado como cadena y se convierta en entero. En el caso de SN, esto se logra con dos instrucciones y una referencia indirecta:

```
System reads
the integer of it.
```

La mayoría de las limitantes que se observaron al trabajar con SN se relacionan con la API, esto se debe a que al ser un nuevo enfoque que se desprende de la sintaxis tradicional de los lenguajes de programación, se requiere de una API robusta que permita resolver problemas particulares. Por ejemplo, se requería de una biblioteca que permitiera trabajar con bases de datos, de modo que se analizó y adaptó una clase de Java que permite dicha conexión, como resultado se obtuvo un mecanismo de composición que convierte a cualquier sustantivo en una "clase entidad" independiente del dominio y el gestor. Otra limitante es el proceso de generación de bytecode, ya que al no ser la prioridad para este trabajo, se optó por usar Scala y AspectJ para la generación del mismo, esto dio como resultado que el proceso de generación sea lento, además de requerir de los compiladores de los lenguajes que se mencionan. Por último, el uso de los pronombres *it* y *these* dificulta en gran medida el conocer el tipo de dato que contiene, de modo que se optó por trabajar con *reflection* para evitar el casting explícito cuando se necesitara, como consecuencia, la ejecución no es eficiente en términos de velocidad y los errores de parámetros de tipo incorrecto sólo se conocen a tiempo de ejecución.

Con respecto al modelo, la implementación de SN integró todos los elementos que se propusieron para el modelo, a destacar el uso de sintagmas nominales que permiten un refinamiento de sustantivos en tiempo de instanciación, referencias indirectas por medio de sintagmas o pronombres para que el código sea naturalmente legible, además del uso de catáforas para condicionales e iteradores, mientras que el uso de catáforas se aplica a

estos y a la especificación de referencias indirectas para los datos que se encuentran en memoria. Con base en esto, se logró que SN implementara una deixis completa. Además, el uso de circunstancias permite establecer contextos mediante un mecanismo de eventos, de modo que estas en conjunto con el mecanismo de composición de adjetivos, logran un alto nivel de modularidad. Por otro lado, los identificadores se manejan como algo opcional, pero de igual forma se integraron en SN, de modo que no se presentan problemas si un programador desea mantener el uso tradicional de los mismos para trabajar y mantener un mayor control de los datos.

Con base en la propuesta de Cristina Videira [Lopes y cols., 2003] y en la revisión que se realizó al estado de la práctica, se identificaron los elementos necesarios para establecer un modelo donde se utilizan referencias indirectas, lo que a su vez permitió trabajar con referencias reflexivas donde el código está en función de las instrucciones "anteriores o siguientes", también se consideraron las referencias estructurales, donde se accede a los datos en función de su posición respecto a otros datos de su mismo tipo, o respecto al valor particular de alguno de sus atributos. Respecto a la implementación, la propuesta naturalística que se presenta en el artículo de Videira contiene varios elementos propios de lenguajes como Java o C, con SN se prescindió de ellos, ya que las firmas son inflexibles y poco expresivas, al tiempo que limitan la forma en que se construyen las oraciones, en su lugar se utilizó un mecanismo donde se especifica de forma explícita cómo se conforma el verbo para la oración, lo que dio mayor robustez a SN y una apariencia menos "procedural".

Por último, aunque SN implementa de forma exitosa los elementos del modelo, la naturaleza conceptual del mismo requiere de implementaciones donde otras personas generen una especificación personal del modelo, esto se debe a que SN representa la interpretación del modelo de sólo una persona, lo que deja pie a que se genere un sesgo en la validación del mismo.

A partir de las métricas de legibilidad lingüística se observó que la complejidad de lectura y comprensión de los escenarios abarca rangos entre 58.5 y el 81.9 para Flesch, pero en general todos los escenarios menos uno presentaron un nivel por encima de 60, esto implica que el código de todos los escenarios equivale a un nivel de inglés entendible para personas con estudios de octavo grado del sistema estadounidense. La escala Flesch-Kincaid indica que la puntuación abarcó desde el 3.3, hasta el 6.6. Por último, la métrica Gunning mostró resultados entre el 4.2 y el 8.6, lo que implica que todos los escenarios

tienen el nivel mínimo de complejidad para ser entendibles por cualquier persona que hable y lea inglés de forma nativa.

Se destaca el escenario 5B, que es la implementación no naturalística de la descripción de fórmulas, ya que según las escalas Flesch y Flesch-Kincaid, presenta un mayor nivel de legibilidad lingüística que la versión naturalística del mismo, lo que coincide con los valores que se obtuvieron con la métrica Gunning. Esto se debe a que el ejemplo utiliza gramáticas embebidas, que a un nivel matemático tan simple resultan en un mayor nivel de legibilidad lingüística respecto a la versión naturalística.

El escenario 6 presenta un alto nivel de complejidad de acuerdo a la métricas Flesch y Flesch-Kincaid indicando que se requiere de un nivel equivalente al sexto grado del sistema educativo estadounidense. Por otro lado, la escala Gunning indica que el código posee una complejida equivalente a un texto enfocado a un alumno de octavo grado del sistema educativo estadounidense.

Los valores de las escalas Flesch-Kincaid y Gunning dieron como resultado valores que no coinciden entre sí, ya que ambos muestran los grados educativos que se requieren para leer un texto, las diferencias varían desde *0.4* hasta *3.2*, esto se debe al enfoque de ambas escalas: mientras que Flesch-Kincaid emplea el total de sílabas, palabras y oraciones, Gunning emplea palabras, oraciones y palabras complejas. Gunning posee un sesgo ya que utiliza palabras complejas, que el autor define como palabras con tres o más sílabas, sin contar nombres propios, palabras compuestas o sufijos (-es, -ed, -ing), esto se debe a que se espera que una palabra con menos sílabas sea menos compleja de entender, lo que en la práctica no es así. Otra limitante que se observó es que ambas mediciones dieron como resultado que el código es legible por prácticamente cualquier persona con estudios de séptimo grado para Flesch-Kincaid y noveno para Gunning, lo que es anormal si se considera que los escenarios 6, 7 y 8 poseen tecnicismos que se esperaría en niveles académicos superiores.

Con base en los resultados, se observa que un análisis cuantitativo no refleja la complejidad real de un algoritmo implementado en SN, esto se debe a que al ser un lenguaje de programación, SN permite describir como válidas oraciones muy cortas con un contexto complejo; palabras muy largas que consisten en la unión de dos o más palabras simples, tales como *toCharArray*; o incluso palabras cortas para describir identificadores, tales como *spf*. Por otro lado, el análisis cuantitativo enfocado a lenguajes de programación busca

que el código se describa con la menor cantidad de caracteres posible, lo que restringe en gran medida la expresividad respecto al inglés, ya que se espera que se adjunte una correcta documentación para el código. Como resultado, se considera que se requiere del diseño de métricas que permitan una correcta evaluación de un lenguaje de programación naturalístico de propósito general donde no sólo se considere la longitud del código, sino también su contexto.

Respecto a los resultados de la encuesta, el lenguaje obtuvo un buen nivel de aceptación entre los encuestados, quienes al final mencionaron que la mayor limitante es el idioma, algo entendible si se considera que tan sólo un encuestado mencionó que el inglés es su lengua materna. Otro problema que se observó radica en que los encuestados, al tener conocimientos de lenguajes como Java, tienden a comparar y a estructurar las ideas en el paradigma orientado a objetos, lo que da como resultado una resistencia al cambio que se previó, ya que es un fenomeno que ocurre al surgimiento de un nuevo paradigma. La mayor prueba de esta resistencia se observó cuando se trató de explicar la encapsulación de código disperso entre objetos con AspectJ para su comparación con las circunstancias de SN, ya que se observó que los encuestados no pudieron asimilar el concepto con facilidad ya que en su experiencia (los alumnos no conocen AspectJ), el código se entiende mejor con la dispersión y no se requiere encapsular.

Capítulo 5

Conclusiones

Un modelo naturalístico establece los elementos a tomar en cuenta para el diseño e implementación de lenguajes de programación naturalísticos, considera elementos tales como las refrencias indirectas, un mayor peso a los verbos y el contexto de una instrucción en forma de circunstancias. El diseño e implemencación de un lenguaje naturalístico requiere de instrucciones con un mayor nivel de complejidad dado que el modelo le otorga mayor relevancia a la estructura gramatical de los verbos. Por otro lado, el uso de referencias indirectas en conjunto con un diseño complejo de verbos implica introducir ambigüedad al código, de modo que se requiere un análisis adicional para resolverla, por lo cual es ideal reducir el alcance de las expresiones ambiguas al verbo.

Con base en la experimentación y la opinión de dos expertos (véase Apéndice F), se considera que el modelo posee los elementos mínimos, se llegó a esta conclusión porque el modelo toma en cuenta que para formar oraciones se requiere de al menos un sujeto y un predicado. El sujeto es tanto un sustantivo o un sintagma nominal, mientras que el predicado se compone de verbos y opcionalmente, sus complementos directos e indirectos. Cabe destacar que la diferenciación entre complementos directo e indirecto es meramente lingüística, de modo que en una implementación dichos elementos son tratables de diversas formas, ya sea como parámetros o como contenedores del verbo. Además, aunque no son parte de las oraciones como tal, las circunstancias proveen un contexto para las mismas, lo que permite establecer precondiciones y postcondiciones para las oraciones. Por último, la deixis (particularmentre las anáforas) permite omitir información innecesaria o que es implícita desde la perspectiva del lenguaje natural. Por ejemplo, el sujeto elíptico permite

definir instrucciones imperativas donde se le dice a la computadora qué hacer, mientras que las relaciones anafóricas permiten definir referencias indirectas que no dependen de identificadores, sino del tipo de dato, la posición de la instancia o del valor de alguno de sus atributos.

Con base en la experimentación que se tiene hasta el momento, el modelo implementtado en SN no requiere de más elementos, de modo que se considera completo y por tanto, sirve como base para la definición de otros lenguajes de programación naturalística. Además, como se mencionó en el párrafo anterior, el modelo posee los elementos necesarios para formar oraciones procesables por la computadora sin ambigüedad, con un contexto específico, legibles por programadores y expertos, además de poseer un alto nivel de modularidad.

El lenguaje SN permite trabajar con instrucciones con un nivel de expresividad similar al idioma inglés, no requiere de un diccionario de entidades, ontologías o heurísticas para mantener la expresividad. Al ser un lenguaje de programación formal, las bibliotecas que se programan en SN funcionan como diccionario, además de que la formalidad y la ambigüedad controlada reemplazan a las técnicas de inteligencia artificial que se reportan en otros trabajos.

Programar con el lenguaje SN requiere de un proceso de aprendizaje dado que a diferencia de otros paradigmas, su sintaxis tiene la particularidad de que permite programar instrucciones imperativas que son autodescriptivas, pero a su vez un programador con conocimientos de objetos tiene la facilidad de construir abstracciones similares a las que se presentan en los lenguajes orientados a objetos. Aunque ambas implementaciones resuelvan el mismo problema, una implementación similar a la POO no necesariamente es naturalística, de modo que para este paradigma el proceso cognitivo aplicado a la escritura tiene un mayor peso que en los objetos.

En [A. Bruckman y Edwards, 1999] se plantea que emplear lenguaje natural es un inconveniente. Lo anterior resulta de 314 errores que se obtuvieron cuando emplearon lenguaje natural para programar, los errores se observaron cuando niños intentaron programar con la herramienta MOOSE Crossing, ya que ellos emplearon descripciones en inglés que la herramienta no interpretó de forma correcta. De estos errores, 147 son errores de sintaxis, 68 son errores de suposición, 58 son errores de interpretación, 14 consistieron en que los niños supusieron que el sistema entendería lo que escribían, y 7 son errores de precedencia

de operadores. Con esto en mente, un lenguaje naturalístico definido a partir de un modelo ayudaría a evitar la ambigüedad de un lenguaje natural. Por tanto, se concluye que la opción no es el uso de lenguaje natural propiamente dicho, sino el uso de una formalización del mismo que a su vez sea suficientemente flexible para que los usuarios expresen sus ideas de forma clara y descriptiva, pero al mismo tiempo restringiendo instrucciones altamente ambiguas o que carezcan de un contexto. El lenguaje SN trabaja por medio de una sintaxis naturalística rígida y a su vez requiere de un contexto en forma de una API que se programa también con el mismo lenguaje SN.

El modelo se estableció para el idioma inglés, por tanto no considera elementos de otras lenguas tales como el género o la aglutinación de conceptos. Además, no se consideraron otros elementos gramaticales como adverbios o pronombres personales. Los adverbios se dejaron de lado porque incrementan la complejidad de las oraciones, mientras que los pronombres personales se consideraron innecesarios, aunque es fácil implementarlos en el lenguaje SN por medio de identificadores. Dado que no se consideraron elementos aglutinantes, tampoco se consideró lo que en inglés se conoce como *portmanteau*, que básicamente es la generación de neologismos a partir de la combinación de dos o más palabras.

