Representación semántica de lenguaje natural en el dominio de fórmulas lógicas

Diego Piloni

Directores: Miguel Pagano y Demetrio Vilela

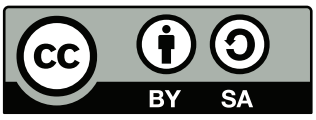
Facultad de Matemática, Astronomía, Física y Computación

Universidad Nacional de Córdoba

Tesis con el objetivo de conseguir el título

Licenciatura en Ciencias de la Computación

2018



Agradecimientos

A Migue y Deme, directores de este trabajo, quienes me acompañaron y guiaron a lo largo de todo el desarrollo de la Tesis. Juntos transitamos un camino de mucha incertidumbre y su conocimiento y buena voluntad fueron imprescindibles para que haya concluido este trabajo.

A Franco y Martín, quienes aceptaron formar parte del tribunal y corregir este trabajo en muy poco tiempo.

A todos los compañeros y amigos con los que compartí excelentes momentos en los años de cursada en FaMAF. Particularmente gracias Ger, Agu, Karen, Edgar, Tincho, Fran, Juan y Fede.

A los amigos que están desde antes o que conocí fuera de la facu, son una parte fundamental de mi día a día. Particularmente gracias Juan, Nahuel, Fede y Dani.

A mi Mamá y Papá, que han hecho todo lo posible para que pudiera comenzar y terminar esta carrera.

¡Gracias!

Resumen

Este trabajo final parte del supuesto de que el primer desafío que se encuentra en el aprendizaje de la lógica está dado por la barrera lingüística impuesta por el lenguaje simbólico y artificial de la misma. Consecuentemente parece razonable comenzar un curso de lógica con la traducción de sentencias expresadas en lenguaje natural (digamos español) a su correspondiente formulación simbólica en la lógica en cuestión (proposicional, de primer orden, etc.). Las dificultades de esta traducción vienen dadas, entre otras, por la ambigüedad de la sentencia en lenguaje natural y por la puntillosidad con la que se deben construir las fórmulas lógicas.

El objetivo de este proyecto es extender el programa Sat [5] con un módulo que permita a estudiantes la exploración de la traducción de lenguaje natural a fórmulas de lógica de primer orden. De esta manera, la/el estudiante puede ir familiarizándose con la escritura formal a través de ejemplos que ella/el construye. Esta traducción se basará teóricamente en la formalización de lenguaje natural utilizando teoría de tipos propuesta por Aarne Ranta [13] y para la implementación se utilizará la herramienta Grammatical Framework [12].

Índice general

Capítulo 1

Introducción

1.1 Gramáticas y estadística en procesamiento de lenguaje

¿Cómo podemos hacer que las computadoras procesen lenguaje? Existen dos tipos de respuestas a esta pregunta. Una respuesta es que la computadora trabaje con una gramática del lenguaje, esto es, un conjunto de reglas para analizar y producir texto. Otra respuesta, mucho más popular en los últimos años, es que la computadora aprenda al analizar datos. En el caso del lenguaje natural, la entrada debe ser texto del cual se buscará extraer información para construir modelos estadísticos o aplicar técnicas de machine learning.

Estas respuestas dividen a la comunidad de procesamiento de lenguaje natural en dos campos, simbólico y estadístico, respectivamente. Sin embargo, estas propuestas no son mutuamente excluyentes. Existe la posibilidad de definir gramáticas probabilísticas, las cuales suelen ser construidas bajo un enfoque estadístico.

El enfoque estadístico está dominando actualmente tanto la investigación como la aplicación práctica del procesamiento de lenguaje natural. En áreas como búsqueda y recuperación de información, traducción automática y reconocimiento del habla, este enfoque ha producido aplicaciones realmente útiles, capaces de manipular el lenguaje natural que las personas usan diariamente.

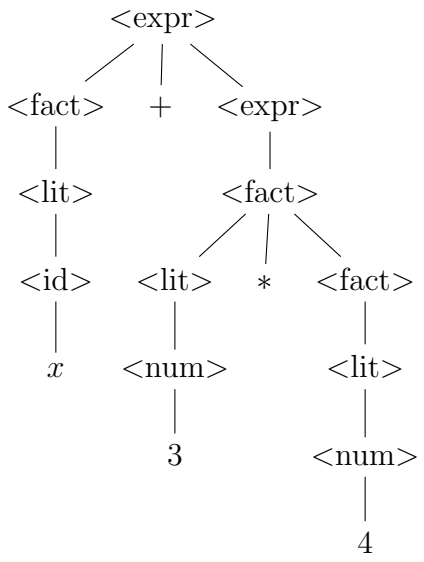
La aproximación con gramáticas suele tener un objetivo más teórico y un alcance más limitado. A pesar de ello, las gramáticas han sido un caso muy exitoso en la representación de lenguajes formales. Un ejemplo claro es el de los lenguajes de programación. Estos son analizados por una computadora a través de un compilador, el cual es un programa basado en una gramática del lenguaje. Sin embargo, existe una diferencia fundamental entre lenguajes formales y lenguajes naturales: Para los

primeros, existen sistemas de reglas completos, sin embargo, para los segundos este no es el caso. De hecho, los lenguajes de programación están definidos a partir de su gramática, es decir, no existirían sin ella.

Como ejemplo veamos una gramática simple que define un lenguaje de expresiones aritméticas, común en la mayoría de los lenguajes de programación.

```
<expr> ::= <fact> + <expr> | <fact>
<fact> ::= <lit> * <fact> | <lit>
<lit> ::= <id> | <num> | (<expr>)
```

En esta gramática la expresión $x+3*4$ se analiza con el siguiente árbol sintáctico:



Para los lenguajes naturales, en cambio, las gramáticas son teorías formadas a partir de observar un sistema ya existente. Capturar cada fenómeno presente de un lenguaje natural con un sistema de reglas a partir de la observación parece ser una tarea difícil, sino imposible. Una cita del lingüista Edward Sapir en la cual señala los límites de las gramáticas de lenguaje natural es la siguiente:

> Were a language ever completely "grammatical" it would be a perfect engine of conceptual expression. Unfortunately, or luckily, no language is tyrannically consistent. All grammars leak.

En la que concluye su frase diciendo que todas las gramáticas fallarán en el objetivo de representar de manera completa los lenguajes naturales. En la práctica, estas gramáticas suelen tener los siguientes inconvenientes: O bien son *incompletas*, en el sentido que no cubren todo el lenguaje objetivo o bien *sobregeneran*, cubriendo expresiones que no deberían ser "gramaticales".

Más allá de este escenario negativo, una justificación sencilla de su uso es que las gramáticas son útiles, tanto para humanos como para computadoras. Para ambos, una gramática es un atajo, ya que provee reglas generales que reemplazan una gran cantidad de frases concretas. Además, conocer las reglas de una gramática usualmente mejora la calidad en la producción del lenguaje.

Tomemos como ejemplo la conjugación de verbos. El español posee más de 20 conjugaciones distintas para un mismo verbo, lo que hace prácticamente imposible encontrar todas las conjugaciones de un verbo en los textos del corpus[1] usado para entrenar un modelo. Sin embargo, sí es posible, aunque a veces también complejo, escribir reglas que computen todas las conjugaciones de un verbo.

La falta de datos en el conjunto de entrenamiento de modelos estadísticos es un problema del procesamiento de lenguaje natural y el machine learning en general. Si bien existen técnicas que intentan disminuir el impacto de la falta de información en los datos, para casos como el de conjugación de verbos parece que la mejor solución es atacar el problema usando reglas definidas en una gramática.

Para varias tareas de procesamiento de lenguaje natural es común usar n-gramas, las cuales son secuencias de n palabras dentro de una misma oración, para algún número natural n no muy grande. Ejemplos de uso de n-gramas son generación de lenguaje, POS tagging y traducción automática. Una problemática común para sistemas de traducción automática basados en n-gramas es la de traducir correctamente frases con dependencias lejanas dentro de una misma oración. Un ejemplo sacado del libro de GF [16, p. 6] en donde un traductor estadístico sencillo basado en tri-gramas fallaría para dependencias de larga distancia es el siguiente: Dada la frase "My mother immediately became very worried", una traducción erronea podría ser "Mi madre inmediatamente estaba muy preocupado", donde la concordancia con el sustantivo femenino "madre" no es correcta entre palabras alejadas (distancia mayor a tres, en el caso de tri-gramas), como se puede ver en el adjetivo "preocupado". El uso de gramáticas resuelve el problema del análisis correcto de dependencias alejadas, sin importar la cantidad de palabras que haya entre ellas.

La elección entre métodos basados en datos y métodos basados en reglas depende además si estamos priorizando la *robustez* del sistema, es decir, aceptar y producir cualquier frase posible o si priorizamos la *precisión* del sistema, es decir, producir resultados precisos, con la posible desventaja de no lograr aceptar o producir cualquier

[1] Un corpus lingüístico es un conjunto amplio y estructurado de ejemplos reales de uso de la lengua. Estos ejemplos pueden ser textos (los más comunes), o muestras orales (generalmente transcritas).

frase. Sistemas basados en reglas tienden a priorizar la precisión, mientras que sistemas basados en datos tienden a priorizar la robustez. Sin embargo, no deberíamos descartar la posibilidad de crear un sistema híbrido, es decir, uno que use reglas en conjunto con métodos basados en datos.

La temática desarrollada en la tesis nos llevó a elegir trabajar con gramáticas, y más específicamente con Grammatical Framework que es un lenguaje de programación de propósito particular para este tipo de tareas. A continuación repasamos las principales características del trabajo y de la herramienta que justifican nuestra elección:

(i) Al tener como objetivo representar semántica de lenguaje natural con fórmulas lógicas, es decir, expresiones de un lenguaje formal, nos parece una buena idea priorizar la precisión de la traducción, antes que la robustez.

(ii) Al ser un trabajo ideado como una herramienta educativa, también consideramos adecuado priorizar una traducción correcta. El hecho de no cubrir algunas frases, por otro lado, no es tan preocupante. Una frase no aceptada puede ser aprovechada para que el usuario de la aplicación intente realizar la traducción por su cuenta.

(iii) GF provee para una mejor experiencia de usuario un sistema de autocompletado, para generar frases que sean aceptadas por el sistema.

(iv) No conocemos ningún corpus de datos anotado que nos pueda ser útil para la traducción de lenguaje natural a fórmulas lógicas.