Por último, con base en los resultados de la evaluación por parte de estudiantes, se observó que la expresividad del código en SN es elevada y permite describir de forma coherente instrucciones cuyo significado se infiere de la lectura sin la necesidad de documentación adicional, aunque esto se refleja a partir de ejercicios muy simples y cuyo contexto es fácil de descifrar. Además, el Escenario 5 sirve como antecedente de que la expresividad de un lenguaje recibe influencia del programador, ya que es él quien define el contexto de las instrucciones. A mayor nivel de complejidad y con un equipo de desarrollo cada vez más grande, es de esperarse que la complejidad del código dificulte su correcta interpretación por parte de los programadores, pero a su vez se espera tal incremento en la dificultad de lectura sea menor respecto al de otros paradigmas ya que al ser especificaciones más cercanas al inglés, el programador invierte menos tiempo en leer e interpretar, siempre y cuando el código se diseñe de forma naturalística y no adaptando estilos de programación de otros paradigmas.

5.1. Trabajo a futuro

Como trabajo a futuro, se recomienda el diseño de más lenguajes naturalísticos que se basen en el modelo que se propone en esta tesis, de modo que otros enfoques permitan robustecer o, en su defecto, encontrar deficiencias en el modelo, esto se debe a que la naturaleza conceptual del modelo provoca que los mecanismos para su validación requieran de varios lenguajes y además, que se cuente con el apoyo de la industria del desarrollo de software.

Continuar con el refinamiento del prototipo permitirá optimizar su funcionamiento y resolver problemas de eficiencia en la generación de código. Otro punto a destacar es la generación de bytecode, para el primer prototipo se optó por utilizar una combinación de Scala con AspectJ, por tanto se tiene código intermedio en dos lenguajes de programación. Como trabajo a futuro se recomienda realizar un análisis que permita optimizar la generación de código y que además esta se lleve a cabo de forma directa sin pasar por lenguajes intermedios.

Complementar el modelado del lenguaje, de modo que se abarque la funcionalidad desde la definición de abstracciones, verbos y circunstancias, además de que se ajuste el trabajo que se presenta en esta tesis de acuerdo a la evolución que tenga el prototipo de lenguaje SN.

Analizar el modelo y el prototipo de lenguaje SN desde el punto de vista de la ingeniería de software, en particular desde la etapa de especificación de requerimientos y de este modo complementar y ajustar las definiciones que se tienen tanto del modelo, como del lenguaje SN. Verificar si los procesos de desarrollo de software actuales se ajustan al paradigma naturalístico, o si en caso contrario, se requiere de un proceso de desarrollo nuevo que englobe diversas etapas de análisis y diseño de software naturalístico.

El prototipo de lenguaje SN que se presenta en esta tesis trabaja con una API que contiene los elementos mínimos que se necesitaron para realizar pruebas, de modo que se recomienda robustecer la API de forma que se cubran más dominios tales como el manejo de archivos, interfaces gráficas o programación Web, por mencionar algunos.

Dado que el modelo que se presenta en esta tesis se basa en el idioma inglés, se requiere analizar qué elementos se requieren o se descartan para el diseño e implementación de lenguajes naturalísticos que se basen en otros idiomas, a destacar el caso del español dada

su complejidad gramatical, además de lenguas aglutinantes tales como el alemán, donde elementos lingüísticos como el género son relevantes a la hora de escribir oraciones.

Por último, es deseable conocer el grado de eficiencia (tanto en velocidad de ejecución como en expresividad) de la programación naturalística respecto a otros paradigmas, de forma que se aprecie cómo la reducción de los dominios del problema y la solución de este paradigma representa una ventaja respecto a otros paradigmas tales como el de la programación orientada a componentes, la programación funcional o la programación lógica.

Apéndice A

Productos académicos

Publicaciones JCR

- Oscar Pulido-Prieto, Ulises Juárez-Martínez, A Survey of Naturalistic Programming Technologies, ACM Comput. Surv. xx (x) (2017) 17:1-17:39. doi:10.1145/3037755. URL http://doi.acm.org/10.1145/3037755 (Citado por [Hsiao, 2018], [X. P. Mai, Pastore, Göknil, y Briand, 2018], [P. X. Mai, Pastore, Goknil, y Briand, 2019] y [Van Brummelen, 2019])
- Oscar Pulido-Prieto, Ulises Juárez-Martínez, A Model for Naturalistic Programming with Implementation, MDPI Appl. Sci, (2019) 9(18):3936, doi:10.3390/app9183936 URL https://doi.org/10.3390/app9183936
- Oscar Pulido Prieto, Ulises Juárez Martínez, Naturalistic Programming: Model and Implementation, IEEE Latin América Transactions, ISSN: 1548-0992, factor de impacto: 0.804. (enviado para segunda revisión).

Artefactos entregables

Prototipo de lenguaje de programación naturalístico de nombre SN.

Apéndice B

Modelado

En este apartado se presenta el modelado en Haskell de las oraciones del lenguaje SN.