1.2 Motivación práctica

Este trabajo parte del supuesto de que el aprendizaje del lenguaje simbólico de lógica formal es problemático (Oller, 2013). Algunos aspectos de esta dificultad son analizados por los autores de "Language, Proof and Logic" utilizando datos empíricos (Barker-Plummer et al., 2011). Sin embargo, las razones por las cuales la formalización es una tarea difícil tanto de aprender para los alumnos como enseñar para los profesores, no son tan evidentes. La enseñanza de formalización, como traducción de un lenguaje no-formal o semi-formal a un lenguaje formal de la lógica, constituye generalmente una parte de los cursos de lógica de nivel universitario. Se presupone la posibilidad de llevar a cabo exitosamente la traducción de argumentos en lenguaje natural a algún lenguaje de la lógica formal. Un ejemplo típico de las dificultades de

comprensión de las que se suelen ocupar los textos introductorios de lógica es la interpretación del significado de la expresión "Si ... entonces..." en oraciones como "Si 2 es par entonces Argentina es un país". Si representamos esta frase como una proposición lógica usando el conectivo de implicación material[2], podemos concluir que la misma es verdadera, ya que tanto el antecedente como el consecuente son verdaderos. Sin embargo, una frase como esta no es intuitivamente cierta para personas que no están familiarizadas con el lenguaje formal de la lógica y su interpretación. Las expresiones en lenguaje natural "Si *antecedente* entonces *consecuente*" suelen ser usadas y consideradas razonables cuando existe una relación causal entre el *antecedente* y el *consecuente*. Otro ejemplo cuya interpretación suele atentar contra la intuición es el de la frase "Si 2 es impar entonces 2 es par", la cual representada como proposición lógica usando el conectivo de implicación material también es verdadera, ya que el antecedente (2 es impar) es falso.

En general, existen dos fuentes de dificultad cuando se traduce un texto de un lenguaje de origen a una lengua destino. La primera es la comprensión del texto en la lengua de origen y la segunda es la producción del texto en la lengua de destino. Oller considera que el obstáculo cognoscitivo principal al que las/los estudiantes deben hacer frente cuando intentan traducir un texto castellano al lenguaje de la lógica de predicados es del orden de la producción del texto en la lengua de destino, y tiene que ver con el hecho de que el número y la naturaleza de las categorías sintácticas del lenguaje de la lógica de primer orden son extremadamente diferentes del número y de la naturaleza de las categorías sintácticas del castellano.

Para traducir un texto al lenguaje de la lógica de primer orden es necesario realizar, después de evaluar si las limitaciones expresivas de la lengua de destino lo permiten, cambios complejos de clases de palabras y plasmar el significado del texto en una forma gramatical totalmente distinta a la que tenía en la lengua de origen. Como si esto fuera poco, las reglas que nos dicen cómo traducir categorías sintácticas de la lengua de origen a categorías sintácticas de la lengua de destino, tienen excepciones cuya explicación requiere cierto esfuerzo teórico. Consideremos, por ejemplo, la regla de traducción según la cual los adjetivos y los sustantivos comunes castellanos deben traducirse como predicados en el lenguaje de la lógica de primer orden, y los dos siguientes casos, aparentemente simples y paralelos, del uso de esta regla.

Podemos pensar que $Tr.ABC \wedge Eq.ABC$ es una traducción correcta para la frase "ABC es un triangulo equilátero", donde Tr y Eq representan los predicados "ser triángulo" y "ser equilátero", respectivamente.

[2] `https://es.wikipedia.org/wiki/Condicional_material`

Sin embargo, $Basq.Facundo \wedge Bajo.Facundo$, podría no ser una traducción correcta para la frase "Facundo es un basquetbolista bajo", donde $Basq$ y $Bajo$ representan los predicados "ser basquetbolista" y "ser bajo", respectivamente. Facundo podría ser considerada una persona alta, pero baja para ser un jugador de basquet, donde los estándares de altura son mayores.

1.2.1 Utilidad de la formalización y herramientas para su aprendizaje

En el área de las ciencias de la computación, dado un problema especificado de manera informal (usualmente en lenguaje natural), se considera de mucho valor poder construir una especificación formal que represente de la manera más fiel posible dicho problema. Por ejemplo, un programador que desea crear un programa en un lenguaje de programación particular (lenguaje formal), debe constantemente formalizar de manera correcta las necesidades expresadas en lenguaje natural, ya sean de un cliente o propias. El resultado de esta formalización puede ser directamente el programa en el lenguaje de programación elegido, aunque también es posible formalizar el problema en otro lenguaje formal como paso intermedio, como por ejemplo el lenguaje de la lógica de primer orden, el cual suele ser más conveniente para manipular y garantizar la corrección de la formalización con respecto a las necesidades e incluso facilitar la construcción correcta del programa.

Actualmente existen propuestas didácticas con soportes informáticos para el aprendizaje de lógica (Barrionuevo, 2008), sin embargo, la mayoría se centra en los sistemas deductivos o en la semántica formal de fórmulas lógicas. Pareciera, por lo tanto, que faltan soportes digitales que transparenten las dificultades inherentes a la traducción de lenguajes naturales a lenguajes simbólicos.

El objetivo principal de este trabajo es desarrollar una herramienta didáctica informática que facilite la adquisición del lenguaje formal de lógica de primer orden. Se toma como punto de partida la herramienta didáctica Sat [5], creada por profesores de la facultad y usada por estudiantes de primer año en la materia de Introducción a los Algoritmos. Esta herramienta permite crear "mundos" de figuras geométricas en un tablero y evaluar si ciertos predicados son válidos en estos "mundos". Nos pareció interesante y útil que esta herramienta posea una nueva funcionalidad, en la cual el usuario pueda ingresar una oración en castellano y se construya a partir de ella una fórmula lógica de primer orden, para que luego Sat pueda chequear su satisfactibilidad. De esta manera, la/el estudiante puede ir familiarizándose con la escritura formal a través de ejemplos que ella/el construye y contrastar su formalización con todas las

traducciones posibles. Creemos que esta nueva funcionalidad puede ser útil no solo para estudiantes de la carrera de Computación, sino también para estudiantes de otras carreras con cursos introductorios a lógica de primer orden, como es el caso de la carrera de Filosofía.

Capítulo 2

Grammatical Framework

En la introducción vimos por qué para nuestra tarea es conveniente utilizar un enfoque basado en gramáticas. En este capítulo introduciremos Grammatical Framework (GF) [12], que es la herramienta utilizada para definir la gramática de este trabajo, y además abarcaremos algunos de los supuestos teóricos subyacentes de este lenguaje.

2.1 Introducción a GF

GF es un lenguaje de programación diseñado para escribir gramáticas multilingües, creado por Aarne Ranta [13], profesor de Ciencias de la Computación en la Universidad de Gotemburgo.

El desarrollo de este lenguaje comenzó en 1998 en el centro de investigación Xerox en Grenoble, y hoy en día es mantenido por su comunidad como un proyecto open source, disponible para ser instalado y usado directamente en sistemas operativos Linux, macOS y Windows. Algunos lenguajes de programación de uso general como Haskell, Python, Java y Javascript poseen librerías para importar y utilizar las gramáticas generadas en GF.

GF puede ser usado principalmente como: (i) Un lenguaje de propósito específico en gramáticas, como YACC, Bison, Happy y BNFC (ii) Un lenguaje de programación funcional, como Haskell, Lisp, OCaml, SML y Scheme (iii) Una plataforma de desarrollo para gramáticas de lenguaje natural, como LKB, XLE y Regulus (iv) Un formalismo de gramáticas categoriales, como ACG y CCG. (v) Un framework lógico, como Agda, Coq y Isabelle, equipado con sintaxis concreta en adición a lógica (vi) Una plataforma para traducción automática, como Moses y Apertium.

Como herramienta de procesamiento de lenguaje, dada una gramática definida en GF, es posible realizar las siguientes operaciones: *Parsear*, definido como el proceso

de transformar cadenas de símbolos en árboles de acuerdo a las reglas definidas en la gramática; *linealizar*, vista como la operación inversa de parsear, es decir, definido como el proceso de transformar árboles en cadenas de símbolos de acuerdo a las reglas definidas en la gramática, y por último, *generar* texto perteneciente al lenguaje definido por la gramática.

Algunas de las razones por las que GF nos pareció una herramienta apropiada para realizar este trabajo son listadas a continuación:

(i) A nivel sintáctico la expresividad de GF es superior al de gramáticas libres de contexto, permitiendo representar estructuras sintácticas complejas presentes en lenguaje natural. Este punto será desarrollado más adelante en la sección 2.4.

(ii) El modo de construir las gramáticas (vistas como programas en GF) es semejante al modo de definir programas declarativos en lenguajes funcionales, lo que permite escribir las gramáticas de una forma más compacta y abstracta.

(iii) Es posible definir módulos para distintas partes de la gramática, lo que permite lograr una mejor organización y división del trabajo.

(iv) GF posee una librería propia (Resource Grammar Library), la cual provee una interfaz a gramáticas que definen la estructura sintáctica de varios idiomas (entre ellos el español).

(v) Es posible definir gramáticas multilingües, es decir, gramáticas capaces de manipular texto de varios idiomas en paralelo. Esto permite generar un traductor entre distintos lenguajes de una manera sencilla. Pensaremos al lenguaje de la lógica de primer orden y al castellano, como los dos lenguajes de nuestra gramática multilingüe.

Profundizaremos un poco más sobre estas ventajas en las siguientes secciones.

2.2 Sintaxis abstracta y concreta

La manera en que GF trabaja está inspirada en gran parte por el funcionamento de los compiladores. Una gramática está definida por dos componentes principales, una sintaxis abstracta y una o más definiciones de sintaxis concreta. Podemos pensar al modelo de GF como un "Multi-source Multi-target compiler", es decir, un compilador en el cual se pueden aceptar varios lenguajes de entrada y generar expresiones en

varios lenguajes de salida. Un diagrama de ejemplo de un "Multi-source Multi-target compiler" puede observarse en la Figura 2.1.

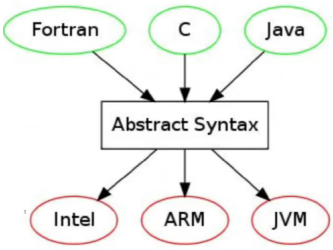


Figura 2.1: Ejemplo de Multi-source Multi-target compiler.

Podemos diferenciar entonces entre gramáticas concretas y gramáticas abstractas. En las gramáticas concretas las producciones definen de manera directa el lenguaje concreto y se deben tener en cuenta detalles de la sintaxis del lenguaje que no son sustanciales, como el uso de paréntesis para desambiguar el análisis de expresiones aritméticas. La gramática de expresiones aritméticas tiene en cuenta estos detalles, definiendo la precedencia del producto ante la suma, la asociatividad de los operadores binarios y el uso de paréntesis. En las gramáticas abstractas, en cambio, las producciones no definen frases concretas, sino que definen la estructura del lenguaje intentando capturar la semántica relevante del mismo. A las frases del lenguaje de las gramáticas abstractas las denotaremos por frases abstractas y las representaremos con árboles a los que llamaremos árboles de sintaxis abstractos (AST).