B.1. Modelado de reglas de oraciones

```
module SNGrammar where
import System.IO
import Control.Monad
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language
import qualified Text.ParserCombinators.Parsec.Token as Token

-- DATA STRUCTURES
data Noun
 = Noun String
 deriving(Show)
data Adjective
 = Adjective String
 deriving(Show)
data AdjectivesAnd
 = AdjAnd Adjective
 | AdjsAnd AdjectivesAnd Adjective
 deriving(Show)
data Adjectives
 = AdjAP Adjective
 | AdjsAP AdjectivesAnd String Adjective
 deriving(Show)
data NounPhrase
 = NounNP Noun
 | AdjsNP Adjectives
```

```
 | AdjsNounNP Adjectives Noun
 deriving(Show)
-- DATA INSTRUCTIONS
data NatInstruction
 = Seq [NatInstruction]
 | NatAssign Assign
 | IndirectInstance NounPhrase
 | VerbCall NaturalisticSentence
 deriving(Show)
-- DATA ASSIGN
data Assign
 = AssignNP String NounPhrase
 | AssignID String String
 | AssignInt String Integer
 | AssignReal String Double
 | AssignBool String Bool
 deriving(Show)
-- DATA VERB
-- VERB COMPLIMENTS
data Compliment
 = CNP NounPositionalPhrase
 | CID String
 | CInt Integer
 | CReal Double
 | CBool Bool
 deriving(Show)
data ComplimentsAnd
 = CAnd Compliment
 | CsAnd ComplimentsAnd Compliment
 deriving(Show)
data Compliments
 = C Compliment
 | Cs ComplimentsAnd String Compliment
 deriving(Show)
-- RELATIONAL VERB
data EqualGreaterLesser
 = GT' String String
 | LT' String String
 | ET String String
 deriving(Show)
data GreaterLesserOrEqual
 = GET String String String String
 | LET String String String String
 deriving(Show)
-- GRAMMAR SUBJECT
data Subject
```

```
 = SubjectNP NounPositionalPhrase
 | SubjectID String
 | SubjectInt Integer
 | SubjectReal Double
 | SubjectBool Bool
 | SubjectFieldRef FieldRef
 deriving(Show)
data Ordinal
 = NamedO String
 | NthO String
 deriving(Show)
data NounPositionalPhrase
 = TheNthNounP String Ordinal NounPhrase
 | TheNounP String NounPhrase
 deriving(Show)
data FieldRef
 = TheRef String String String Subject -- the string of A
 | TheNestedRef FieldRef String Subject -- the string of the number of A
 deriving(Show)
-- VERB STRUCTURE
data NaturalisticSentence
 = SubjPred Subject String
 | PredObj String Subject
 | SubjPredObjs Subject String Compliments
 | SubjPredPrepObj Subject String String Compliment
 | PredObjsPrepObj String Compliments String Subject
 | ComparatorsGLT Subject String EqualGreaterLesser Compliment
 | ComparatorsGLEQT Subject String GreaterLesserOrEqual Compliment
 deriving(Show)

-- LEXER
languageDef =
  emptyDef { Token.identStart      = letter
           , Token.identLetter     = alphaNum
           , Token.reservedNames   = [ "is"
                                     , "a"
                                     , "an"
                                     , "and"
                                     , "of"
                                     , "true"
                                     , "false"
                                     , ".\n"
                                     , "the"
                                     , "first"
                                     , "second"
                                     , "third"
                                     , "fourth"
                                     , "sixth"
                                     , "seventh"
                                     , "eighth"
```

```
                                    , "ninth"
                                    , "tenth"
                                    , "it"
                                    ,"these"
                                    ]
           , Token.reservedOpNames = [
                                    ]
           }
lexer = Token.makeTokenParser languageDef

identifier = Token.identifier    lexer -- parses an identifier
reserved   = Token.reserved      lexer -- parses a reserved name
integer    = Token.integer       lexer -- parses an integer
real       = Token.float         lexer -- parses a float
character  = Token.charLiteral   lexer -- parses a float
string     = Token.stringLiteral lexer -- parses a float
dot        = Token.dot           lexer -- parses a dot
comma      = Token.comma         lexer -- parses a comma
semi       = Token.semi          lexer -- parses a semicolon
ws         = Token.whiteSpace    lexer -- parses whitespace

-- MAIN PARSER

verbBody :: Parser NatInstruction
verbBody = ws >> natInstruction

natInstruction :: Parser NatInstruction
natInstruction
 = natInstructions

endLine :: Parser String
endLine = dot

natInstructions =
 do
  list <- (sepBy1 natInstruction' dot)
  return $ if length list == 1 then head list else Seq list

-- INSTRUCCIONES

natInstruction' :: Parser NatInstruction
natInstruction'
 = indirectInstance
 <|> naturalisticSentence

-- ASIGNACION

natAssignSentence :: Parser NatInstruction
natAssignSentence
 = natAssignID
 <|> natAssignANP
 <|> natAssignAnNP
 <|> natAssignInt
 <|> natAssignReal
 <|> natAssignTrue
 <|> natAssignFalse

natAssignID :: Parser NatInstruction
natAssignID =
 do
```

```
  var <- identifier
  reserved "is"
  expr <- identifier
  return $ NatAssign $ AssignID var expr

natAssignANP :: Parser NatInstruction
natAssignANP =
 do
  var <- identifier
  reserved "is"
  reserved "a"
  np <- nounPhrase
  return $ NatAssign $ AssignNP var np

natAssignAnNP :: Parser NatInstruction
natAssignAnNP =
 do
  var <- identifier
  reserved "is"
  reserved "an"
  np <- nounPhrase
  return $ NatAssign $ AssignNP var np

natAssignInt :: Parser NatInstruction
natAssignInt =
 do
  var <- identifier
  reserved "is"
  i <- integer
  return $ NatAssign $ AssignInt var i

natAssignReal :: Parser NatInstruction
natAssignReal =
 do
  var <- identifier
  reserved "is"
  f <- real
  return $ NatAssign $ AssignReal var f

natAssignTrue :: Parser NatInstruction
natAssignTrue =
 do
  var <- identifier
  reserved "is"
  reserved "true"
  return $ NatAssign $ AssignBool var True

natAssignFalse :: Parser NatInstruction
natAssignFalse =
 do
  var <- identifier
  reserved "is"
  reserved "false"
  return $ NatAssign $ AssignBool var False

-- INSTANCIA INDIRECTA

indirectInstance :: Parser NatInstruction
indirectInstance
 = indirectA
 <|> indirectAn

indirectA :: Parser NatInstruction
indirectA =
```

```
 do
  reserved "a"
  np <- nounPhraseInstance
  dot
  return $ IndirectInstance np

indirectAn :: Parser NatInstruction
indirectAn =
 do
  reserved "an"
  np <- nounPhraseInstance
  dot
  return $ IndirectInstance np

nounPhraseInstance :: Parser NounPhrase
nounPhraseInstance
 = nounPhraseNoun
 <|> nounPhraseAdjectivesNoun

-- NOUN PHRASE

nounPhrase :: Parser NounPhrase
nounPhrase
 = nounPhraseNoun
 <|> nounPhraseAdjectives
 <|> nounPhraseAdjectivesNoun

nounPhraseAdjectivesNoun :: Parser NounPhrase
nounPhraseAdjectivesNoun =
 do
  adjs <- adjectivePhrase
  noun' <- noun
  return $ AdjsNounNP adjs noun'

nounPhraseAdjectives :: Parser NounPhrase
nounPhraseAdjectives =
 do
  adjs <- adjectivePhrase
  return $ AdjsNP adjs

nounPhraseNoun :: Parser NounPhrase
nounPhraseNoun =
  do
   noun' <- noun
   return $ NounNP noun'

-- NOUN

noun :: Parser Noun
noun =
 do
  noun' <- identifier
  return $ Noun noun'

-- ADJECTIVE

adjectivePhrase :: Parser Adjectives
adjectivePhrase
 = adjective
 <|> andAdjective
 <|> andAdjectives

adjective' :: Parser Adjective
adjective' =
 do
```

```
  adj <- identifier
  return $ Adjective adj

adjectives'' :: Parser AdjectivesAnd
adjectives''
 = adjectives
 <|> adjectives'

adjectives :: Parser AdjectivesAnd
adjectives =
 do
  adj1 <- adjective'
  comma
  adj2 <- adjective'
  return $ AdjsAnd (AdjAnd adj1) adj2

adjectives' :: Parser AdjectivesAnd
adjectives' =
 do
  adj1 <- adjectives
  comma
  adj2 <- adjective'
  return $ AdjsAnd adj1 adj2 --  return $ AdjsAP adjs adj

adjective :: Parser Adjectives
adjective =
 do
  adj <- adjective'
  return $ AdjAP adj

andAdjective :: Parser Adjectives
andAdjective =
 do
  adj1 <- adjective'
  reserved "and"
  adj2 <- adjective'
  return $ AdjsAP (AdjAnd adj1) "and" adj2

andAdjectives :: Parser Adjectives
andAdjectives =
 do
  adjs <- adjectives''
  reserved "and"
  adj <- adjective'
  return $ AdjsAP adjs "and" adj

-- NATURALISTIC SENTENCE

naturalisticSentence :: Parser NatInstruction
naturalisticSentence
 =
  do
   s <- naturalisticSentence'
   return $ VerbCall s

naturalisticSentence' :: Parser NaturalisticSentence
naturalisticSentence'
 = naturalisticSentenceSP
 <|> naturalisticSentencePO
 <|> naturalisticSentenceSPOs
 <|> naturalisticSentenceSPpO
 <|> naturalisticSentencePOspO
```

```
 <|> naturalisticSentenceCGLT
 <|> naturalisticSentenceCGLEQT

naturalisticSentenceSP :: Parser NaturalisticSentence
naturalisticSentenceSP
 =
  do
   s <- subject
   v <- identifier
   return $ SubjPred s v

naturalisticSentencePO :: Parser NaturalisticSentence
naturalisticSentencePO
 =
  do
   v <- identifier
   s <- subject
   return $ PredObj v s

naturalisticSentenceSPOs :: Parser NaturalisticSentence
naturalisticSentenceSPOs
 =
  do
   s <- subject
   v <- identifier
   o <- compliments
   return $ SubjPredObjs s v o

naturalisticSentenceSPpO :: Parser NaturalisticSentence
naturalisticSentenceSPpO
 =
  do
   s <- subject
   v <- identifier
   p <- identifier
   o <- compliment
   return $ SubjPredPrepObj s v p o

naturalisticSentencePOspO :: Parser NaturalisticSentence
naturalisticSentencePOspO
 =
  do
   v <- identifier
   o <- compliments
   p <- identifier
   s <- subject
   return $ PredObjsPrepObj v o p s

naturalisticSentenceCGLT :: Parser NaturalisticSentence
naturalisticSentenceCGLT
 =
  do
   s <- subject
   reserved "is"
   glt <- greaterLesser
   o <- compliment
   return $ ComparatorsGLT s "is" glt o

naturalisticSentenceCGLEQT :: Parser NaturalisticSentence
naturalisticSentenceCGLEQT
 =
  do
   s <- subject
   reserved "is"
   gloet <- greaterLesserOrEqual
```

```
   o <- compliment
   return $ ComparatorsGLEQT s "is" gloet o

-- ((GREATER | LESSER) THAN | EQUAL TO)

greaterLesser :: Parser EqualGreaterLesser
greaterLesser
 = greaterThan
 <|> lesserThan
 <|> equalTo

-- GREATER THAN

greaterThan :: Parser EqualGreaterLesser
greaterThan
 =
  do
   reserved "greater"
   reserved "than"
   return $ GT' "greater" "than"

-- LESSER THAN

lesserThan :: Parser EqualGreaterLesser
lesserThan
 =
  do
   reserved "lesser"
   reserved "than"
   return $ LT' "lesser" "than"

-- EQUAL TO

equalTo :: Parser EqualGreaterLesser
equalTo
 =
  do
   reserved "equal"
   reserved "to"
   return $ ET "equal" "to"

-- (GREATER | LESSER) OR EQUAL THAN

greaterLesserOrEqual :: Parser GreaterLesserOrEqual
greaterLesserOrEqual
 = greaterOrEqualThan
 <|> lesserOrEqualThan

-- GREATER OR EQUAL THAN

greaterOrEqualThan :: Parser GreaterLesserOrEqual
greaterOrEqualThan
 =
  do
   reserved "greater"
   reserved "or"
   reserved "equal"
   reserved "than"
   return $ GET "greater" "or" "equal" "than"

-- LESSER OR EQUAL THAN

lesserOrEqualThan :: Parser GreaterLesserOrEqual
lesserOrEqualThan
 =
```

```
  do
   reserved "lesser"
   reserved "or"
   reserved "equal"
   reserved "than"
   return $ LET "lesser" "or" "equal" "than"
-- SUBJECT
subject :: Parser Subject
subject
 = subjectNP
 <|> subjectFieldRef
 <|> subjectInt
 <|> subjectReal
 <|> subjectIt
 <|> subjectThese
-- SUBJECT NOUN PHRASE
subjectNP :: Parser Subject
subjectNP
 =
  do
   np <- singularNounPositionalPhrase
   return $ SubjectNP np
-- SUBJECT ID
subjectID :: Parser Subject
subjectID
 =
  do
   id <- identifier
   return $ SubjectID id
-- SUBJECT INTEGER
subjectInt :: Parser Subject
subjectInt
 =
  do
   i <- integer
   return $ SubjectInt i
-- SUBJECT REAL
subjectReal :: Parser Subject
subjectReal
 =
  do
   r <- real
   return $ SubjectReal r
-- SUBJECT IT THESE
subjectIt :: Parser Subject
subjectIt
 =
  do
   reserved "it"
   return $ SubjectID "these"
subjectThese :: Parser Subject
```

```
subjectThese
 =
  do
   reserved "these"
   return $ SubjectID "these"

-- SUBJECT FIELD REF

subjectFieldRef :: Parser Subject
subjectFieldRef
 = subjectSimpleFieldRef'
 <|> subjectNestedFieldRef'

subjectSimpleFieldRef' :: Parser Subject
subjectSimpleFieldRef'
 =
  do
   s <- subjectSimpleFieldRef
   return $ SubjectFieldRef s

subjectNestedFieldRef' :: Parser Subject
subjectNestedFieldRef'
 =
  do
   s <-subjectNestedFieldRef
   return $ SubjectFieldRef s

subjectSimpleFieldRef :: Parser FieldRef
subjectSimpleFieldRef
 =
  do
   reserved "the"
   id <- identifier
   reserved "of"
   s <- subject
   return $ TheRef "the" id "of" s

subjectNestedFieldRef :: Parser FieldRef
subjectNestedFieldRef
 =
  do
   r <- subjectSimpleFieldRef
   reserved "of"
   s <- subject
   return $ TheNestedRef r "of" s

-- COMPLIMENT

compliment :: Parser Compliment
compliment
 = complimentNP
 <|> complimentInt
 <|> complimentDouble
 <|> complimentIt
 <|> complimentThese

complimentNP :: Parser Compliment
complimentNP =
 do
  np <- singularNounPositionalPhrase
  return $ CNP np
```

```
complimentID :: Parser Compliment
complimentID =
 do
  id <- identifier
  return $ CID id

complimentInt :: Parser Compliment
complimentInt =
 do
  i <- integer
  return $ CInt i

complimentDouble :: Parser Compliment
complimentDouble =
 do
  r <- real
  return $ CReal r

complimentIt :: Parser Compliment
complimentIt
 =
  do
   reserved "it"
   return $ CID "these"

complimentThese :: Parser Compliment
complimentThese
 =
  do
   reserved "these"
   return $ CID "these"

-- COMPLIMENTS

compliments :: Parser Compliments
compliments
 = compliment'
 <|> complimentsAnd

compliment' :: Parser Compliments
compliment'
 =
  do
   c <- compliment
   return $ C c


complimentsComma :: Parser ComplimentsAnd
complimentsComma
 =
  do
   c1 <- compliments'
   comma
   c2 <- compliment
   return $ CsAnd c1 c2

compliments' :: Parser ComplimentsAnd
compliments'
 = compliment''
 <|> complimentsComma

compliment'' :: Parser ComplimentsAnd
compliment''
```

```
 =
  do
   c <- compliment
   return $ CAnd c

complimentsAnd :: Parser Compliments
complimentsAnd
 =
  do
   c1 <- compliments'
   reserved "and"
   c2 <- compliment
   return $ Cs c1 "and" c2

-- NOUN POSITIONAL PHRASE

singularNounPositionalPhrase :: Parser NounPositionalPhrase
singularNounPositionalPhrase =
 do
  reserved "the"
  pos <- position
  np <- nounPhrase
  return $ TheNthNounP "the" pos np

singularNounPositionalPhrase' :: Parser NounPositionalPhrase
singularNounPositionalPhrase' =
 do
  reserved "the"
  np <- nounPhrase
  return $ TheNounP "the" np

-- ORDINALS

position :: Parser Ordinal
position
 = first
 <|> second
 <|> third
 <|> fourth
 <|> fifth
 <|> sixth
 <|> seventh
 <|> eighth
 <|> ninth
 <|> tenth

first :: Parser Ordinal
first
 =
  do
   reserved "first"
   return $ NamedO "first"

second :: Parser Ordinal
second
 =
  do
   reserved "second"
   return $ NamedO "second"

third :: Parser Ordinal
third
 =
  do
```

```
   reserved "third"
   return $ NamedO "third"

fourth :: Parser Ordinal
fourth
 =
  do
   reserved "fourth"
   return $ NamedO "fourth"

fifth :: Parser Ordinal
fifth
 =
  do
   reserved "fifth"
   return $ NamedO "fifth"

sixth :: Parser Ordinal
sixth
 =
  do
   reserved "sixth"
   return $ NamedO "sixth"

seventh :: Parser Ordinal
seventh
 =
  do
   reserved "seventh"
   return $ NamedO "seventh"

eighth :: Parser Ordinal
eighth
 =
  do
   reserved "eighth"
   return $ NamedO "eighth"

ninth :: Parser Ordinal
ninth
 =
  do
   reserved "ninth"
   return $ NamedO "ninth"

tenth :: Parser Ordinal
tenth
 =
  do
   reserved "tenth"
   return $ NamedO "tenth"

-- MAIN

parseString :: String -> NatInstruction
parseString str =
  case parse verbBody "" str of
    Left e  -> error $ show e
    Right r -> r

parseFile :: String -> IO NatInstruction
parseFile file =
  do
   program <- readFile file
   case parse verbBody "" program of
    Left e -> print e >> fail "parse error"
    Right r -> return r
```

Apéndice C

Escenarios de prueba

En este apartado se presentan los ejemplos completos de los escenarios de prueba que se describen en la sección 4.6

C.1. Escenario 6: analizador de expresiones

C.1.1. Adjetivo Numeric

```
adjective Numeric:
  attribute value is 0.
  verb print itself:
    System prints value.
    return this.
  verb itself calculates:
    return value.
  overrided derived attribute string as an String:
    return the string of value.
```

C.1.2. Adjetivo Variable

```
adjective Variable:
  attribute value is "X".
  verb print itself:
    System prints value.
    return this.
  verb itself calculates:
    return value.
  overrided derived attribute string as an String:
    return value.
```

C.1.3. Adjetivo Binary

```
adjective Binary:
  attribute left as an Expression.
  attribute operator as an String.
  attribute right as an Expression.
  verb print itself:
    print left.
    System prints operator.
    print right.
    return this.
  verb itself calculates:
    execute the next 5 instructions when left is a Numeric.
    execute the next 4 instructions when right is a Numeric.
    execute the next instruction when operator is equal to "+".
    the value of left plus the value of right; and return it.
    execute the next instruction when operator is equal to "-".
    the value of left minus the value of right; and return it.
    execute the next 4 instructions when left is a Variable.
    res is the string of the value of left.
    concat operator with it.
    concat the value of right with it.
    return res.
    return "Error".
  overrided derived attribute string as an String:
    an String with the string of left as value; add operator to it;
     add the string of right to it; and return it.
```

C.1.4. Adjetivo Unary

```
adjective Unary:
  attribute operator as an String.
  attribute value as an Expression.
  verb print itself:
    System prints operator.
    print value.
    return this.
  verb itself calculates:
    execute the next 4 instructions when value is a Numeric Expression.
    execute the next instruction when operator is equal to "+".
    return value.
    execute the next instruction when operator is equal to "-".
    subtract the value of value from 0; and return value.
    execute the next 5 instructions when left is a Variable.
    res is "".
    execute the next instruction when operator is equal to "-".
    concat operator to res.
    concat the value of value to it.
    return res.
    return "Error".
  overrided derived attribute string as an String:
    an String with operator as value; add the string of value to it;
     and return it.
```

C.1.5. Sustantivo Parser

```
noun Parser:
  attribute operators are '-' and '+'.
  verb itself parses str as String:
    this orders str.
    this parses the array of it.
    return it.
  verb itself orders str as String:
    some Characters.
    some Characters are the array of str.
    execute the next 4 instructions when str matches
     "(([-+]|\\s*)*\\s*(\\d+|[a-zA-z]+))+".
    repeat the next 3 instructions until the size of
     the second Characters is equal to 0.
    remove 1 from the second Characters.
    execute the next instruction when it is distinct to ' '.
    add it to the first Characters.
    execute the next instruction when the size of
     the first Characters is greater than 0.
    return the string of the first Characters.
    return "Wrong format".
  verb itself parses characters as Characters:
    execute the next instruction when the size of characters is equal to 0.
    return null.
    operators contains the head of characters.
    execute the next instruction when the Boolean is true.
    remove 1 from characters; and this validates characters and it.
    execute the next instruction when the Boolean is false.
    this validates characters.
    execute the next instruction when it is an String.
    return it.
    execute the next 5 instructions when the size of
     characters is distinct to 0.
    operators contains the head of characters.
    execute the next instruction when the Boolean is true.
    remove 1 from characters; and a String with the string of
     the Character as value.
    execute the next instruction when the size of
     characters is greater than 0.
    this parses characters.
    execute the next instruction when the size of
     all Expressions in this verb is equal to 2.
    a Binary Expression with the first Expression as left,
     the String as operator and the second Expression as right;
     and return it.
    execute the next instruction when the size of
     all Expressions in this verb is equal to 1.
    return the last Expression.
    return "Empty expression".
  verb itself validates characters as Characters:
    an String with "" as value.
    repeat the next instruction until operators contains the head of characters.
    remove 1 from characters; and add it to the String.
    this validates the String.
    return it.
  verb itself validates characters as Characters and
   op as Character Number:
    an String with "" as value.
```

```
    repeat the next instruction until operators contains
     the head of characters.
    remove 1 from characters; and add it to the String.
    this validates the String.
    execute the next instruction when it is an String.
    return the String.
    a Unary Expression with the string of op as operator and
     the first Expression as value; and return it.
  verb itself validates str as String:
    execute the next instruction when str matches "[0-9]+".
    return a Numeric Expression with the integer of str as value.
    execute the next instruction when str matches "[A-Za-z]+".
    return a Variable Expression with str as value.
    return "Wrong format".
```

C.2. Escenario 7: conexión a base de datos de PostgreSQL

C.2.1. Select

```
import naturalistic.sql.DBEntity as Entity.
import naturalistic.sql.DBPersistent as Persistent.
main Select:
  System prints "-----------------------------------" and newline.
  a Persistent String with "BMW" as value.
  System prints it and newline.
  an Entity Car with the first String as model.
  the driver of the Car is "org.postgresql.Driver".
  the jar of the Car is "C:\\SN_examples\\lenguaje\\Escenarios
   \\database\\postgresql-42.2.1.jar".
  the user of the Car is "admin".
  the password of the Car is "admin1".
  the URL of the Car is "jdbc:postgresql://localhost:5432/Agenda".
  some Strings.
  add "model" to these.
  select the Strings from the Car.
  System prints these and newline.
  System prints "-----------------------------------" and newline.
  a Persistent String with "Person1" as value.
  an Entity Person with the String as name.
  the driver of the Person is "org.postgresql.Driver".
  the jar of the Person is "C:\\SN_examples\\lenguaje\\Escenarios
   \\database\\postgresql-42.2.1.jar".
  the user of the Person is "admin".
  the password of the Person is "admin1".
  the URL of the Person is "jdbc:postgresql://localhost:5432/Agenda".
  some Strings.
  add "name" to these.
```

```
  add "age" to these.
  select the Strings from the Person.
  System prints these and newline.
```

C.2.2. Insert

```
import naturalistic.sql.DBEntity as Entity.
import naturalistic.sql.DBPersistent as Persistent.
main Insert:
  System prints "-----------------------------------" and newline.
  an Entity Person.
  the driver of the Person is "org.postgresql.Driver".
  the jar of the Person is "C:\\SN_examples\\lenguaje\\Escenarios
   \\database\\postgresql-42.2.1.jar".
  the user of the Person is "admin".
  the password of the Person is "admin1".
  the URL of the Person is "jdbc:postgresql://localhost:5432/Agenda".
  a Persistent and Integer Number with 25 as value.
  the age of the Person is it.
  a Persistent String with "Person2" as value.
  the name of the Person is it.
  some Strings.
  add "name" to these.
  add "age" to these.
  insert the Strings into the Person.
  select the Strings from the Person.
  System prints these and newline.
```

C.2.3. Delete

```
import naturalistic.sql.DBEntity as Entity.
import naturalistic.sql.DBPersistent as Persistent.
main Delete:
  System prints "-----------------------------------" and newline.
  an Entity Person.
  a Persistent String with "Person2" as value.
  the name of the Person is it.
  the driver of the Person is "org.postgresql.Driver".
  the jar of the Person is "C:\\SN_examples\\lenguaje\\Escenarios
   \\database\\postgresql-42.2.1.jar".
  the user of the Person is "admin".
  the password of the Person is "admin1".
  the URL of the Person is "jdbc:postgresql://localhost:5432/Agenda".
  some Strings.
  add "name" to these.
  add "age" to these.
  delete the Strings from the Person.
  System prints these and newline.
```

C.2.4. Update

```
import naturalistic.sql.DBEntity as Entity.
import naturalistic.sql.DBPersistent as Persistent.
main Update:
  System prints "-----------------------------------" and newline.
  an Entity Person.
  the driver of the Person is "org.postgresql.Driver".
  the jar of the Person is "C:\\SN_examples\\lenguaje\\Escenarios
   \\database\\postgresql-42.2.1.jar".
  the user of the Person is "admin".
  the password of the Person is "admin1".
  the URL of the Person is "jdbc:postgresql://localhost:5432/Agenda".
  some Strings.
  add "age" to these.
  a Persistent and Integer Number with 50 as value.
  the age of the Person is it.
  a Persistent String with "Person1" as value.
  the name of the Person is it.
  update these from the Person.
  select the Strings from the Person.
  System prints these and newline.
```

C.3. Escenario 8: rule 110

C.3.1. Implementación en Java

```
public class Automaton {
  public static void main(String...a) throws Exception {
    String cad = "00000000000001";
    new Tape(cad, 10, '1','0').start();}}
class Tape {
  private Cell[] tape;
  private int cycles = -1;
  private String entry;
  private char dead = ' ';
  private char alive = 'X';
  public Tape(String entry) throws Exception {
    tape = new Cell[entry.length()];
    this.entry = entry;}
  public Tape(String entry, int cycles) throws Exception {
    this(entry);
    this.cycles = cycles;}
  public Tape(String entry, char alive, char dead)
   throws Exception {
    this(entry);
    this.alive = alive;
```

```
  this.dead = dead;}
public Tape(String entry, int cycles, char alive, char dead)
 throws Exception {
  this(entry, alive, dead);
  this.cycles = cycles;}
public void start() throws Exception {
  breakEntry(entry);
  int i = 0;
  while(i != cycles) {
    printPattern();
    nextPattern();
    i++;}}
private void breakEntry(String entry) throws Exception {
  int i = 0;
  for(char symbol : entry.toCharArray()) {
    if(symbol == alive) {
      tape[i++] = new LivingCell(alive);
    } else if(symbol == dead) {
      tape[i++] = new DeadCell(dead);
    } else {
      throw new Exception("Illegal value at: " + symbol);}}}
public void printPattern() {
  for(Cell c : tape) {
    System.out.print(c);}
  System.out.print("\n");}
public void nextPattern() {
  Cell[] nextTape = new Cell[tape.length];
  for(int i = 0; i < tape.length; i++) {
    int l = 0; int c = 0; int r = 0;
    if(i == 0) {
      l = tape.length-1; c = i; r = i+1;
    } else if(i == tape.length-1) {
      l = i-1; c = i; r = 0;
    } else {
      l = i-1; c = i; r = i+1;}
    nextTape[i] = fillAux(l, c, r);}
  tape = nextTape;}
public Cell fillAux(int l, int c, int r) {
  Cell[] auxTape = new Cell[3];
  auxTape[0] = tape[l];
  auxTape[1] = tape[c];
  auxTape[2] = tape[r];
  return nextValidator(auxTape);}
public Cell nextValidator(Cell[] group) {
  if(group[0].isAlive() && group[1].isAlive()
   && group[2].isAlive()) {//111
    return new DeadCell(dead);
  } else if(group[0].isAlive() && group[1].isAlive()
   && !group[2].isAlive()) {//110
    return new LivingCell(alive);
```

```
    } else if(group[0].isAlive() && !group[1].isAlive()
     && group[2].isAlive()) {//101
      return new LivingCell(alive);
    } else if(group[0].isAlive() && !group[1].isAlive()
     && !group[2].isAlive()) {//100
      return new DeadCell(dead);
    } else if(!group[0].isAlive() && group[1].isAlive()
     && group[2].isAlive()) {//011
      return new LivingCell(alive);
    } else if(!group[0].isAlive() && group[1].isAlive()
     && !group[2].isAlive()) {//010
      return new LivingCell(alive);
    } else if(!group[0].isAlive() && !group[1].isAlive()
     && group[2].isAlive()) {//001
      return new LivingCell(alive);
    } else if(!group[0].isAlive() && !group[1].isAlive()
     && !group[2].isAlive()) {//000
      return new DeadCell(dead);}
    return null;}
  public static String reformat(String entry, char[] originals,
   char[] replaced) {
    return entry.replace(originals[0],
     replaced[0]).replace(originals[1], replaced[1]);}}
abstract class Cell {
  private final boolean alive;
  private final char state;
  public Cell(boolean alive, char state) {
    this.alive = alive;
    this.state = state;}
  public String toString() {return "" + state;}
  public boolean isAlive() {return alive;}}
class LivingCell extends Cell {
  public LivingCell(char state) {super(true, state);}}
class DeadCell extends Cell {
  public DeadCell(char state) {super(false, state);}}
```

C.3.2. Implementación en SN

```
main Automaton:
  cad is "00000000000001";
  a Tape with '0' as dead, '1' as alive, 10 as cycles and cad as entry.
  start it.
noun Tape:
  attribute tape are some Cells.
  attribute cycles is -1.
  attribute entry as a String.
  attribute dead is ' '.
  attribute alive is 'X'.
  verb start itself:
```

```
  format this.
  cycle is 0.
  repeat the next 2 instructions until cycle is equal to cycles.
  print this.
  add 1 to cycle.
verb print itself:
  System prints the string of tape.
  System prints newline.
  update this.
verb update itself:
  updated are some Cells.
  counter is 0.
  left is 0.
  center is 0.
  right is 0.
  repeat the next 10 instructions until counter is equal to
   the size of tape.
  execute the next instruction when counter is equal to 0.
  the size of tape - 1; left is it; center is counter; counter + 1;
   and right is it.
  the size of tape - 1.
  execute the next instruction when counter is equal to it.
  counter - 1; left is it; center is counter; and right is 0.
  the size of tape - 1.
  execute the next instruction when counter is distinct to it.
  counter - 1; left is it; center is counter; counter + 1; right is it;
   and execute when counter is distinct to 0.
  select left, center and right from this; and add it to updated.
  add 1 to counter.
  tape is updated.
verb select left as Integer, center as Integer and right as Integer
 from itself:
  auxiliar are some Cells.
  get left from tape; and add it to auxiliar.
  get center from tape; and add it to auxiliar.
  get right from tape; and add it to auxiliar.
  this parses auxiliar; and return it.
verb itself parses auxiliar as Cells:
  get 0 from auxiliar; get 1 from auxiliar; and get 2 from auxiliar.
  execute the next 3 instructions when the state of the first Cell
   is equal to true.
  execute the next 2 instructions when the state of the second Cell
   is equal to true.
  execute the next instruction when the state of the third Cell
   is equal to true.
  return a Dead Cell with dead as value.
  execute the next 3 instructions when the state of the first Cell
   is equal to true.
  execute the next 2 instructions when the state of the second Cell
   is equal to true.
  execute the next instruction when the state of the third Cell
   is equal to false.
  return a Living Cell with alive as value.
  execute the next 3 instructions when the state of the first Cell
   is equal to true.
  execute the next 2 instructions when the state of the second Cell
   is equal to false.
```

```
    execute the next instruction when the state of the third Cell
     is equal to true.
    return a Living Cell with alive as value.
    execute the next 3 instructions when the state of the first Cell
     is equal to true.
    execute the next 2 instructions when the state of the second Cell
     is equal to false.
    execute the next instruction when the state of the third Cell
     is equal to false.
    return a Dead Cell with dead as value.
    execute the next 3 instructions when the state of the first Cell
     is equal to false.
    execute the next 2 instructions when the state of the second Cell
     is equal to true.
    execute the next instruction when the state of the third Cell
     is equal to true.
    return a Living Cell with alive as value.
    execute the next 3 instructions when the state of the first Cell
     is equal to false.
    execute the next 2 instructions when the state of the second Cell
     is equal to true.
    execute the next instruction when the state of the third Cell
     is equal to false.
    return a Living Cell with alive as value.
    execute the next 3 instructions when the state of the first Cell
     is equal to false.
    execute the next 2 instructions when the state of the second Cell
     is equal to false.
    execute the next instruction when the state of the third Cell
     is equal to true.
    return a Living Cell with alive as value.
    execute the next 3 instructions when the state of the first Cell
     is equal to false.
    execute the next 2 instructions when the state of the second Cell
     is equal to false.
    execute the next instruction when the state of the third Cell
     is equal to false.
    return a Dead Cell with dead as value.
    return null.
  verb format itself:
    tape are some Cells.
    repeat the next 4 instructions for each array of entry as symbol.
    execute the next instruction when symbol is equal to alive.
    a Living Cell with symbol as value; and add it to tape.
    execute the next instruction when symbol is equal to dead.
    a Dead Cell with symbol as value; and add it to tape.
  verb reformat cero as Character and uno as Character in itself:
    mark uno in entry; and replace alive into it.
    mark cero in entry; and replace dead into it.
noun Cell with plural as Cells:
  attribute value as a Character.
  overrided derived attribute string as a String:
    return the string of value.
  derived attribute state as a Boolean:
    execute the next instruction when this is a Living Cell.
    return true.
    execute the next instruction when this is a Dead Cell.
    return false.
    return null.
```

```
  overrided plural derived attribute string as a String:
    result is "".
    repeat the next instruction for each element of this as cell.
    add the string of cell to result.
    return result.
adjective Living.
adjective Dead.
```