Para construir los árboles de nuestra sintaxis abstracta, GF nos provee dos sentencias:

cat define las categorías o tipos permitidos para un árbol.

fun define un constructor de árboles teniendo en cuenta las categorías definidas.

La sentencia **fun** $f : C_1 \to C_2 \to \ldots \to C_n \to C$ define un constructor f de árboles de categoría C, a partir de árboles de categorías C_i, con $1 \leq i \leq n$.

La sintaxis concreta, por su parte, cumple la función de definir de qué manera se linealizan los posibles AST de la sintaxis abstracta, creando de esta manera el lenguaje concreto de la gramática. A partir de la definición de la sintaxis concreta, GF genera automáticamente un parser del lenguaje producido. Para definir la linealización de un AST, GF nos provee dos sentencias:

lincat define la categorías o tipos para la linealización de un árbol. Ejemplos: strings, records, tables.

lin define de qué manera se linealizan los AST, teniendo en cuenta las categorías de linealización.

Decimos que una sintaxis concreta es *completa* con respecto a una sintaxis abstracta si para cada categoría definida por **cat** en la sintaxis abstracta existe una definición de la categoría de linealización en la sintaxis concreta mediante **lincat**.

Además, una sintaxis concreta es *correcta* si las definiciones de linealización, mediante **lin** en la sintaxis concreta, son bien tipadas con respecto a los constructores de AST, definidos mediante la sentencia **fun**. Formalmente, esto puede ser expresado de la siguiente manera: Si un constructor f fue introducido por la sentencia

$$\textbf{fun } f : C_1 \to C_2 \to \ldots \to C_n \to C$$

entonces la regla de linealización para f debe ser de la forma

$$\textbf{lin } f = t$$

donde, si denotamos el tipo de la linealización de una categoría C por C^*

$$t : C_1^* \to C_2^* \to \ldots \to C_n^* \to C^*$$

Veamos como podemos representar una gramática libre de contexto dada en notación BNF con una gramática de GF. Definiremos con estas gramáticas el lenguaje $L = \{a^n b^n | n \geq 0\}$.

Sea la siguiente gramática libre de contexto, con ε el string vacío,

```
<S> ::= a<S>b | ε
```

en GF la podemos representar de la siguiente manera: definiendo primero la sintaxis abstracta

```
cat S ;
fun AB : S -> S ;
fun Empty : S ;
```

y luego la sintaxis concreta

```
lincat S = Str ;
lin AB s = "a" ++ s ++ "b" ;
lin Empty = "" ;
```

De esta manera podemos parsear el string “aaabbb” generando el árbol de sintaxis abstracto `AB (AB (AB (Empty))`.

GF permite que una única sintaxis abstracta esté acompañada de varias definiciones de sintaxis concreta. De esta manera es posible realizar traducciones entre lenguajes concretos usando como intermediaria a la sintaxis abstracta. Para traducir una oración de un lenguaje A a un lenguaje B, los pasos a seguir son:

(i) *Parsear* la oración del lenguaje A. Esto genera un AST de la sintaxis abstracta.

(ii) *Linealizar* el AST generado en el paso anterior a una frase del lenguaje B.

Veamos un ejemplo en el que definiremos una gramática con una sintaxis abstracta acompañada de dos definiciones de sintaxis concreta para representar expresiones aritméticas en Java. Una sintaxis concreta definirá las expresiones para el lenguaje Java y la otra definirá las mismas expresiones en código assembler para JVM (Java Virtual Machine).

Es importante notar que lo que estaremos haciendo es definir un compilador muy sencillo de código Java. Es posible en este caso también trabajar con GF de manera inversa, es decir, decompilar el código assembler para producir código Java.

Definamos la sintaxis abstracta para representar la suma y multiplicación de expresiones enteras,

```
cat Exp ;
fun N : Int -> Exp ;
fun plus : Exp -> Exp -> Exp ;
fun mult : Exp -> Exp -> Exp ;
```

la sintaxis concreta para representar estas expresiones en Java, donde los paréntesis en la suma y multiplicación son agregados para evitar ambigüedad al momento de parsear expresiones como $1 + 2 * 3$,

```
lincat Exp = Str ;
-- por defecto: lincat Int = {s : Str}
lin N i = i.s ;
lin plus x y = "(" ++ x ++ "+" ++ y ++ ")" ;
lin mult x y = "(" ++ x ++ "*" ++ y ++ ")" ;
```

y la sintaxis concreta para representar estas expresiones en assembler de JVM

```
lincat Exp = Str ;
-- por defecto: lincat Int = {s : Str}
lin N i = "iconst_" ++ i.s ;
lin plus x y =  x ++ y ++ "iadd" ;
lin mult x y = x ++ y ++ "imul" ;
```

De esta manera es posible traducir la siguiente expresión de Java en una expresión de código assembler de JVM.

```
((1 + 2) * 3)

parse
⟹

mult (plus (N 1) (N 2)) (N 3)

lin
⟹ iconst_1
   iconst_2
   iadd
   iconst_3
   imul
```

2.2.1 Gramáticas en GF para representar lenguaje natural

Como hemos visto, cada sintaxis concreta definida en GF es *reversible* en el sentido que puede ser usada tanto para parsear como para linealizar. Esto nos permite, al momento de crear gramáticas con más de una sintaxis concreta, definir una traducción entre cada uno de los lenguajes involucrados de manera bidireccional. En otras herramientas de traducción, si uno desea definir un traductor para un conjunto de n lenguajes, lo normal sería definir un traductor para cada par de lenguajes y en ambos sentidos. Podemos observar el diagrama de traducción que se genera en este caso, en la Figura 2.2.

GF realiza la traducción entre cada par de lenguajes usando la sintaxis abstracta como lengua intermedia, por lo que no es necesario definir una traducción independiente para cada par de lenguajes. Podemos observar el diagrama de traducción que se genera en este caso, en la Figura 2.3.

Veamos un nuevo ejemplo en el que definiremos una única sintaxis abstracta y una sintaxis concreta para distintos tipos de lenguajes naturales, según el orden en que aparecen sus constituyentes sintácticos. Analizaremos el caso simple de oraciones transitivas, donde los constituyentes sintácticos que evaluaremos son sujeto (S), objeto (O) y verbo (V).

La oración transitiva que usaremos de ejemplo es "Juan ama a María". En este caso, el orden en que aparecen los constituyentes es SVO, es decir, primero el sujeto, luego el verbo y por último el objeto. Este orden es también el que aparece en oraciones transitivas en inglés. En este caso, la oración en inglés sería "John loves Mary". Sin embargo, no todos los idiomas usan el orden SVO. Idiomas como el latín usan el orden SOV e idiomas como el árabe usan el orden VSO.

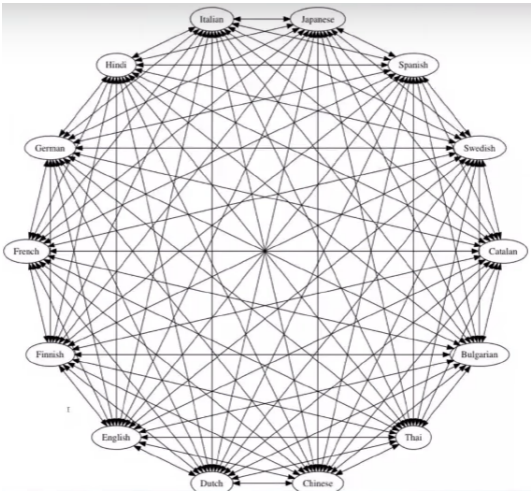


Figura 2.2: Diagrama de traducción sin sintaxis abstracta.

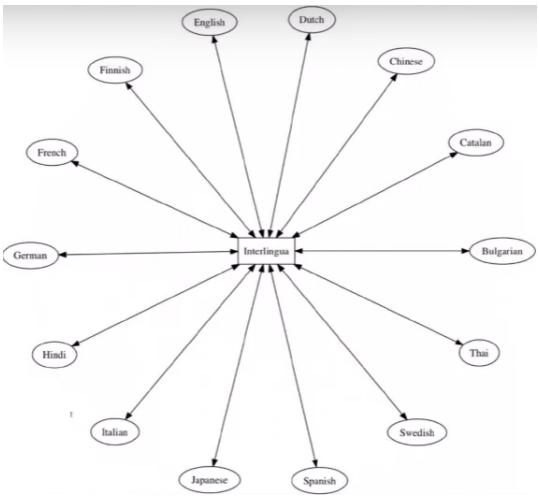


Figura 2.3: Diagrama de traducción con sintaxis abstracta.

Definiremos entonces una sintaxis concreta para cada uno de estos tres ordenes: SVO, SOV y VSO. La sintaxis abstracta para este ejemplo se puede definir de la siguiente manera:

```
cat S ; NP ; VP ; VTrans ;
fun Pred : NP -> VP -> S ;
fun Compl : VTrans -> NP -> VP ;
fun John, Mary : NP ;
fun Love : VTrans ;
```

Las categorías de esta sintaxis abstracta reflejan categorías sintácticas, con S representando una oración completa, NP un sintagma nominal, VP un sintagma verbal y $VTrans$ un verbo transitivo.

Las categorías de linealización serán identicas para el orden SVO y SOV:

```
lincat S, NP, VP, VTrans = Str ;
```

La linealización de los constructores John, Mary y Love dependerá del idioma de cada lenguaje concreto. Para el caso del inglés (SVO) una posible linealización es la siguiente

```
lin John = "John" ;
lin Mary = "Mary" ;
lin Love = "loves" ;
```

Para un idioma de orden SVO definimos las siguientes reglas de linealización

```
lin Pred s vo = s ++ vo ;
lin Compl v o = v ++ o ;
```

y para un idioma de orden SOV definimos las siguientes reglas de linealización

```
lin Pred s vo = s ++ vo ;
lin Compl v o = o ++ v ;
```

Sin embargo, al intentar definir la sintaxis concreta para el orden VSO, nos damos cuenta que esto no es posible. Esta es una limitación conocida de gramáticas libres de contexto, en la que una producción no puede descomponer las subproducciones inmediatas, sino que debe usar cada una como un único string. GF soluciona este problema y extiende la expresividad de gramáticas libres de contexto incluyendo **records** como tipos de datos de linealización. Un record puede ser pensado como una tupla en la que cada componente tiene una etiqueta para ser identificada.

El *tipo* de un record se define de la siguiente manera,

$$\{t_1 : C_1; \ldots; t_n : C_n\}$$

donde t_i y C_i, con $1 \leq i \leq n$, definen la etiqueta y la categoría de linealización correspondiente al componente i-ésimo del record, respectivamente.

Un *record* se define de la siguiente manera,

$$\{t_1 = c_1; \ldots; t_n = c_n\}$$

donde t_i y c_i, con $1 \leq i \leq n$, son la etiqueta del componente i-ésimo y el componente i-ésimo de tipo C_i, respectivamente. Para obtener el componente i-ésimo de un record r, escribimos $r.t_i$

Veamos entonces una representación del orden VSO usando records,

```
lincat S, NP, VTrans = Str ;
lincat VP = { verb : Str ; obj : Str } ;

lin Pred s vo = vo.verb ++ s ++ vo.obj ;
lin Compl v o = { verb = v ; obj = o } ;
```

en la cual el tipo de linealización de VP es ahora un record, siendo posible así usar por separado sus componentes internos, es decir, el verbo transitivo y el objeto.

2.3 Intérprete de GF

Para utilizar las gramáticas definidas, GF cuenta con un intérprete en el cual podemos ejecutar comandos. A continuación son listados los comandos que aparecerán a lo largo del trabajo:

`import (i)`: Importa una gramática en código fuente o compilada.

`parse (p)`: Parsea un string a un árbol de sintaxis abstracto.

`linealize (l)`: Linealiza un árbol de sintaxis abstracto.

`generate_random (gr)`: Genera aleatoriamente un árbol de sintaxis abstracto de la gramática.

`visualize_tree (vt)`: Permite usar una herramienta de visualización para mostrar los árboles de sintaxis abstractos gráficamente.

`put_string (ps)`: Toma un string el cual puede ser procesado por una función. Usualmente este comando es utilizado para aplicar un lexer sobre el string de entrada.

`put_tree (pt)`: Toma un árbol de sintaxis abstracto el cual puede ser procesado por una función. Veremos con más detalle la utilidad de este comando en la sección 2.6.

`define_command (dc)`: Permite crear un atajo a un comando.

Uso: `dc nombre_atajo comando`. Para ejecutar el atajo en el intérprete este debe ser llamado como `%nombre_atajo`.

Veamos un ejemplo de uso del intérprete, tomando como referencia la gramática de expresiones enteras de la sección anterior. Supongamos que la sintaxis abstracta está definida en un archivo `exp.gf` y la definición de la sintaxis concreta de Java y JVM en archivos `expJava.gf` y `expJVM.gf`, respectivamente.

```
-- importamos las distintas componentes de la gramática
> import exp.gf expJava.gf expJVM.gf

-- parseamos una expresión de Java
> parse -lang="expJava" "( 1 + ( 2 * 3 ) )"
plus (N 1) (mult (N 2) (N 3))
```

Usando pipes (`|`) podemos redireccionar la salida de `parse` a la entrada de `linearize`, logrando así una compilación de código Java a código de la JVM.

```
> parse -lang="expJava" "( 1 + ( 2 * 3 ) )" | linearize -lang="expJVM"
iconst_1 iconst_2 iconst_3 imul iadd

-- generamos un AST aleatorio y lo visualizamos
> generate_random | vt -view="open"

-- usamos el lexer code para tokenizar strings de expresiones enteras
> put_string -lexcode "(1+(2*3))"
( 1 + ( 2 * 3 ) )

-- usamos el lexer junto con parse y linearize
-- notar el uso de abreviaciones de los comandos
> ps -lexcode "(1+(2*3))" | p -lang="expJava" | l -lang="expJVM"
iconst_1 iconst_2 iconst_3 imul iadd

-- definimos y usamos un atajo compile
-- ?0 representa un parámetro del comando
> dc compile p -lang="expJava" ?0 | l -lang="expJVM"
> %compile "( 1 + ( 2 * 3 ) )"
iconst_1 iconst_2 iconst_3 imul iadd
```

2.4 Expresividad de GF

El poder expresivo de GF como un formalismo para definir gramáticas ha sido probado en (Ljunglöf, 2004). En este trabajo se muestra que GF es más expresivo que las conocidas gramáticas libres de contexto (CFG) y además que es equivalente al formalismo denominado "Parallel multiple context-free grammar" (PMCFG) (Seki et al., 1993). Las PMCFG's pueden ser pensadas de manera informal como gramáticas sobre tuplas de strings.

Un dato importante sobre el formalismo PMCFG es que el parsing puede realizarse en tiempo polinomial con respecto al largo de la frase. La implementación del parser de GF está detallada en (Angelov, 2009), con la cualidad de que el parsing se efectúa de manera incremental, permitiendo a GF mostrar las posibles continuaciones de una frase que aún no esté completa.

Una ventaja de trabajar con este formalismo gramatical es que permite representar estructuras sintácticas complejas presentes en lenguaje natural, que gramáticas libres de contexto no son capaces de representar (Shieber, 1985). Otra ventaja importante es que la definición de gramáticas de lenguaje natural resulta más simple y compacta con formalismos más expresivos que CFG, como es el caso de PMCFG.

Un ejemplo para el cual definir gramáticas de lenguaje natural con CFG puede ser tedioso es el de concordancia. Como ejemplo, definamos las siguientes reglas

```
S := NP VP
NP := Det N
VP := V

Det := El | Los
N := perro | perros
V := ladra | ladran
```

Con esta definición, las oraciones "El perro ladra" y "El perro ladran" pertenecen al lenguaje de la gramática. Esto se debe a que no hay una forma de establecer concordancia de número entre la frase nominal y la frase verbal. Para lograr esta concordancia, una solución es replicar las reglas de la siguiente manera,

```
S := S_sg | S_pl
S_sg := NP_sg VP_sg
S_pl := NP_pl VP_pl
NP_sg := Det_sg N_sg
NP_pl := Det_pl N_pl
VP_sg := V_sg
VP_pl := V_pl
```

```
Det_sg := El
Det_pl := Los
N_sg := perro
N_pl := perros
V_sg := ladra
V_pl := ladran
```

donde "El perro ladran" ya no es una frase perteneciente al lenguaje de la gramática. Sin embargo, la concordancia puede ser también de género y número, como en el caso de adjetivos junto con frases nominales. Por ejemplo, en la frase "El perro enojado ladra", el adjetivo "enojado" concuerda en género y número con la frase nominal "El perro", formando así una nueva frase nominal. Replicar estas reglas a lo largo de la gramática no parece ser una buena solución y es aquí donde puede ser de gran ayuda contar con formalismos más expresivos, como PMCFG, que es el formalismo gramatical subyacente en GF. Veamos a continuación una solución al problema de concordancia de número en GF, aunque antes introduzcamos dos conceptos nuevos de GF: *parámetros* y *tablas*, que nos permitirán excluir de la gramática las frases que no respetan la concordancia sin tener que replicar la producción de cada categoría sintáctica.

Los *parámetros* nos servirán para definir en la sintaxis concreta las características sobre las cuales puede haber concordancia. Por ejemplo, para el caso de concordancia de número, definimos un parámetro para distinguir número singular y plural.

```
param Num = Sg | Pl ;
```

Además, podemos pensar que para ciertas categorías sintácticas la característica de género ya está fijada, mientras que para otras será una característica variable. Para el sustantivo común "vaca" el género está fijado (femenino), sin embargo, el género del adjetivo "ruidosa/o" podría variar en femenino y masculino, según el sustantivo al que acompañe. Usaremos *records* para determinar las características inherentes a una categoría. Por ejemplo, al sustantivo "vaca" lo podemos representar de la siguiente manera:

```
param Gen = Fem | Masc ;
lincat N = { s : Str ; gen : Gen } ;
lin vaca = { s = "vaca" ; gen = Fem } ;
```

Usaremos *tables* para establecer que una categoría sintáctica tiene una característica variable. La sintaxis de *tables* es definida a continuación. El *tipo* de una *table* se define `P => C`, donde `P` es un tipo definido mediante `param` y `C` es una categoría de linealización. Una *table* de tipo `P => C`, con parámetro `P` definido por

`param P = p1 | ... | pn`, y `c1, ..., cn` pertenecientes a la categoría de linealización `C`, se creará de la siguiente manera

```
table { p1 => c1 ; ... ; pn => cn }.
```

Dado que `P` es una enumeración podemos pensar que las tablas nos dan una manera conveniente de definir una función de `P` en `C`. Si `t` es una tabla, con `t!p` obtenemos el valor asociado al parámetro `p` en la tabla `t`.

Definamos entonces la sintaxis abstracta de la gramática que respeta la concordancia en número entre la frase nominal y verbal,

```
cat S ; NP ; VP ; N ; V ; Det ;
fun
  Pred : NP -> VP -> S ;
  mkNP : Det -> N -> NP ;
  mkVP : V -> VP ;
  el : Det ;
  perro : N ;
  ladra : V ;
```

y la sintaxis concreta para el español usando parámetros y tablas

```
param Num = Sg | Pl ;

lincat S = Str ;
lincat Det, N, NP, V, VP = Num => Str ;

lin
  Pred np vp = (np!Sg ++ vp!Sg) | (np!Pl ++ vp!Pl) ;
  mkNP det n = table { num => det!num ++ n!num } ;
  mkVP v = v ;

  el = table { Sg => "el" ; Pl => "los" } ;
  perro = table { Sg => "perro" ; Pl => "perros" }  ;
  ladra = table { Sg => "ladra" ; Pl => "ladran" } ;
```

Con esta gramática podemos parsear tanto “El perro ladra” como “Los perros ladran”. Para agregar concordancia de género la sintaxis abstracta no debería cambiar. En la sintaxis concreta debería agregarse un nuevo parámetro para representar variaciones de género y usarlo apropiadamente en *records* para representar género inherente y *tables* para representar género variable. Veamos otro ejemplo en donde la expresividad de GF permite generar lenguajes que no son libres de contexto. Usando el *pumping*

lemma[1] (Hopcroft et al., 2006) se puede probar que los lenguajes libres de contexto no permiten modelar correctamente los siguientes fenómenos:

(i) multiple agreement: ejemplificado por el lenguaje $L = \{a^n b^n c^n | \ n \geq 0\}$

(ii) crossed agreement: ejemplificado por el lenguaje $L = \{a^n b^m c^n d^m | \ n \geq 0\}$

(iii) duplication: ejemplificado por el lenguaje $L = \{ww : w \in \Sigma^*\}$

Veremos como producir el lenguaje $L = \{a^n b^n c^n | n \geq 0\}$ con una gramática definida en GF. Definamos primero la sintaxis abstracta,

```
flags startcat = S ;  -- categoria incial
cat S ; ABC ;

fun mkS : ABC -> S ;
fun build : ABC -> ABC ;
fun empty : ABC ;
```

y la sintaxis concreta

```
lincat S = Str ;
lincat ABC = { a : Str ; b : Str ; c : Str } ;

lin mkS r = r.a ++ r.b ++ r.c ;
lin build r =  { a = r.a ++ "a" ; b = r.b ++ "b" ; c = r.c ++ "c" };
lin empty = { a = "" ; b = "" ; c = "" } ;
```

Como ejemplo, el string "aabbcc" será parseado al AST `mkS (build (build (empty)))`.