Apéndice D

API básica de SN

En esta sección se presenta la API de SN, cabe destacar que esta información se generó de forma automática a partir de los archivos fuente que se cargaron con reflection y se le dio formato a todos los que son utilizables por el programador. Por ejemplo, un método de nombre *itself_ minus_ arg* en la clase *naturalistic.lang.Character*, en SN se emplea como *itself minus Number*, donde *itself* se reemplaza por la instancia ya sea un símbolo (*'c'*), una referencia indirecta (*the last Character*) o un identificador (*c is 'c'.*). Todo método en el archivo utilizable por SN *class* tiene un formato que incluye la palabra *itself*, opcionalmente la palabra arg por cada argumento que el método reciba y una preposición cuando se definan desde SN, esto para evitar definiciones ambiguas.

```
************************************************************
Adjectives: naturalistic.lang.Adjective
Verbs: 7
Declared Verbs: 8
Verb List:
************************************************************
Singular Noun: naturalistic.lang.Boolean
Inherits from: naturalistic.lang.Thing
Verbs: 23
Declared Verbs: 7
Verb List:
************************************************************
Plural Noun: naturalistic.lang.Booleans
Inherits from: naturalistic.lang.Things
Verbs: 39
Declared Verbs: 1
Verb List:
************************************************************
Adjectives: naturalistic.lang.Character
Implements Adjectives:
Adjective: naturalistic.lang.NumberProperty
Verbs: 52
Declared Verbs: 34
Verb List:
itself minus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself plus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself times naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
```

```
itself / naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself + naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself by naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
add naturalistic.lang.NumberProperty to itself returns: naturalistic.lang.NumberProperty
itself - naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself * naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
multiply naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
integer of itself returns: naturalistic.lang.Integer
itself % naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
divide naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
subtract naturalistic.lang.NumberProperty from itself returns: naturalistic.lang.NumberProperty
************************************************************
Adjectives: naturalistic.lang.Comparable
Implements Adjectives:
Adjective: naturalistic.lang.Adjective
Verbs: 15
Declared Verbs: 12
Verb List:
itself < naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself lesser or equal than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself lesser than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself !== naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself == naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself greater than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself distinct to naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself equal to naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself > naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself <= naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself greater or equal than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself >= naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
************************************************************
Adjectives: naturalistic.lang.Integer
Implements Adjectives:
Adjective: naturalistic.lang.NumberProperty
Verbs: 53
Declared Verbs: 35
Verb List:
itself minus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself plus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself times naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself / naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself + naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself by naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
add naturalistic.lang.NumberProperty to itself returns: naturalistic.lang.NumberProperty
itself - naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself * naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
multiply naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
itself % naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
divide naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
subtract naturalistic.lang.NumberProperty from itself returns: naturalistic.lang.NumberProperty
character of itself returns: naturalistic.lang.Character
real of itself returns: naturalistic.lang.Real
************************************************************
Singular Noun: naturalistic.lang.Number
Inherits from: naturalistic.lang.Thing
Implements Adjectives:
Adjective: naturalistic.lang.NumberProperty
Verbs: 51
Declared Verbs: 23
Verb List:
string of itself returns: naturalistic.lang.String
itself < naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself lesser or equal than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself lesser than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself !== naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
```

```
itself == naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself greater than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself distinct to naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself equal to naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself > naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself <= naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself greater or equal than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself >= naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
**********************************************************
Adjectives: naturalistic.lang.NumberProperty
Implements Adjectives:
Adjective: naturalistic.lang.Comparable
Verbs: 46
Declared Verbs: 43
Verb List:
itself minus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself plus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself times naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself / naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself < naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself + naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself by naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
add naturalistic.lang.NumberProperty to itself returns: naturalistic.lang.NumberProperty
itself - naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself * naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself lesser or equal than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself lesser than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
multiply naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
itself !== naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself == naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself % naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
divide naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
itself greater than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself distinct to naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself equal to naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself > naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
subtract naturalistic.lang.NumberProperty from itself returns: naturalistic.lang.NumberProperty
itself <= naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself greater or equal than naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
itself >= naturalistic.lang.Comparable returns: naturalistic.lang.Boolean
**********************************************************
Plural Noun: naturalistic.lang.Numbers
Inherits from: naturalistic.lang.Things
Verbs: 91
Declared Verbs: 54
Verb List:
itself minus naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself plus naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself times naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself / naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself < naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself + naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself by naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
add naturalistic.lang.NumberProperty to itself returns: naturalistic.lang.Numbers
itself - naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself * naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself lesser or equal than naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself lesser than naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
multiply naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.Numbers
itself !== naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself == naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself % naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
divide naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.Numbers
itself greater than naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
```

```
itself distinct to naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself equal to naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself > naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself <= naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself greater or equal than naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
itself >= naturalistic.lang.NumberProperty returns: naturalistic.lang.Numbers
tail of itself returns: naturalistic.lang.Numbers
length of itself returns: naturalistic.lang.Number
head of itself returns: naturalistic.lang.Number
pivot of itself returns: naturalistic.lang.Number
************************************************************
Adjectives: naturalistic.lang.Real
Implements Adjectives:
Adjective: naturalistic.lang.NumberProperty
Verbs: 51
Declared Verbs: 33
Verb List:
itself minus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself plus naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself times naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself / naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself + naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself by naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
add naturalistic.lang.NumberProperty to itself returns: naturalistic.lang.NumberProperty
itself - naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
itself * naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
multiply naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
integer of itself returns: naturalistic.lang.Integer
itself % naturalistic.lang.NumberProperty returns: naturalistic.lang.NumberProperty
divide naturalistic.lang.NumberProperty by itself returns: naturalistic.lang.NumberProperty
subtract naturalistic.lang.NumberProperty from itself returns: naturalistic.lang.NumberProperty
************************************************************
Singular Noun: naturalistic.lang.String
Inherits from: naturalistic.lang.Thing
Verbs: 46
Declared Verbs: 37
Verb List:
string of itself returns: naturalistic.lang.String
add naturalistic.lang.String to itself returns: naturalistic.lang.String
itself equal to naturalistic.lang.Thing returns: naturalistic.lang.Boolean
character of itself returns: naturalistic.lang.Number
length of itself returns: naturalistic.lang.Number
hash of itself returns: naturalistic.lang.Number
empty of itself returns: naturalistic.lang.Boolean
array of itself returns: naturalistic.lang.Numbers
index of itself returns: naturalistic.lang.Number
itself contains naturalistic.lang.String returns: naturalistic.lang.Boolean
compare naturalistic.lang.String with itself returns: naturalistic.lang.Number
search naturalistic.lang.Character in itself returns: naturalistic.lang.String
choose naturalistic.lang.Integer from itself returns: naturalistic.lang.String
concat naturalistic.lang.String with itself returns: naturalistic.lang.String
characters of itself returns: naturalistic.lang.Numbers
itself ends with naturalistic.lang.String returns: naturalistic.lang.Boolean
uppercase of itself returns: naturalistic.lang.String
itself starts with naturalistic.lang.String returns: naturalistic.lang.Boolean
itself splits with naturalistic.lang.String returns: naturalistic.lang.Strings
lowercase of itself returns: naturalistic.lang.String
************************************************************
Plural Noun: naturalistic.lang.Strings
Inherits from: naturalistic.lang.Things
Verbs: 72
Declared Verbs: 35
Verb List:
character of itself returns: naturalistic.lang.Numbers
length of itself returns: naturalistic.lang.Numbers
```

```
hash of itself returns: naturalistic.lang.Numbers
empty of itself returns: naturalistic.lang.Booleans
array of itself returns: naturalistic.lang.Things
index of itself returns: naturalistic.lang.Numbers
itself contains naturalistic.lang.String returns: naturalistic.lang.Booleans
compare naturalistic.lang.String with itself returns: naturalistic.lang.Numbers
choose naturalistic.lang.Integer from itself returns: naturalistic.lang.Strings
concat naturalistic.lang.String with itself returns: naturalistic.lang.Strings
itself ends with naturalistic.lang.String returns: naturalistic.lang.Booleans
uppercase of itself returns: naturalistic.lang.Strings
itself starts with naturalistic.lang.String returns: naturalistic.lang.Booleans
itself splits with naturalistic.lang.String returns: naturalistic.lang.Things
lowercase of itself returns: naturalistic.lang.Strings
**********************************************************
Singular Noun: naturalistic.lang.Thing
Inherits from: java.lang.Object
Implements Adjectives:
Adjective: naturalistic.lang.Adjective
Verbs: 16
Declared Verbs: 10
Verb List:
string of itself returns: naturalistic.lang.String
itself equal to returns: naturalistic.lang.Boolean
**********************************************************
Plural Noun: naturalistic.lang.Things
Inherits from: naturalistic.lang.Thing
Verbs: 38
Declared Verbs: 24
Verb List:
string of itself returns: naturalistic.lang.String
remove naturalistic.lang.Integer from itself returns: naturalistic.lang.Adjective
last of itself returns: naturalistic.lang.Adjective
add naturalistic.lang.Adjective to itself returns: naturalistic.lang.Things
element of itself returns: scala.collection.immutable.List
first of itself returns: naturalistic.lang.Adjective
reverse of itself returns: naturalistic.lang.Things
get naturalistic.lang.Integer from itself returns: naturalistic.lang.Adjective
```

Apéndice E

Instalación

En este apartado se describe el proceso de instalación del lenguaje SN.

E.1. Instalación de SN

1. Requisitos

   - Java SE Runtime Environment 1.8, que se encuentra disponible para su descarga en la siguiente dirección:
     - http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html
   - Compilador Scala 2.12.3 (no SBT), que se encuentra disponible para su descarga en la siguiente dirección:
     - https://scala-lang.org/download/2.12.3.html
   - AspectJ 1.8.10, que se encuentra disponible para su descarga en la siguiente dirección:
     - http://www.eclipse.org/aspectj/downloads.php

2. Proceso de instalación

   - Descomprimir el archivo SN.zip en la carpeta donde se desea instalar SN.

- Crear la variable de entorno SN_PATH y agregar las direcciones donde se ubican los archivos aspectjrt.jar y scala-library.jar (carpeta lib de sus respectivos compiladores).
- Agregar los compiladores scalac y ajc a la variable de entorno PATH o, agregar las direcciones donde se encuentran dichos compiladores (carpeta bin de cada uno).
- Agregar la dirección sn/bin a la variable de entorno PATH.

E.2. Compilación y ejecución de SN

SN trabaja con archivos cuya extensión es *.sn* por medio de los comandos *snc* y *sn*, el primero compila y genera los ejecutables y el segundo permite correrlos. La sintaxis para la compilación es la siguiente:

```
snc archivo.sn
```

Para ejecutar un programa se requiere de una abstracción main que generará un archivo con extensión *.class*, mismo que se ejecuta de la siguiente forma:

```
sn mainFile
```

Donde, *mainFile* es una abstracción en la forma de *main mainFile: ...* Nótese que los programas sólo se ejecutan por medio de un punto de entrada en forma de una abstracción main, cuya única función es servir de punto de acceso al programa. Otra cosa que se debe tomar en cuenta es que en el estado actual del compilador sólo se permite compilar un archivo a la vez y además, sólo permite trabajar con los archivos que se encuentren en la misma carpeta. Por último, dado que la gran mayoría de las abstracciones realizan composición durante la instanciación, es común que aparezcan avisos de AspectJ notificando que los cortes no se aplicaron.