2.5 Resource Grammar Library

Las gramáticas de lenguaje natural creadas en GF pueden dividirse en dos grandes campos. Por un lado las *gramáticas semánticas o de dominio* son aquellas que hacen énfasis en una aplicación particular. Estas gramáticas no suelen cubrir frases fuera del dominio y sus categorías no están vinculadas de manera directa a categorías sintácticas del lenguaje, sino a categorías semánticas presentes en dicho dominio. Comúnmente los usuario de GF son programadores de gramáticas semánticas, es decir, poseen un conocimiento avanzado del dominio de aplicación y no desean representar

[1]No pude conseguir el artículo original donde se presenta el pumping lemma, por lo que cito el libro Introduction to Automata Theory, Languages, and Computation (3rd Edition), el cual posee la definición y demostración de dicho lema.

explícitamente las categorías sintácticas del lenguaje en su gramática, ya sea por el tiempo que esto tomaría o por falta de conocimiento específico en lingüística.

Por otro lado, las *gramáticas sintácticas*, definidas usualmente por lingüistas, son aquellas que hacen énfasis en las categorías sintácticas del lenguaje y no cubren ningún dominio en particular. Las categorías de las gramáticas sintácticas son las categorías sintácticas y sus reglas representan directamente las reglas de sintaxis del lenguaje.

El propósito de la Resource Grammar Library[2] (RGL) es proveer al programador de gramáticas de lenguaje natural las reglas principales de morfología y sintaxis de distintos lenguajes. De este manera se genera una división de trabajo adecuada para definir gramáticas en GF.

La RGL provee una interfaz de gramáticas sintácticas de distintos idiomas, actualmente 32 incluido el español, lo cual es muy útil para la mayoría de los usuarios de GF, cuya intención suele ser definir gramáticas para un determinado dominio.

Veamos un ejemplo de uso de la RGL en una gramática semántica sencilla sobre comentarios de animales. Definamos la sintaxis abstracta

```
cat Comment, Subject, Animal, Quality, Action ;

fun
  Pred : Subject -> Action -> Comment ;
  The : Animal -> Subject ;
  Kind : Animal -> Quality -> Animal ;

  perro : Animal ;
  blanco : Quality ;
  ladra, corre : Action ;
```

y la sintaxis concreta usando la RGL

```
lincat
  Comment = S ;
  Subject = NP ;
  Animal = CN ;
  Quality = AP ;
  Action = VP ;

lin
  Pred subj act = mkS subj act ;
  The anim = mkNP the_Det animal ;
  Kind anim qual = mkCN qual anim ;
```

[2] http://www.grammaticalframework.org/lib/doc/synopsis.html

```
perro = mkCN (mkN "perro") ;
blanco = mkAP (mkA "blanco") ;
ladra = mkVP (mkV "ladra") ;
corre = mkVP (mkV "corre") ;
```

Esta definición usando la RGL nos proporciona concordancia de género y número sin usar estructuras de bajo nivel como records y tables. De esta manera podemos crear gramáticas semánticas sin tener que preocuparnos por detalles sintácticos como la inflexión morfológica de las palabras (la RGL genera cada inflexión) ni de la concordancia.

Se puede observar que esta gramática genera frases correctas y variables tanto en género y número como "El perro blanco ladra", "La perra blanca ladra" y "Los perros blancos ladran".

2.6 Condiciones y acciones semánticas en sintaxis abstracta

Hasta el momento, la expresividad de la sintaxis abstracta de las gramáticas dadas en GF no es diferente a la sintaxis abstracta implícita de las gramáticas libres de contexto. Los tipos de los constructores en ambos casos tienen la forma

$$C_1 \to C_2 \to \ldots \to C_n \to C$$

Sin embargo, GF nos permite extender la expresividad de la sintaxis abstracta agregando tipos dependientes al sistema de tipos. Veremos luego que extender la expresividad de la sintaxis abstracta nos permitirá crear gramáticas capaces de definir *condiciones semánticas* sobre los elementos del lenguaje. Un ejemplo clásico de una frase gramatical, pero semánticamente sin sentido, dada por Noam Chomsky (Chomsky, 1957), es:

> Colorless green ideas sleep furiously[3]

Aunque la oración sea gramaticalmente correcta, no es posible derivar un significado obvio y comprensible de ella, demostrando de este modo, la distinción entre la sintaxis y la semántica.

En (Luque, 2017) se menciona una diferencia entre *gramaticalidad* y *aceptabilidad* de una oración. La gramaticalidad hace referencia a la posibilidad de una gramática

[3]En español: Las ideas verdes incoloras duermen furiosamente

de explicar mediante reglas dicha oración. Como las gramáticas de lenguaje natural son objetos ideales, la gramaticalidad de una oración suele ser determinada de manera indirecta mediante el juicio intuitivo de un hablante-oyente competente. La aceptabilidad refiere a qué tan aceptable es una oración para un hablante-oyente no ideal, al margen del cumplimiento de reglas gramaticales. En la aceptabilidad suelen influir otros factores como la comprensibilidad, simplicidad y frecuencia de la oración en cuestión.

Bajo esta perspectiva la frase de Chomsky es considerada gramatical por hablantes-oyentes del idioma inglés sin embargo no es considerada aceptable ya que carece de un sentido claro.

2.6.1 Tipos Dependientes

Los tipos dependientes en GF son heredados de la teoría de tipos intuicionista de Per Martin-Löf (Martin-Löf, 1998; Nordström et al., 1990), distinguiendo a GF de muchos formalismos gramaticales y lenguajes de programación funcional, que no poseen tipos dependientes. Un *tipo dependiente* es un tipo cuya definición depende de uno o más valores de otro tipo.

Un ejemplo simple de uso de tipos dependientes es el de vectores de números reales de una cierta dimensión. Podemos definir el tipo de un vector de dimensión n como: $Vector\ n$ y una función que tome un número natural n y nos devuelva un vector de dimensión n de la siguiente manera $f : (n : Nat) \rightarrow Vector\ n$

Para definir tipos dependientes en GF, generalizaremos la definición de categorías dada hasta el momento:

$$\textbf{cat}\ C\ G$$

donde denotaremos a G como un *contexto*. Un contexto es una secuencia de *hipótesis* de la forma

$$(x : T)$$

donde x es una variable de tipo T. Podemos pensar que una categoría no dependiente es un caso particular de esta definición, en donde el contexto es una secuencia vacía de hipótesis.

Un *tipo básico* en GF es una categoría aplicada a cada uno de los argumentos especificados por su contexto. Más formalmente, si definimos una categoría como

$$\textbf{cat}\ C\ (x_1 : T_1) \ldots (x_n : T_n)$$

podemos formar un tipo básico

$$C\ a_1 \ldots a_n$$

si los argumentos están bien tipados con respecto al contexto, es decir,

$$a_1 : T_1, \ldots, a_n : T_n$$

reemplazando en cada T_i las ocurrencias de x_j por a_j, con $1 \leq i, j \leq n$.

De esta manera podemos definir tipos de funciones dependientes cuyo valor de retorno dependa de sus argumentos. La forma general de esto es

$$(x : A) \to B$$

donde la variable x podría ocurrir en el tipo B.

Veamos un ejemplo en el que el uso de tipos dependientes nos permitirá establecer condiciones semánticas sobre los elementos del lenguaje. Daremos primero una gramática en GF sin el uso de tipos dependientes, y veremos cuales son sus defectos. Definamos primero, como es usual, su sintaxis abstracta

```
flags startcat = Sent ;  -- categoria incial
cat Sent ; Sujeto ; Accion ; Objeto ;

fun
  mkSent : Sujeto -> Objeto -> Accion -> Sent ;
  Hombre, Mujer : Sujeto ;
  Leer, Comer : Accion ;
  Libro, Manzana : Objeto ;
```

Un problema de esta gramática es que la frase abstracta `mkSent Hombre Libro Comer` es posible, sin embargo, no tiene mucho sentido que la acción `Comer` vaya acompañada del objeto `Libro`.

Una segunda versión de esta gramática abstracta es la siguiente,

```
flags startcat = Sent ;  -- categoria incial
cat Sent ; Sujeto ; Objeto ;
  Accion Objeto ;  -- Azucar sintáctico para Accion (o : Objeto)
  Legible Objeto ;
  Comible Objeto ;
fun
  mkSent : Sujeto -> (o : Objeto) -> Accion o -> Sent ;
  Hombre, Mujer : Sujeto ;

  manzana_comible : Comible Manzana ;
  libro_legible : Legible Libro ;
```

```
Leer : (o : Objeto) -> Legible o -> Accion o ;
Comer : (o : Objeto) -> Comible o -> Accion o ;

Libro, Manzana : Objeto ;
```

en la cual no es posible construir la frase abstracta `mkSent Hombre Libro Comer`, ya que `Comer` no es una acción permitida para el objeto `Libro`. Esto se logra creando dos categorías dependientes, `Legible` y `Comible`, que denominaremos *clases*, las cuales nos servirán para determinar los objetos que pueden ser leídos y comidos, respectivamente. Los elementos `manzana_comible` y `libro_legible`, a los que llamaremos *objetos de prueba*, determinan que el objeto `Manzana` es comible y que el objeto `Libro` es legible. Luego, podemos ver que `Leer` y `Comer` son ahora funciones dependientes que necesitan un objeto y una prueba de que se puede realizar dicha acción en ese objeto. La teoría subyacente en la definición de esta gramática es el Isomorfismo de Curry-Howard[4], en el cual se establece una equivalencia entre proposiciones y tipos (y entre pruebas y términos).

Teniendo en cuenta este isomorfismo, en nuestro ejemplo la *clase* `Legible` define un predicado sobre `Objeto` el cual es válido para un objeto `o` si existe una *prueba*, es decir, un elemento de tipo `Legible o`. En nuestro caso, podemos decir que el objeto `Libro` es legible ya que para el mismo existe la prueba `libro_legible`.

2.6.2 Definiciones semánticas y funciones de transferencia

Hasta el momento, en la sintaxis abstracta solo hemos definido la signatura de las funciones, pensadas como constructores de frases abstracta, lo que fue suficiente para crear todas las gramáticas que hemos visto. Sin embargo, en GF también es posible dar una definición semántica para las funciones de la sintaxis abstracta. Podemos interpretar a las definiciones semánticas como acciones semánticas sobre los árboles abstractos, es decir, una computación del valor semántico del árbol usando la definición semántica de sus subárboles inmediatos. Para dar una definición semántica de una función, GF nos provee la sentencia `def`.