E.3. Editor

Para ejecutar el editor, sólo se requiere del comando sn-editor, este abre una interfaz gráfica simple para crear, compilar y ejecutar programas escritos en SN.

```
sn-editor
```

Un ejemplo de la interfaz se observa en la figura 4.3.

Comunicación con expertos

En este apartado se presentan las opiniones de dos expertos internacionales a quienes se les contactó para que emitieran una opinión sobre el trabajo.

E.4. Henry Lieberman del Massachusetts Institute of Technology (MIT)

Dear Dr. Henry Lierberman

My name is Oscar Pulido Prieto, I?m a PhD Student from the Instituto Tecnológico de Orizaba in México; my thesis consists in defining a naturalistic model for general purpose programming. I write you because your experience with Metafor can be useful for my project. From the model that I propose, I?m developing a programming language that uses indirect references, types and noun phrases for describing instances; but also, the language defines abstractions like nouns or adjectives that are used for composition, classification and selection for a desired instance. I send you an attached document where I describe the model and an implementation example.

It would very useful for my research if you could receive me, or if you can?t do it, give me an opinion about my work by electronic means.

Sincerely, Oscar Pulido-Prieto.

–

Oscar,

Thanks for sending me this description of your work. I find it very interesting. As you know, I am very interested in the possibility of doing programming in a natural language like English or Spanish. I think natural language understanding has improved to the extent that this now may be possible.

There are two approaches. One is to redesign a formal, textual programming language to make it more like a natural language. That is the approach that you are taking. It is an approach that started with Cobol (though it is not popular today, for generations it was the standard language for business computing, so it was arguably a success). I think today one of the best examples is the game language Inform 7.

I think you identify some aspects of natural language, like deixis, that could be implemented in a computer language framework, bringing the language closer to natural

ones. This makes the program more readable. With this approach, though, the question is whether it is easier to write than a traditional program. An advantage of a conventional language is its fixed syntax, which at least makes it possible for the user to determine what is legal syntax. An open question is whether richer syntax, which entails a more complicated parser, is also easier for people. They still have to remember what is acceptable syntax and what not, and similarity to natural language might lull people into thinking the parser is more capable than it is. A key component is how this will work in the IDE – can you give the user good feedback when they get a syntax error? This is a question that needs to be decided by user testing, and I'd encourage you to try to test out your language on users to get feedback.

With our work on Metafor, we opted for the second approach, which is to try to accept natural language as it is. The tradeoff is that you can't be assured of fully understanding the input. The language is both readable as natural language and easily writable by the user. But the user may use constructs that the system does not understand, or that may be misunderstood. In Metafor, we positioned it as a code editor, assuming that the human user would read and correct any lack of understanding. In a more practical system, if I were designing it today, I'd try to engage in more interaction with the user about clarifying the user's intent through a natural language dialogue.

In any event, I'm happy to see you are working on this problem, and wish you the best of success!

Regards,

Henry Lieberman

E.5. Roman Knöll de la Technische Universitäd Darmstadt (TU Darmstadt)

Los comentarios en azul son cuestiones a las cuales el autor quiso responder con especial atención.

Hola Oscar,

Thank you for writing! I browsed through your document and it seems an interesting, straight-forward approach to me. May I ask, is your background programming language design or natural-language processing (or both)?

I didn?t quite understand what exactly you would expect from my side at the moment? You will be in Germany and wanted to meet and talk a bit? I think that should not be a problem to realize. Concerning my work, I like to keep you up to date (in case you would want that), so my current plan is as follows: finish programming on the prototype of Pegasus/n. It will cover also the examples that you mentioned in your essay like factorial function etc. then finishing official research project. Hopefully that within the coming year, i.e. until 2018 automn. I tell you only because after finishing I could send you my book/dissertation, where I described all base theory for natural(istic) programming from

the perspective of programming-language design. In the case that your background would be natural-language processing (which seems to be the case for the very few other people around the world that are at all interested in the field), that would complement very well.

I think, maybe after 2018 it would make sense to organize a meeting of all people around the globe that are active in this tiny field to exchange ideas and set up a research agenda. Unfortunately, at the moment I am focussed to much in finishing what I begun since I thought it would be one third the work it actually turned out to be. ;-)

If you like to be kept up to date, I would store your contact data.

Viele Grüße to beautiful México! :-)

Roman

–

Dear Roman

I really appreciate your answer and observations. I write you because I need the opinon of an expert in order to know if my work is well-focused. Sadly. My background is programming language design, so I start from the idea of raise a little the expressiveness level of formal languages in order to approach them to natural languages, but without leaving formalisms, avoiding ambiguity and keeping the abstraction level of languages like Java or Scala.

I wish your opinion not just about the prototype, but also of the conceptual model that I propose where I describe what I consider as mimum elements for a naturalistic language (is it complete or there is something that I lost?). Also I was hoping you could recieve me because research stays are desirable in my formation, but if you consider it unnecessary, I will be very grateful if you can give an opinion for mail when I need it because I'm working only with that I can infer from papers, but I do not have a real opinion from an expert in naturalistic programming.

I have received the opinion of an expert in programming languages, he told me he don't like the language and he wants something more mathematical because it's easiest describing formalisms. I presented him an extension mechanism to defining new embedded grammars to solve domain-specific problems, I'm still testing it, but it works. I send you an example to calculate the area of a triangle:

noun Triangle: attribute height is a Real Number. attribute base as a Real Number.

verb area from itself: A is base * height. (math) A = (base * height) / 2 A = base - 1 var z = A - 25 . return A.

In this example, I use the (unnatural) sentence "(math) ....ªnd this means: a grammar math is loaded and it deals with the enclosed instructions. I thought in this because in natural language documents, domain formalisms are placed using special marks like italic, bold or even sections like .ªlgorithm 1:". If you could give me an opinion about this, I will be very pleased.

Sincerely, Oscar.

–

Hallo Oscar,

Sadly. My background is programming language design, so I start from the idea of raise a little the expressiveness level of formal languages in order to approach them to natural languages

Yes, I forgot, that would be another approach, from my perspective an application of what I am doing now. I chose the radical path, rebuilding everything from Scratch.

but without leaving formalisms, avoiding ambiguity and keeping the abstraction level of languages like Java or Scala.

In the end any natural(astic) language has to offer such structures, for example a sophisticated module system. So you start where I like to be in a few years. ;-) Also complementing.

You start right now with your doctorate and try to specify your topic? What I am curious about: what is your personal ? or you got an assigned topic? ? motivation to create such a language? How come you think about this idea?

I wish your opinion not just about the prototype, but also of the conceptual model that I propose where I describe what I consider as mimum elements for a naturalistic language (is it complete or there is something that I lost?).

Well, the last thing I like to do is to demotivate you: I think there are more relevant elements but I do think you got the core ones. I worked several years only with those, intensely, so I think this would be far enough realistically for a doctorate. Mine, in my opinion, is a total exaggeration of that. I would keep to these elements if your goal is the doctorate. If you want to change the world, then figure out everything you can think of. I can?t actually give your any hints here because it does not make sense in my opinion to influence other scientists. Every one should rather come up with genuine ideas, I don?t think we have a lot of progress in current ?science? because of the flawed system (peer-review).

Also I was hoping you could recieve me because research stays are desirable in my formation

I do understand that, and in the past I did have students here but at the moment I can?t support them since I fully focus on finishing my programming work. In the future that might change but I can?t promise anything yet.

an opinion for mail when I need it because I'm working only with that I can infer from papers, but I do not have a real opinion from an expert in naturalistic programming.

I think you can write any time. Don?t expect too much though since I work with that topic already for 10 years, and I think ? without having done any comprehensive comparison the last two years ? my research is far ahead of everything that is there so far in that field. I don?t want to sound arrogant though, it is not meant like this, just I haven?t anything comparable (not what I published in the articles, the current system is far advanced). I would be happy though for any feedback or hint or link or whatever.

Just don?t let my sometimes harsh critique demotivate you. ;-) I think, as a scientist, you should be able to deal with that.

I like your approach of introducing natural(istic) elements into mainstream languages. In addition we don?t compete here at all. My expertise here is limited though because it is different mind set and different goal to some extent. I think the more precise your question would be you send to me, the more precise an answer I can send to you. Global questions like the one you gave me are hard to answer.

I have received the opinion of an expert in programming languages, he told me he don't like the language and he wants something more mathematical because it's easiest describing formalisms.

That?s why I developed another language, which I call ??? (the current interpreter is not functional since I can?t keep the web sites up to date at the moment but I made huge progress since then), you could read about the ?pattern language?, should be only, too. My approach is to combine natural language with formal languages (I am a mathematician myself, and I strongly see their point but any true mathematician works with both natural and formal language, just have a look at the books).

I presented him an extension mechanism to defining new embedded grammars to solve domain-specific problems

I developed something like this, as you will see in the article. Here, too, I tried to push the boundaries as far as possible. We will see if it works out entirely.

Your example would be something like in Pegasus/n:

?a tringle has a height and a base? (both known to be ?floating-point numbers?) ?the area of a triangle is (its height) × (its base) / 2?

Alternatively:

?the area of a triangle with height ?h? and base ?b? is h × b / 2?

I thought in this because in natural language documents, domain formalisms are placed using special marks like italic, bold or even sections like .ªlgorithm 1:". To me that seems acceptable for contemporary languages like Java. I don?t know how to extend them since I have never dealt with that. I used to build languages from scratch as I like them to be as pure as possible.

I hope I could help you some.

Viele Grüße!

–

Hola Roman

You start right now with your doctorate and try to specify your topic? What I am curious about: what is your personal ? or you got an assigned topic? ? motivation to create such a language? How come you think about this idea?

I'm ending my sixth semester of eight, the idea of this project come from my adviser (we work together since my bachelor project, then we work in a modelling tool for my master thesis). We share the taste for programming languages, so he proposed the idea

based on the paper of Cristina Videira Lopes and I accept it because I always have the idea of create someting like that. In fact, the focus is the model and the language is a mean to demonstrate it.

Well, the last thing I like to do is to demotivate you: I think there are more relevant elements but I do think you got the core ones. I worked several years only with those, intensely, so I think this would be far enough realistically for a doctorate.

Well, in fact you are motivating me with this and I thank you so much, I will be working in refining it.

I would keep to these elements if your goal is the doctorate. If you want to change the world, then figure out everything you can think of.

The goal is the doctorate, but also I want to change the world, but one step at a time. I believe I need more experience.

I think you can write any time. Don?t expect too much though since I work with that topic already for 10 years, and I think ? without having done any comprehensive comparison the last two years ? my research is far ahead of everything that is there so far in that field. I don?t want to sound arrogant though, it is not meant like this, just I haven?t anything comparable (not what I published in the articles, the current system is far advanced). I would be happy though for any feedback or hint or link or whatever. Just don?t let my sometimes harsh critique demotivate you. ;-) I think, as a scientist, you should be able to deal with that.

I understand and I totally agreed with you, I am just begining and your "harsh critiqueïs something for what I am prepared.

My approach is to combine natural language with formal languages.

Mine too, but I think my perspective is different. Instead using natural languages, I propose a model for bring natural language expressiveness to mainstream languages, as you mentioned.

Sincerely, Oscar.

Referencias

Apple Inc. (2016). *Applescript language guide.* Descargado 2016-03-01, de https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html

Arnold, K. C., y Lieberman, H. (2010a). Embracing ambiguity. En *Proceedings of the fse/sdp workshop on future of software engineering research* (pp. 1–6). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1882362.1882364 doi: 10.1145/1882362.1882364

Arnold, K. C., y Lieberman, H. (2010b). Managing ambiguity in programming by finding unambiguous examples. En *Proceedings of the acm international conference on object oriented programming systems languages and applications* (pp. 877–884). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1869459.1869531 doi: 10.1145/1869459.1869531

Aubit-4gl team. (2001). *X-4gl reference manual.* Descargado 2016-03-01, de http://aubit4gl.sourceforge.net/aubit4gldoc/4glreference/pages/4GLREFINFORMIX4GL_Reference_Help.htm

Ballard, B. W. (1984). The syntax and semantics of user-defined modifiers in a transportable natural language processor. En *Proceedings of the 10th international conference on computational linguistics and 22nd annual meeting on association for computational linguistics* (pp. 52–56). Stroudsburg, PA, USA: Association for Computational Linguistics. Descargado de http://dx.doi.org/10.3115/980491.980504 doi: 10.3115/980491.980504

Ballard, B. W., y Lusth, J. C. (1983). An english-language processing system that "learns.ªbout new domains. En *Proceedings of the may 16-19, 1983, national computer conference* (pp. 39–46). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1500676.1500682 doi: 10.1145/1500676.1500682

Ballard, B. W., y Stumberger, D. E. (1986). Semantic acquisition in teli: A transportable, user-customized natural language processor. En *Proceedings of the 24th annual meeting on association for computational linguistics* (pp. 20–29). Stroudsburg, PA, USA: Association for Computational Linguistics. Descargado de http://dx.doi.org/10.3115/981131.981136 doi: 10.3115/981131.981136

Bartle, R. (2003). *Designing virtual worlds.* New Riders Games.

Begel, A., y Graham, S. L. (2005). Spoken programs. En *Proceedings of the 2005 ieee symposium on visual languages and human-centric computing* (pp. 99–106). Washington, DC, USA: IEEE Computer Society. Descargado de http://dx.doi.org/10.1109/VLHCC.2005.58 doi: 10.1109/VLHCC.2005.58

Begel, A., y Graham, S. L. (2006a). An assessment of a speech-based programming environment. En *Proceedings of the visual languages and human-centric computing*

(pp. 116–120). Washington, DC, USA: IEEE Computer Society. Descargado de http://dx.doi.org/10.1109/VLHCC.2006.9 doi: 10.1109/VLHCC.2006.9

Begel, A., y Graham, S. L. (2006b, agosto). Xglr: An algorithm for ambiguity in programming languages. *Sci. Comput. Program.*, *61*(3), 211–227. Descargado de http://dx.doi.org/10.1016/j.scico.2006.04.003 doi: 10.1016/j.scico.2006.04.003

Berkholz, D. (2013). *Programming languages ranked by expressiveness.* Descargado 2018-11-15, de https://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness

Biermann, A. W., y Ballard, B. W. (1980, abril). Toward natural language computation. *Comput. Linguist.*, *6*(2), 71–86. Descargado de http://dl.acm.org/citation.cfm?id=972439.972440

Bolshakov, I. A., y Gelbukh, A. (2004). *Computational linguistics models, resources, applications.* Ciencia de la computación.

Booch, G., Maksimchuk, R. A., Engle, M. W., J. Young, B., Conallen, J., y Kelli, A. H. (2007). *Object-oriented analysis and design with applications* (3rd ed.). Boston, MA, USA: Addison Wesley Longman.

Bruckman, A. (1998, enero). Community support for constructionist learning. *Comput. Supported Coop. Work*, *7*(1-2), 47–86. Descargado de http://dx.doi.org/10.1023/A:1008684120893 doi: 10.1023/A:1008684120893

Bruckman, A., y Edwards, E. (1999). Should we leverage natural-language knowledge? an analysis of user errors in a natural-language-style programming language. En *Proceedings of the sigchi conference on human factors in computing systems* (pp. 207–214). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/302979.303040 doi: 10.1145/302979.303040

Bruckman, A. S. (1997). *Moose crossing: Construction, community, and learning in a networked virtual world for kids* (Tesis Doctoral no publicada). Cambridge, MA, USA. (AAI0598541)

Carlos, C. S. (2011). Natural language programming using class sequential rules. En *Ijcnlp* (pp. 237–245).

Chamberlin, D. D., y Boyce, R. F. (1974). Sequel: A structured english query language. En *Proceedings of the 1974 acm sigfidet (now sigmod) workshop on data description, access and control* (pp. 249–264). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/800296.811515 doi: 10.1145/800296.811515

Chitchyan, R., Greenwood, P., Sampaio, A., Rashid, A., Garcia, A., y Fernandes da Silva, L. (2009). Semantic vs. syntactic compositions in aspect-oriented requirements engineering: An empirical study. En *Proceedings of the 8th acm international conference on aspect-oriented software development* (pp. 149–160). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1509239.1509260 doi: 10.1145/1509239.1509260

Chitchyan, R., Rashid, A., Rayson, P., y Waters, R. (2007). Semantics-based composition for aspect-oriented requirements engineering. En *Proceedings of the 6th international conference on aspect-oriented software development* (pp. 36–48). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1218563.1218569 doi: 10.1145/1218563.1218569

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, *2*(3), 113–124.

Chong, S., y Pucella, R. (2004). A framework for creating natural language user interfaces for action-based applications. *CoRR*, *abs/cs/0412065*. Descargado de http://arxiv.org/abs/cs/0412065

Clark, P., Harrison, P., Jenkins, T., Thompson, J., y Wojcik, R. (2005). Acquiring and using world knowledge using a restricted subset of english. En *Proceedings of the eighteenth international florida artificial intelligence research society conference (flairs 2005)* (pp. 506–511). AAAI Press.

Clark, P., Murray, W. R., Harrison, P., y Thompson, J. (2010). Controlled natural language: Workshop on controlled natural language, cnl 2009, marettimo island, italy, june 8-10, 2009. revised papers. En N. E. Fuchs (Ed.), (pp. 65–81). Berlin, Heidelberg: Springer Berlin Heidelberg. Descargado de http://dx.doi.org/10.1007/978-3-642-14418-9_5 doi: 10.1007/978-3-642-14418-9_5

Cook, M. (2004). Universality in elementary cellular automata. *Complex systems*, *15*(1), 1–40.

Coombe, C., y Davidson, P. (2015). Constructing questionnaires. *The Cambridge Guide to Research in Language Teaching and Learning*, *217*.

Cozzie, A., Finnicum, M., y King, S. T. (2011). Macho: Programming with man pages. En *Proceedings of the 13th usenix conference on hot topics in operating systems* (pp. 7–7). Berkeley, CA, USA: USENIX Association. Descargado de http://dl.acm.org/citation.cfm?id=1991596.1991606

Cozzie, A., y King, S. T. (2011). *Macho ii: Even more macho* (Inf. Téc.). Descargado de https://pdfs.semanticscholar.org/367b/0b950dd76177074c9c86297ab7c188bfb2b9.pdf

Cozzie, A., y King, S. T. (2012). *Macho: Writing programs with natural language and examples* (Inf. Téc.). University of Illinois at Urbana-Champaign.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., ... Roy, S. (2016). Program synthesis using natural language. En *Proceedings of the 38th international conference on software engineering* (pp. 345–356). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/2884781.2884786 doi: 10.1145/2884781.2884786

De Saussure, F. (1916). Nature of the linguistic sign. *Course in general linguistics*, 65–70.

Dyer, R., Rajan, H., y Cai, Y. (2012). An exploratory study of the design impact of language features for aspect-oriented interfaces. En *Proceedings of the 11th annual international conference on aspect-oriented software development* (pp. 143–154). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/2162049.2162067 doi: 10.1145/2162049.2162067

Eisenstadt, M. (1993). Why HyperTalk Debugging is More Painful than it ought to be. En J. Alty, D. Diaper, y S. P. Guest (Eds.), *People and computers viii.* Cambridge, UK: Cambridge University Press.

Flesch, R. (1948). A new readability yardstick. *Journal of applied psychology*, *32*(3), 221.

Fowler, M. (2005). *Language workbenches: The killer-app for domain specific languages?* Descargado de http://www.cime.cl/archivos/ILI253/8870_cl2-MartinFowler-Language-Workbench-DSL.pdf

Fuchs, N., y cols. (1999). Attempto controlled english language manual version 3.0 [Manual de software informático]. The address of the publisher.

Fuchs, N. E. (2018). Understanding texts in attempto controlled english. En *Controlled natural language: Proceedings of the sixth international workshop, cnl 2018, maynooth,*

co. kildare, ireland, august 27-28, 2018 (Vol. 304, p. 75).

Fuchs, N. E., Kaljurand, K., y Kuhn, T. (2008). Reasoning web. En C. Baroglio, P. A. Bonatti, J. Maluszyński, M. Marchiori, A. Polleres, y S. Schaffert (Eds.), (pp. 104–124). Berlin, Heidelberg: Springer-Verlag. Descargado de http://dx.doi.org/10.1007/978-3-540-85658-0_3 doi: 10.1007/978-3-540-85658-0_3

Fuchs, N. E., y Schwitter, R. (1996). Attempto controlled english (ACE). *CoRR*, *cmp-lg/9603003*.

Gaintzarain, J., y Lucio, P. (2009). A new approach to temporal logic programming. En *Proceedings of the 9thspanish conference on programming and languages (prole'09)* (pp. 341–350). Descargado de http://www.sistedes.es/ficheros/actas-conferencias/PROLE/2009.pdf

Gordon, B. M., y Luger, G. F. (2012, Nov). English for spoken programming. En *Soft computing and intelligent systems (scis) and 13th international symposium on advanced intelligent systems (isis), 2012 joint 6th international conference on* (p. 16-20). doi: 10.1109/SCIS-ISIS.2012.6505414

Gregory, F. (1993). Cause, effect, efficiency and soft systems models. *Journal of the Operational Research Society*, *44*(4), 333-344. Descargado de https://doi.org/10.1057/jors.1993.63 doi: 10.1057/jors.1993.63

Gulwani, S., y Marron, M. (2014). Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. En *Proceedings of the 2014 acm sigmod international conference on management of data* (pp. 803–814). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/2588555.2612177 doi: 10.1145/2588555.2612177

Gunning, R. (1969). The fog index after twenty years. *Journal of Business Communication*, *6*(2), 3–13.

Gybels, K., y Brichau, J. (2003). Arranging language features for more robust pattern-based crosscuts. En *Proceedings of the 2nd international conference on aspect-oriented software development* (pp. 60–69). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/643603.643610 doi: 10.1145/643603.643610

Halstead, M. H. (1977). *Elements of software science (operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc.

Heering, J., y Mernik, M. (2002, Jan). Domain-specific languages for software engineering. En *Proceedings of the 35th annual hawaii international conference on system sciences (hicss-35)* (p. 3649-3650). doi: 10.1109/HICSS.2002.994484

Heidorn, G. E. (1973). An interactive simulation programming system which converses in english. En *Proceedings of the 6th conference on winter simulation* (pp. 781–794). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/800293.811628 doi: 10.1145/800293.811628

Hennig, D., Schummer, R., Slater, J., Granor, T. E., y Feltman, T. (2005). *What's new in nine: Visual foxpro's latest hits*. Hentzenwerke Publishing.

Hirschfeld, R., Costanza, P., y Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, *7*(3), 125–151.

Hsiao, M. S. (2018). Automated program synthesis from object-oriented natural language for computer games. En *Controlled natural language-proceedings of the sixth international workshop, cnl 2018, maynooth, co. kildare, ireland, august 27-28* (pp.

71–74).

I. Androutsopoulos, G. D. R., y Thanisch, P. (1995). Natural language interfaces to databases - an introduction. *In Natural Language Engineering*, *1*, 29–81.

Kamina, T., Aotani, T., y Masuhara, H. (2010). Designing event-based context transition in context-oriented programming. En *Proceedings of the 2nd international workshop on context-oriented programming* (pp. 2:1–2:6). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1930021.1930023 doi: 10.1145/1930021.1930023

Kemper, A., Kossmann, D., y Zeller, B. (1999). Performance tuning for sap r/3. *IEEE Data Eng. Bull.*, *22*(2), 32-39.

Kiczales, G., y Hilsdale, E. (2001). Aspect-oriented programming. En *Proceedings of the 8th european software engineering conference held jointly with 9th acm sigsoft international symposium on foundations of software engineering* (pp. 313–). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/503209.503260 doi: 10.1145/503209.503260

Kim, W. (1982, septiembre). On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, *7*(3), 443–469. Descargado de http://doi.acm.org/10.1145/319732.319745 doi: 10.1145/319732.319745

Kincaid, J. P., Fishburne Jr, R. P., Rogers, R. L., y Chissom, B. S. (1975). Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel.