Veamos una nueva gramática abstracta que define listas de enteros y demos una definición semántica para concatenar listas.

```
cat List ;

data Empty : List ;
```

[4] `https://en.wikipedia.org/wiki/Curry-Howard_correspondence`

```
data Cons : Int -> List -> List ;

fun Concat : List -> List -> List ;
def
  Concat Empty l = l ;
  Concat (Cons h t) l = Cons h (Concat t l) ;
```

De esta gramática abstracta podemos hacer las siguientes observaciones: Usamos `def` para definir la concatenación de dos listas, haciendo pattern matching sobre los posibles constructores de lista, de una manera muy similar al pattern matching en Haskell.

Además, podemos ver que usamos por primera vez la sentencia `data`, que reemplaza a `fun`, definiendo a `Empty` y `Cons` como constructores de la categoria `List`. Esto es necesario para que el pattern matching interprete a `Empty` y `Cons` como constructores de la categoría `List`. Definamos una sintaxis concreta para esta gramática:

```
lincat List = Str ;
-- por defecto: lincat Int = {s : Str}

lin Empty = "[]" ;
lin Cons x l = "(" ++ x.s ++ ":" ++ l ++ ")" ;

lin Concat l1 l2 = l1 ++ "++" ++ l2 ;
```

GF nos permite computar un árbol parseado usando el comando `put_tree` con la opción `-compute`. Veamos el siguiente ejemplo en el que parseamos la concatenación de dos listas, computamos dicha concatenación y linealizamos el resultado.

```
-- el flag -tr imprime el resultado del comando
> p -tr "( 2 : ( 1 : [] ) ) ++ ( 0 : [] )" | pt -tr -compute | l
Concat (Cons 2 (Cons 1 Empty)) (Cons 0 Empty)
Cons 2 (Cons 1 (Cons 0 Empty))
( 2 : ( 1 : ( 0 : [] ) ) )
```

Usando definiciones semánticas podemos crear funciones que transformen el árbol de sintaxis abstracto generado por el parser en un árbol con una estructura distinta, el cual buscaremos que sea más propicio para ser linealizado al lenguaje objetivo. A estas funciones las denominaremos *funciones de transferencia*. Como ya hemos visto, la linealización en GF es una operación *composicional*. Formalmente, decimos que una operación * es composicional con respecto a una estructura $f\ t_1 \dots\ t_n$ si

$$(f\ t_1 \dots\ t_n)^* = f^*(t_1^* \dots\ t_n^*)$$

donde f^* (en GF: `lin f`) es la función definida en la sintaxis concreta encargada de linealizar los árboles generados por el constructor f. De esta definición podemos observar que la linealización de un árbol solo podrá utilizar la linealización de los subárboles inmediatos. Con esto en mente, usaremos las funciones de transferencia cuando se desee representar en GF una operación no composicional, la cual no puede ser interpretada mediante la linealización en GF.

Un fenómeno lingüístico donde las funciones de transferencia son muy útiles es "aggregation"[5]. Un ejemplo es la transformación de "Juan corre y María corre" en "Juan y María corren"; es decir se evita la repetición de un constituyente sintáctico en dos oraciones. Cambios de este tipo suelen ser comunes en generación de lenguaje, cuando se desea que la frase generada sea lo más natural y menos ambigua posible.

Para usar funciones de transferencia en el intérprete de GF, debemos usar el comando `put_tree` con la opción `-transfer`, indicando la función de transferencia.

```
> p "Juan corre y María corre" | pt -transfer=aggr | l
"Juan y María corren"
```

Veamos la sintaxis abstracta definida para este ejemplo, extraída del libro de GF (Ranta, 2011b, p. 148):

```
cat S ; NP ; VP ;
cat Bool ; data True, False : Bool ;
data
  PredVP : NP -> VP -> S ;
  ConjS  : S -> S -> S ;
  ConjVP : VP -> VP -> VP ;
  ConjNP : NP -> NP -> NP ;
  Run, Walk : VP ;
  John, Mary : NP ;
```

Donde los constructores `Conj<T>` permiten formar un constituyente sintáctico de tipo `T` a partir de la conjunción de dos constituyentes sintácticos de tipo `T`.

```
fun aggr : S -> S ;  -- función de transferencia
def aggr (ConjS (PredVP x X) (PredVP y Y)) =
  ifS (eqNP x y)
    (PredVP x (ConjVP X Y))
    (ifS (eqVP X Y)
       (PredVP (ConjNP x y) X)
       (ConjS (PredVP x X) (PredVP y Y))) ;
fun ifS : Bool -> S -> S -> S ;  -- if b then x else y
```

[5] https://en.wikipedia.org/wiki/Aggregation_(linguistics)

```
def
  ifS True  x _ = x ;
  ifS False _ y = y ;
fun eqNP : NP -> NP -> Bool ;  -- x == y
def
  eqNP John John = True ;
  eqNP Mary Mary = True ;
  eqNP _ _ = False ;
fun eqVP : VP -> VP -> Bool ;  -- X == Y
def
  eqVP Run  Run  = True ;
  eqVP Walk Walk = True ;
  eqVP _ _ = False ;
cat Bool ; data True, False : Bool ;
```

Donde la función `aggr` es la encargada de transformar el árbol de sintaxis abstracto de una oración formada a partir de la conjunción de dos oraciones, en el de una oración sin constituyentes sintácticos repetidos, usando los constructores `ConjNP` y `ConjVP`.

Utilizando la opción `-tr` podemos observar cada AST, antes de realizar la linealización:

```
> p -tr "Juan corre y María corre" | pt -tr -transfer=aggr | l
ConjS (PredVP John Run) (PredVP Mary Run)
PredVP (ConjNP John Mary) Run
"Juan y María corren"
```

El código de este ejemplo nos muestra además que GF es muy primitivo como lenguaje de programación si utilizamos definiciones semánticas. En este caso, nos fue necesario dar una definición propia de condicional `if-then-else` tanto como de igualdad entre constituyentes. Por esto suele ser recomendable usar un lenguaje de programación de uso general que permita importar las gramáticas generadas en GF, como Haskell, para definir las funciones de transferencia.

Capítulo 3

Representación semántica en GF

En este capítulo veremos como podemos usar GF para lograr establecer una traducción entre el castellano y el lenguaje simbólico de fórmulas lógicas de primer orden. Veremos que una primera aproximación puede lograrse sin mucha dificultad, sin embargo, esta traducción no será muy abarcativa en el sentido que solo puede traducir un conjunto reducido de oraciones en castellano que no está muy alejado del lenguaje formal de la lógica. Iremos analizando de manera incremental qué nuevas oraciones podrían ser traducidas y ampliaremos la gramática para conseguirlo cuando sea posible.

3.1 Gramática base

La metodología de este trabajo está basada en (Ranta, 2011a), donde se propone definir primero una gramática que priorice la precisión de la traducción para ir luego refinándola paso a paso y así ampliar la cobertura de las proposiciones. Usaremos la expresión "cobertura de expresiones" para referirnos a las oraciones en castellano que pueden aceptarse usando la gramática. Veamos entonces primero las categorías y constructores que definen la sintaxis abstracta de esta gramática base junto con algunos ejemplos asociados a proposiciones que describirán figuras geométricas usando los predicados definidos en SAT.

Categorías (Sintaxis abstracta)

Prop La categoría que representa cualquier proposición bien formada. Las proposiciones bien formadas son aquellas que luego pueden ser transformadas en fórmulas lógicas de primer orden.

Atom La categoría que representa proposiciones atómicas. Las proposiciones atómicas estarán constituidas por la aplicación de predicados. Un ejemplo de una proposición atómica es "A es un triángulo".

Pred1 La categoría que representa predicados unarios. Un ejemplo de una predicado unario es "cuadrado", que determina si una figura geométrica es cuadrada.

Pred2 La categoría que representa predicados binarios. Un ejemplo de un predicado binario es "arriba de", que determina si una figura geométrica está arriba de otra.

Fun1 La categoría que representa funciones de un parámetro. Si bien en SAT no hay actualmente funciones, podríamos imaginar una función `achicar` que modifique el tamaño de las figuras. Por ejemplo, `Chico(achicar(A))` será cierta si y solo si `Mediano(A)` es cierta. Un ejemplo en el dominio de números reales podría ser la función raíz cuadrada. Una proposición que utiliza esta función es "La raíz cuadrada de 2 es irracional".

Fun2 La categoría que representa funciones de dos parámetros. De manera análoga a `Fun1` podemos imaginar una función `intercambia(x,y)` que intercambie las posiciones de la figuras `x` e `y`. Un ejemplo en el dominio de números reales podría ser la función suma. Una proposición que utiliza esta función es $a + b = b + a$.

Conj La categoría que representa conjunciones, es decir, el conjunto de palabras que conectan proposiciones. La noción de conjunción de proposiciones no debe confundirse con el operador binario de conjunción lógica. En esta categoría encontraremos tanto a la conjunción "y" como la conjunción "o". Los operadores lógicos de condicional y bicondicional no estarán representados por esta categoría. La razón de esto será explicada en la siguiente sección.

Ind La categoría que representa individuos. En SAT cada figura en el tablero es un individuo y está identificada por uno o más nombres que serán palabras de una o dos letras mayúsculas. Ejemplo de nombres de individuos son "A" y "AB".

Var La categoría que representa variables de cuantificación. Ejemplos de variables en SAT son las letras "x" e "y".

La decisión de tener predicados y funciones de aridad 2 como máximo se justifica porque no es habitual encontrar frases en lenguaje natural que utilicen funciones o predicados de mayor aridad.

Constructores (Sintaxis abstracta)

Comenzaremos definiendo la lista de constructores de la gramática que nos permitirán representar la signatura propia de SAT, donde pensamos de manera informal a una signatura como el conjunto de símbolos no lógicos de un lenguaje formal. Diremos que la signatura es el *lexicón* de nuestra gramática.

Si bien SAT posee una única signatura que define figuras geométricas y predicados sobre estas, podemos imaginar una versión más completa en la cual uno pueda definir su propia signatura y generar mundos y predicados que no tengan que ver con figuras geométricas.

En SAT es posible establecer ciertas características de las figuras geométricas, como su color, forma y tamaño. Por ejemplo, se puede definir que una figura es de color roja. Para cada característica posible contamos con un predicado unario que nos permitirá determinar si una figura posee o no dicha característica. Los constructores de nuestra gramática que representan los predicados unarios de SAT son los siguientes.

Color: `Rojo, Azul, Verde: Pred1`

Tamaño: `Chico, Mediano, Grande: Pred1`

Forma: `Triangulo, Cuadrado, Circulo: Pred1`

En SAT también es posible establecer la posición de las figuras en un tablero. Para ello contamos con predicados binarios que nos permiten determinar, por ejemplo, si una figura está arriba de otra. Los constructores de nuestra gramática que representan los predicados binarios disponibles en SAT son los siguientes.