Knöll, R., Gasiunas, V., y Mezini, M. (2011). Naturalistic types. En *Proceedings of the 10th sigplan symposium on new ideas, new paradigms, and reflections on programming and software* (pp. 33–48). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/2048237.2048243 doi: 10.1145/2048237.2048243

Knöll, R., y Mezini, M. (2006). Pegasus: First steps toward a naturalistic programming language. En *Companion to the 21st acm sigplan symposium on object-oriented programming systems, languages, and applications* (pp. 542–559). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1176617.1176628 doi: 10.1145/1176617.1176628

Kuhn, T. (2014, marzo). A survey and classification of controlled natural languages. *Comput. Linguist.*, *40*(1), 121–170. Descargado de http://dx.doi.org/10.1162/COLI_a_00168 doi: 10.1162/COLI_a_00168

Kuhn, T., y Bergel, A. (2014, diciembre). Verifiable source code documentation in controlled natural language. *Sci. Comput. Program.*, *96*(P1), 121–140. Descargado de http://dx.doi.org/10.1016/j.scico.2014.01.002 doi: 10.1016/j.scico.2014.01.002

Kung, C. H., y Sölvberg, A. (1986). Activity modeling and behavior modeling. En *Proc. of the ifip wg 8.1 working conference on information systems design methodologies: Improving the practice* (pp. 145–171). Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co. Descargado de http://dl.acm.org/citation.cfm?id=20143.20149

Laird, P., y Barrett, S. (2009). Towards context sensitive domain specific languages. En *Proceedings of the 1st international workshop on context-aware middleware and services: Affiliated with the 4th international conference on communication system software and middleware (comsware 2009)* (pp. 31–36). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1554233.1554241 doi:

10.1145/1554233.1554241
Lämmel, R. (1998, september). Object-oriented cobol: Concepts & implementation. *COBOL Unleashed. Macmillan Computer Publishing*, *44* .
Lämmel, R., y De Schutter, K. (2005). What does aspect-oriented programming mean to cobol? En *Proceedings of the 4th international conference on aspect-oriented software development* (pp. 99–110). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1052898.1052907 doi: 10.1145/1052898.1052907
Landhäußer, M., Hey, T., y Tichy, W. F. (2014). Deriving time lines from texts. En *Proceedings of the 3rd international workshop on realizing artificial intelligence synergies in software engineering* (pp. 45–51). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/2593801.2593809 doi: 10.1145/2593801.2593809
Landhäußer, M., y Hug, R. (2015). Text understanding for programming in natural language: Control structures. En *Proceedings of the fourth international workshop on realizing artificial intelligence synergies in software engineering* (pp. 7–12). Piscataway, NJ, USA: IEEE Press. Descargado de http://dl.acm.org/citation.cfm?id=2820668.2820671
Landhäußer, M., Weigelt, S., y Tichy, W. F. (2016). Nlci: a natural language command interpreter. *Automated Software Engineering*, 1–23. Descargado de http://dx.doi.org/10.1007/s10515-016-0202-1 doi: 10.1007/s10515-016-0202-1
Le, V., Gulwani, S., y Su, Z. (2013). Smartsynth: Synthesizing smartphone automation scripts from natural language. En *Proceeding of the 11th annual international conference on mobile systems, applications, and services* (pp. 193–206). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/2462456.2464443 doi: 10.1145/2462456.2464443
Li, A. (2013). *Handbook of sas data step programming*. CRC Press.
Liblit, B., Begel, A., y Sweetser, E. (2006). Cognitive perspectives on the role of naming in computer programs. En *Annual psychology of programming workshop.*
Lieberherr, K. J. (1995). *Adaptive object-oriented software: The demeter method with propagation patterns* (1st ed.). Boston, MA, USA: PWS Publishing Co.
Lieberman, H. (2006). The continuing quest for abstraction. En *Proceedings of the 20th european conference on object-oriented programming* (pp. 192–197). Berlin, Heidelberg: Springer-Verlag. Descargado de http://dx.doi.org/10.1007/11785477_12 doi: 10.1007/11785477_12
Lieberman, H., y Ahmad, M. (2010). Knowing what you're talking about: Natural language programming of a multi-player online game. *No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann.*
Likert, R. (1932). A technique for the measurement of attitudes. *Archives of psychology.*
Little, G., y Miller, R. C. (2006). Translating keyword commands into executable code. En *Proceedings of the 19th annual acm symposium on user interface software and technology* (pp. 135–144). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1166253.1166275 doi: 10.1145/1166253.1166275
Liu, H., y Lieberman, H. (2005a). Metafor: Visualizing stories as code. En *Proceedings of the 10th international conference on intelligent user interfaces* (pp. 305–307). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1040830.1040908 doi: 10.1145/1040830.1040908
Liu, H., y Lieberman, H. (2005b). Programmatic semantics for natural language interfaces.

En *Chi '05 extended abstracts on human factors in computing systems* (pp. 1597–1600). New York, NY, USA: ACM. Descargado de http://doi.acm.org/10.1145/1056808.1056975 doi: 10.1145/1056808.1056975

LiveCode Ltd. (2016). *Livecode.* Descargado 2016-03-01, de https://livecode.com/products/livecode-platform/livecode-in-education/

Lobeck, A. (2007). Ellipsis in dp. En *The blackwell companion to syntax* (pp. 145–173). Blackwell Publishing. Descargado de http://dx.doi.org/10.1002/9780470996591.ch22 doi: 10.1002/9780470996591.ch22

Lohmeier, S. (2011). *Shaping statically resolved indirect anaphora for naturalistic programming* (Bachelor's Thesis). Descargado de http://www.monochromata.de/bachelor_thesis/

Lopes, C. V., Dourish, P., Lorenz, D. H., y Lieberherr, K. (2003, diciembre). Beyond aop: Toward naturalistic programming. *SIGPLAN Not.*, *38*(12), 34–43. Descargado de http://doi.acm.org/10.1145/966051.966058 doi: 10.1145/966051.966058

Mai, P. X., Pastore, F., Goknil, A., y Briand, L. C. (2019). Mcp: A security testing tool driven by requirements. En *Proceedings of the 41st international conference on software engineering: Companion proceedings* (pp. 55–58). Piscataway, NJ, USA: IEEE Press. Descargado de https://doi.org/10.1109/ICSE-Companion.2019.00037 doi: 10.1109/ICSE-Companion.2019.00037

Mai, X. P., Pastore, F., Göknil, A., y Briand, L. (2018). A natural language programming approach for requirements-based security testing. *A Natural Language Programming Approach for Requirements-based Security Testing*.

Manshadi, M., Gildea, D., y Allen, J. (2013). Integrating programming by example and natural language programming. En *Proceedings of the twenty-seventh aaai conference on artificial intelligence* (pp. 661–667). AAAI Press. Descargado de http://dl.acm.org/citation.cfm?id=2891460.2891552

McCraken, D. D. (1980). *A guide to nomad for applications development.* Addison-Wesley Publishing Company.

Mefteh, M., Ben Hamadou, A., y Knöll, R. (2012). Ara_pegasus: A new framework for programming using the arabic natural language. En *International conference on computing and information technology (march, 2012)* (pp. 468–473).

Mefteh, M., Bouassida, N., y Ben-Abdallah, H. (2018). Towards naturalistic programming: mapping language-independent requirements to constrained language specifications. *Science of Computer Programming*.

Mernik, M., y Žumer, V. (2001). Domain-specific languages for software engineering. En *Proceedings of the 34th annual hawaii international conference on system sciences (hicss-34)-volume 9 - volume 9* (pp. 9071–). Washington, DC, USA: IEEE Computer Society. Descargado de http://dl.acm.org/citation.cfm?id=820738.820802

Mihalcea, R., Liu, H., y Lieberman, H. (2006). Nlp (natural language processing) for nlp (natural language programming). En *Proceedings of the 7th international conference on computational linguistics and intelligent text processing* (pp. 319–330). Berlin, Heidelberg: Springer-Verlag. Descargado de http://dx.doi.org/10.1007/11671299_34 doi: 10.1007/11671299_34

Mylopoulos, J. (1992). Conceptual modelling and telos. *Conceptual Modelling, Databases, and CASE: an Integrated View of Information System Development, New York: John Wiley & Sons*, 49–68.

Naithani, D., y cols. (2011). Guidelines for developing a robust web survey. *Advances in Information Technology and Management*, *1*(1), 20–23.

Nihalani, N., Silakari, S., y Motwani, M. (2011). *Natural language interface for database: A brief review.* IJCSI.

Nunez, A., y Gasiunas, V. (2009). Ecaesarj user's guide.

O'Brien, S. (2003). Controlling controlled english: an analysis of several controlled language rule sets. *Proceedings of EAMT-CLAW*, *3*, 105–114.

Ogden, C. K., Richards, I. A., Ranulf, S., y Cassirer, E. (1923). *The meaning of meaning. a study of the influence of language upon thought and of the science of symbolism.* JSTOR.

O'Gorman, J. (2010). *The aubit4gl manual.* Descargado 2017-04-23, de `http://www.og.co.nz/aubit4gl/html/index.html`

Orgun, M. A., y Ma, W. (1994). An overview of temporal and modal logic programming. En *Proceedings of the first international conference on temporal logic* (pp. 445–479). London, UK, UK: Springer-Verlag. Descargado de `http://dl.acm.org/citation.cfm?id=645548.659010`

Ostermann, K., Mezini, M., y Bockisch, C. (2005). Expressive pointcuts for increased modularity. En *Proceedings of the 19th european conference on object-oriented programming* (pp. 214–240). Berlin, Heidelberg: Springer-Verlag. Descargado de `http://dx.doi.org/10.1007/11531142_10` doi: 10.1007/11531142_10

Pane, J. F., Myers, B. A., y Miller, L. B. (2002). Using hci techniques to design a more usable programming system. En *Proceedings of the ieee 2002 symposia on human centric computing languages and environments (hcc'02)* (pp. 198–). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dl.acm.org/citation.cfm?id=795687.797801`

Pane, J. F., Ratanamahatana, C. A., y Myers, B. A. (2001, febrero). Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comput. Stud.*, *54*(2), 237–264. Descargado de `http://dx.doi.org/10.1006/ijhc.2000.0410` doi: 10.1006/ijhc.2000.0410

Pinter, L. (2004). *Visual foxpro to visual basic .net.* Pearson Education.

Press, O. U. (2016). *Stats and analysis.* Descargado de `http://www.oxforddictionaries.com/`

Price, D., Rilofff, E., Zachary, J., y Harvey, B. (2000). Naturaljava: A natural language interface for programming in java. En *Proceedings of the 5th international conference on intelligent user interfaces* (pp. 207–211). New York, NY, USA: ACM. Descargado de `http://doi.acm.org/10.1145/325737.325845` doi: 10.1145/325737.325845

Qiu, L., yen Kan, M., y seng Chua, T. (2004). A public reference implementation of the rap anaphora resolution algorithm. En *Proceedings of the fourth international conference on language resources and evaluation (lrec 2004)* (pp. 291–294).

Rebernak, D., Mernik, M., Wu, H., y Gray, J. (2009, June). Domain-specific aspect languages for modularizing crosscutting concerns in grammars. *IET Software*, *3*(3), 184-200. doi: 10.1049/iet-sen.2007.0114

Sadd, J. (2006). *Openedge development: Progress 4gl handbook.* Progress Software Corporation.

Sammet, J. E. (1966, marzo). The use of english as a programming language. *Commun. ACM*, *9*(3), 228–230. Descargado de `http://doi.acm.org/10.1145/365230.365274` doi: 10.1145/365230.365274

SAP-AG. (2014). *Abap - keyword documentation.* Descargado 2016-03-01, de http://help.sap.com/abapdocu_740/en

Schriber, T. J. (1990). *Simulation using gpss.* Melbourne, FL, USA: Krieger Publishing Co., Inc.

Simonyi, C., Christerson, M., y Clifford, S. (2006, octubre). Intentional software. *SIGPLAN Not.*, *41*(10), 451–464. Descargado de http://doi.acm.org/10.1145/1167515.1167511 doi: 10.1145/1167515.1167511

Siobhan, C., y Elisa, B. (2005). *Aspect-oriented analysis and design: The theme approach.* Addison Wesley.

The R Foundation. (2016). *The r project for statistical computing.* Descargado 2016-03-01, de https://www.r-project.org/

Thummalapenta, S., Sinha, S., Singhania, N., y Chandra, S. (2012). Automating test automation. En *Proceedings of the 34th international conference on software engineering* (pp. 881–891). Piscataway, NJ, USA: IEEE Press. Descargado de http://dl.acm.org/citation.cfm?id=2337223.2337327

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. En (Vol. 2, pp. 230–265). Wiley Online Library.

University, S. (2014). *Alice.* Descargado 2016-03-01, de http://www.ps.uni-saarland.de/alice/

Vadas, D., y Curran, J. R. (2005). Programming with unrestricted natural language. En *Proceedings of the australasian language technology workshop (2005)* (pp. 191–199).

Van Brummelen, J. (2019, Oct). Conversational agents to democratize artificial intelligence. En *2019 ieee symposium on visual languages and human-centric computing (vl/hcc)* (p. 239-240). doi: 10.1109/VLHCC.2019.8818805

van Deursen, A., Klint, P., y Visser, J. (2000, junio). Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, *35*(6), 26–36. Descargado de http://doi.acm.org/10.1145/352029.352035 doi: 10.1145/352029.352035

Van Roy, P., y Haridi, S. (2004). *Concepts, techniques, and models of computer programming* (1st ed.). The MIT Press.

Walonick, D. S. (2013). *Survival statistics* (6.ª ed.). StatPac Incorporated.

Wexelblat, R. L. (Ed.). (1981). *History of programming languages i.* New York, NY, USA: ACM.

Wheeler, K. (2004). *Hypertalk: The language for the rest of us.*

Wolfram, S. (2002). *A new kind of science* (Vol. 5). Wolfram media Champaign, IL.