Posición: `Izquierda, Derecha, Arriba, Abajo: Pred2`

Es importante destacar que si uno quisiera podría utilizar esta gramática para un dominio totalmente distinto, como por ejemplo el dominio de números reales, simplemente modificando los constructores del lexicón. Para este caso uno podría definir predicados unarios que determinen, por ejemplo, si un número es par o no, predicados binarios como la igualdad, o la relación de mayor o igual y funciones como el cuadrado o la suma. De esta manera uno podría formalizar proposiciones como "si n es par entonces n^2 es par".

Nos falta listar aún los constructores de la sintaxis abstracta de la gramática base relacionados al lenguaje formal de la lógica. Estos constructores son independientes de cualquier lexicón, por lo que forman el esqueleto de nuestra gramática abstracta.

```
-- Constructores de Prop
PAtom  : Atom  -> Prop
PNeg   : Prop  -> Prop
PConj  : Conj  -> Prop -> Prop -> Prop
PImpl  : Prop  -> Prop -> Prop
PUniv  : Var -> Prop -> Prop -> Prop
PExist : Var -> Prop -> Prop -> Prop
True, False : Prop

-- Constructores de Conj
CAnd, COr : Conj

-- Constructores de Pred2
Igual : Pred2
Diferente : Pred2

-- Constructores de Atom
APred1 : Pred1 -> Ind -> Atom
APred2 : Pred2 -> Ind -> Ind -> Atom

-- Constructores de Ind
IVar   : Var -> Ind
IFun1  : Fun1 -> Ind -> Ind
IFun2  : Fun2 -> Ind -> Ind -> Ind

-- Constructores de Var
VString : String -> Var
```

A continuación una breve descripción de cada uno de estos constructores.

PAtom Construye una proposición de tipo `Prop` en base a una proposición atómica de tipo `Atom`. Simbólicamente será representado como P, donde P es una fórmula lógica de primer orden.

PNeg Negación de una proposición, simbólicamente representado como $\neg P$.

PConj Conjunción de dos proposiciones. Simbólicamente será representado como $(P \square Q)$, con $\square \in \{\wedge, \vee\}$.

PImpl Implicación lógica de dos proposiciones. Simbólicamente será representado como $(P \Rightarrow Q)$.

PEquiv Equivalencia lógica de dos proposiciones. Simbólicamente será representado como $(P \equiv Q)$.

PUniv Cuantificación universal. Simbólicamente será representado como $\langle \forall x : R.x : T.x \rangle$. Denotaremos a R como *rango* de cuantificación y a T como el *término*.

PExist Cuantificación existencial. Simbólicamente será representado como $\langle \exists x : R.x : T.x \rangle$.

True/False Proposiciones básicas. Simbólicamente serán representdos como $True$ y $False$ respectivamente.

Igual/Diferente Predicados binarios de igualdad/desigualdad.

APred1 Aplicación de un predicado unario a un individuo. Simbólicamente será representado como $p.x$, donde p es un predicado unario y x un individuo.

APred2 Aplicación de un predicado binario a dos individuos. Análogamente con el caso unario, a los predicados binarios que determinan posición de figuras los representaremos como $p.x.y$, donde p es un predicado binario y x e y dos individuos. Los predicados `Igual` y `Diferente` son casos especiales y serán representados como $(x = y)$ y $\neg(x = y)$ respectivamente.

IVar Transforma una variable de tipo `Var` en un individuo de tipo `Ind`. Usaremos `IVar` para representar variables dentro de una cuantificación pero también (por comodidad) para dar el nombre de individuos concretos.

IFun1 Aplicación de una función de aridad 1 sobre un individuo. Como en SAT no disponemos de funciones este constructor no será usado. Sin embargo, una posible representación simbólica podría ser $f(x)$.

IFun2 Aplicación de una función de aridad 2 sobre dos individuos. Como en SAT no disponemos de funciones este constructor no será usado. Sin embargo, análogamente con el caso unario, una posible representación podría ser $f(x, y)$.

VString Usado para construir variables desde strings.

Si bien todavía no definimos la sintaxis concreta veamos ejemplos[1]. Definimos para ello primero el comando `translate`, el cual se encargará de traducir proposiciones en lenguaje natural en fórmulas lógicas de primer orden.

```
Logic> dc translate p -tr -lang=Spa ?0 | l -bind -lang=Sym
```

[1]Las fórmulas que devuelve el comando translate son mostradas como comentarios.

```
Logic> %translate "A es rojo"
PAtom (APred1 Rojo (IVar (VString "A")))
# Rojo.A

Logic> %translate "A está arriba de B"
PAtom (APred2 Arriba (IVar (VString "A")) (IVar (VString "B")))
# arriba.A.B

Logic> %translate "A es igual a B"
PAtom (APred2 Igual (IVar (VString "A")) (IVar (VString "B")))
# (A = B)

Logic> %translate "A es triangular y B es grande"
PConj CAnd (PAtom (APred1 Triangulo (IVar (VString "A"))))
           (PAtom (APred1 Grande (IVar (VString "B"))))
# (Tr.A ∧ Grande.B)

Logic> %translate "A es cuadrado o A es chico"
PConj COr (PAtom (APred1 Cuadrado (IVar (VString "A"))))
          (PAtom (APred1 Chico (IVar (VString "B"))))
# (Cuad.A ∨ Chico.A)

Logic> %translate "si A es distinto de B entonces A es mediano"
PImpl (PAtom (APred2 Diferente (IVar (VString "A")) (IVar (VString "B"))))
      (PAtom (APred1 Mediano (IVar (VString "A"))))
# (¬(A = B) ⇒ Mediano.A)

Logic> %translate "B es igual a C si y solo si C es igual a B"
PEquiv (PAtom (APred2 Igual (IVar (VString "B")) (IVar (VString "C"))))
       (PAtom (APred2 Igual (IVar (VString "C")) (IVar (VString "B"))))
# ((B = C) ≡ (C = B))

Logic> %translate "no es cierto que A es verde"
PNeg (PAtom (APred1 Verde (IVar (VString "A"))))
# ¬Verde.A
```

De este último ejemplo podemos observar que para negar una proposición debemos agregar al principio de la misma "no es cierto que". Sin embargo, para proposiciones atómicas como "A es verde" una alternativa más natural de negación sería introducir

la palabra “no” dentro de la proposición y antes del verbo, obteniendo la proposición “A no es verde”.

Agregando el siguiente constructor a la gramática podemos lograr esta versión más natural de negación sobre proposiciones atómicas.

```
-- Negación de proposiciones atómicas
PNegAtom : Atom -> Prop
```

La razón por la cual solamente aplicamos este estilo de negación sobre proposiciones atómicas es para evitar oraciones mal formadas como “A no no es verde”. Veamos un ejemplo de uso de este nuevo constructor.

```
Logic> %translate "A no es verde"
PNegAtom (APred1 Verde (IVar (VString "A")))
# ¬Verde.A
```

Otro hecho que no debería pasar desapercibido es que no dimos un ejemplo de proposición cuya formalización genere una fórmula cuantificada. Si bien Ranta define en su artículo una linealización a lenguaje natural de los constructores `PUniv` y `PExist`, en este trabajo no definiremos esta linealización. Ranta utiliza una versión de fórmulas cuantificadas (con forma $\langle Qx : P.x \rangle$) distinta a la que usaremos en este trabajo en la que sí distinguimos entre rango y término (con forma $\langle Qx : R.x : T.x \rangle$).

En su artículo, Ranta traduce proposiciones como “para todo x, x es rojo” y “hay un elemento x tal que x está arriba de A”. Sin embargo estas frases podrían ser escritas de una manera distinta y más conveniente por su aceptabilidad. Una alternativa a la primera proposición es “todas las figuras son rojas” y una alternativa a la segunda es “hay una figura arriba de A”, sin la necesidad de referir a variables en lenguaje natural.

Esta forma de representar en lenguaje natural proposiciones que se traduzcan a fórmulas cuantificadas es la razón por la cual no definimos una linealización a lenguaje natural de `PUniv` y `PExist`, sino que haremos el esfuerzo de traducir de manera correcta frases como “Todas las figuras son rojas”. A este tipo de proposiciones que no utilizan variables y son transformadas en fórmulas cuantificadas lo denotaremos *cuantificación in-situ*.

Para lograr parsear proposiciones de este tipo en GF agregamos una nueva categoría y nuevos constructores.

```
cat Kind
fun
  Figura : Kind
  UnivIS  : Var -> Kind -> Pred1 -> Prop
  ExistIS : Var -> Kind -> Pred1 -> Prop
  ModKind : Kind -> Pred1 -> Kind
```

La categoría `Kind` representa el tipo de nuestros elementos y será usada como el dominio de cuantificación para los constructores de cuantificación in-situ. Una breve descripción de estos nuevos constructores es la siguiente.

Figura El tipo de los elementos en SAT. Este constructor forma parte del lexicón, junto con los predicados.

UnivIS Cuantificación universal in-situ. El primer parámetro es una variable que no aparecerá en la proposición en lenguaje natural, por lo que GF generará para ella automáticamente una metavariable. El segundo parámetro representará el rango de cuantificación y el tercero representará el término. En el ejemplo "cada figura es grande", `Grande` es el `Pred1` que representa el término. Si bien esto es una limitación nos servirá para lograr una primera aproximación que luego iremos mejorando. Que el término pertenezca a `Pred1` no nos permite, por ejemplo, parsear frases como "todas las figura rojas están arriba de A", donde usamos `Arriba` como un predicado binario que forma parte del término.

ExistIS Cuantificación existencial in-situ. Los parámetros cumplen la misma función que en cuantificación universal. Un ejemplo de una proposición de este tipo es "alguna figura es roja".

ModKind Restricción de dominio de cuantificación. Las restricciones estarán dadas por un predicado unario, lo cual también establece una limitación similar al término de cuantificación, sin embargo, en la próxima sección conseguiremos mayor expresividad con esta misma estructura. Podemos observar un ejemplo de restricción de dominio en la proposición "cada figura grande es roja", donde el predicado `Grande` restringe el dominio de figuras sobre las cuales tiene efecto la cuantificación.

Veamos ejemplos en GF que usen estos constructores.

```
Logic> p -lang=Spa "cada figura es roja"
UnivIS ?1 Figura Rojo
```

Donde `?1` es una metavariable generada por GF.

```
Logic> p -lang=Spa "alguna figura es cuadrada"
ExistIS ?1 Figura Cuadrado

Logic> p -lang=Spa "cada figura grande es azul"
UnivIS ?1 (ModKind Figura Grande) Azul
```

Es importante notar que no definimos directamente una linealización al lenguaje simbólico de fórmulas para los constructores relacionados a cuantificación in-situ. Usaremos una función de transferencia para transformar los AST's generados por estos constructores en AST's generados por los constructores `PUniv` y `PExist` cuya linealización al lenguaje simbólico es directa. La función de transferencia para lograr esto será definida en la próxima sección.

Ahora sí, veamos las categorías lingüísticas de linealización y algunas reglas de linealización para los constructores de la sintaxis abstracta de la gramática base definidos hasta el momento.

3.1.1 Categorías lingüísticas (Sintaxis concreta)

Las proposiciones (`Prop` en la sintaxis abstracta) poseen la categoría lingüística más general que representa una oración (`S` en la RGL).

Las proposiciones atómicas (`Atom` en la sintaxis abstracta) poseen la categoría lingüística que representa una cláusula (`Cl` en la RGL). Los elementos de categoría `Cl` no poseen en GF polaridad ni tiempo definido, es decir, un elemento de esta categoría puede ser usado para definir una oración con polaridad negativa o una oración con polaridad positiva.

Los predicados unarios (`Pred1` en la sintaxis abstracta) poseen la categoría lingüística que representa frases adjetivas (`AP` en la RGL). Generalmente los predicados unarios son usados para determinar si un elemento posee una característica, por lo que la decisión de que su categoría lingüística sea la de frase adjetiva parece ser correcta. Recordemos que los predicados unarios con los que estamos trabajando verifican características de figuras, como el color, forma y tamaño.

Los predicados binarios (`Pred2` en la sintaxis abstracta) tendrán la categoría lingüística `A2` de la RGL, que representa la categoría de "adjetivos binarios". Un ejemplo de un adjetivo binario en la RGL es "divisible" el cual puede aparecer en una frase como "4 es divisible por 2", donde podemos pensar a "divisible por 2" como una frase adjetiva resultado de la aplicación parcial de este adjetivo binario.

Un dato importante es que `Pred2` será la primer categoría cuya linealización estará dada por un record. Necesitamos un record ya que los predicados binarios no se comportan de igual manera al definir su linealización. En la linealización a español algunos predicados estarán acompañados del verbo copulativo "ser", como `Igual` o `Diferente`, mientras que los predicados que determinan la posición de una figura estarán acompañados del verbo copulativo "estar". Usar un record que junte un componente lingüístico (`A2` en este caso) e información del tipo de predicado nos permitirá ajustar de manera adecuada la linealización. Veamos la categoría de linealización que usaremos para `Pred2` en GF

```
param CopC = SerC | EstarC ;
lincat Pred2 = {t : CopC ; s : A2} ;
```

donde usamos el tipo enumerado `CopC` para representar los posibles tipos de `Pred2`. Como ejemplo, el predicado `Arriba` se linealizará como `{t = EstarC ; s = "arriba"}`, mientras que `Igual` se linealizará como `{t = SerC ; s = "igual"}`.

Se listan a continuación las categorías de linealización para los otros constructores de la gramática abstracta:

- Las funciones de ambas aridades (`Fun1` y `Fun2` en la sintaxis abstracta) poseen la categoría lingüística de sustantivo relacional (`N2` en la RGL). Un ejemplo de un elemento de categoría (`N2`) en GF podría ser "hermano de".

- Los individuos (`Ind` en la sintaxis abstracta) poseen la categoría lingüística de sintagma nominal (`NP` en la RGL).

- Los tipos de individuo (`Kind` en la sintaxis abstracta) poseen la categoría lingüística de sustantivo común (`CN` en la RGL). Recordemos que en nuestro caso elementos de de esta categoría serán "figura" y modificaciones mediante adjetivos de este sustantivo como "figura roja".

- Las conjunciones (`Conj` en la sintaxis abstracta) poseen la categoría lingüística propia de conjunción (`SyntaxSpa.Conj` en la RGL).

- Las variables (`Var` en la sintaxis abstracta) no poseen una categoría lingüística asociada.

3.1.2 Reglas de linealización (Sintaxis concreta)

Veamos ahora algunas reglas de linealización definidas en la sintaxis concreta del español usando la RGL.

```
lin
  UnivIS v k p = mkS (mkCl (mkNP every_Det k) p) ;
  ExistIS v k p = mkS (mkCl (mkNP someSg_Det k) p) ;
  ModKind k p = mkCN p k ;
```

Para interpretar estas definiciones debemos tener en mente que `k` pertenece a la categoría `CN` (sustantivos comunes) y que `p` pertenece a la categoría `AP` (frases adjetivas). En GF es posible definir funciones mediante polimorfismo ad-hoc, es decir, definir una función que posea distintas signaturas y se comporte distinto de acuerdo al tipo de sus argumentos. Las funciones `mkT`, para alguna categoría `T`, están definidas mediante polimorfismo ad-hoc. Es importante que veamos la signatura de las funciones `mkS`, `mkCl`, `mkNP` y `mkCN` que nos provee la interfaz de la RGL.

Para la linealización de los constructores de cuantificación in-situ la signatura de las funciones utilizadas es la siguiente.

```
mkS : Cl -> S
mkCl : NP -> AP -> Cl
mkNP : Det -> CN -> NP
mkCN : AP -> CN -> CN
```

Así podemos obtener la oración "cada figura es roja" mediante la aplicación de `(lin UnivIS)` con `k = "figura"` y `p = "roja"`. Notemos que un posible valor para `k` es la linealización de un modificador de clase (`ModKind`); de esta manera podemos parsear frases como "alguna figura grande es azul", donde el sustantivo `k = "figura"` es modificado por el adjetivo `p = "grande"`.

La regla de linealización de `APred1` usa la misma versión de `mkCl` que vimos en las reglas de linealización de cuantificación in-situ.

```
lin APred1 f x = mkCl x f ;
```

De esta manera podemos generar la oración "A es circular" mediante la aplicación de `(lin APred1)` con `f = "circular"` y `x = "A"`.

Para los predicados binarios debemos establecer en la regla de linealización que usaremos el verbo copulativo "estar" para los predicados de tipo posicional y el verbo copulativo "ser" para el resto. GF nos provee construcciones propias del idioma español en un archivo llamado ExtraSpa.gf, del cual usaremos la función `UseComp_estar` para lograr parsear frases como "A está arriba de B".

```
-- (E = ExtraSpa)
APred2 p x y = case p.t of {
  EstarC => mkCl x (E.UseComp_estar (mkComp (mkAP p.s y))) ;
  _ => mkCl x p.s y
} ;
```

donde `mkComp` nos permite crear un complemento necesario para el verbo copulativo "estar" a partir de una frase adjetiva.

Por último, veamos las reglas de linealización de `PConj` y `PNegAtom`

```
CAnd = and_Conj ;
PConj = mkS ;
PNegAtom = mkS negativePol ;
```

donde `PConj = mkS` es equivalente (por η-conversión) a `PConj c p1 p2 = mkS c p1 p2`.

En estas definiciones se hace evidente el polimorfismo ad-hoc de la función `mkS`. En la definición de la linealización de `PConj` la signatura de `mkS` es

```
mkS : Conj -> S -> S -> S
```

permitiendo parsear frases como "A es rojo y B es azul", mientras que en la definición de la linealización de `PNegAtom` la signatura de `mkS` es

```
mkS : Pol -> Cl -> S
```

permitiendo parsear frases como "A no es verde".

3.2 Refinamiento de la gramática base

Como ya vimos, hemos definido una versión de cuantificación no muy usual desde una perspectiva simbólica, y más cercana al lenguaje natural, en la que el rango y término se ven restringidos a la aplicación de predicados unarios y sin una noción explícita de variable de cuantificación. En esta sección extenderemos la gramática con constructores que nos permitirán parsear oraciones más "naturales" bajo la noción de aceptabilidad descrita en (Luque, 2017).

Los constructores que agreguemos para facilitar el parseo de oraciones más naturales no deben afectar el estilo conciso del lenguaje simbólico, por lo que no buscaremos linealizarlos directamente. Como explicamos en el capítulo anterior, usaremos funciones de transferencia para transformar los ASTs que usen los nuevos constructores en ASTs que sólo utilicen constructores que sí linealizamos en el lenguaje simbólico.

Veamos la definición de la función de transferencia para los constructores de cuantificación in-situ:

```
fun transfer : Prop -> Prop ;
def
  transfer (UnivIS v k p) = univIStoP (UnivIS v k p) ;
  transfer (ExistIS v k p) = existIStoP (ExistIS v k p) ;
```

`transfer` será la función de transferencia principal que usaremos para cada constructor nuevo que definamos. La definición de esta función será por pattern matching sobre la estructura de los nuevos constructores y en cada caso se hará uso de una función auxiliar que se encargará de realizar la transformación adecuada.

Veamos las funciones auxiliares para los constructores de cuantificación in-situ.

```
fun
  univIStoP : Prop -> Prop ;
  existIStoP : Prop -> Prop ;
  kindToProp : Kind -> Var -> Prop ;

def
  univIStoP (UnivIS v Figura p) = PUniv v True (PAtom (APred1 p (IVar v))) ;
  univIStoP (UnivIS v k p) = PUniv v (kindToProp k v) (PAtom (APred1 p (IVar v))) ;

def
  existIStoP (ExistIS v Figura p) = PExist v True (PAtom (APred1 p (IVar v))) ;
  existIStoP (ExistIS v k p) = PExist v (kindToProp k v) (PAtom (APred1 p (IVar v))) ;

def
  kindToProp (ModKind Figura p) v = (PAtom (APred1 p (IVar v))) ;
  kindToProp (ModKind k p) v = PConj CAnd (PAtom (APred1 p (IVar v)))
                                          (kindToProp k v) ;
```

Remarquemos dos aspectos de estas funciones:

1. Si una proposición de cuantificación in-situ se parsea con `Figura` como `Kind` entonces se traducirá en una fórmula con rango `True`. Un ejemplo es el de la frase "alguna figura es roja" la cual se traduce en la fórmula $\langle \exists x : True : Rojo.x \rangle$

2. La modificación de elementos de tipo `Kind` aplicando sucesivas veces `ModKind` se traduce en una fórmula con un rango formado por la conjunción de los predicados unarios (`Pred1`) utilizados para la modificación. Un ejemplo en el que se puede observar esto es "alguna figura cuadrada grande es azul", la cual se traduce en $\langle \exists x : Cuad.x \wedge Grande.x : Azul.x \rangle$. Para este último ejemplo una frase más aceptable sería "alguna figura cuadrada y grande es azul", introduciendo una conjunción entre los adjetivos. Uno de los nuevos constructores que definiremos más adelante nos permitirá establecer conjunción de predicados para lograr parsear frases con esta estructura.

Como ya vimos, una limitación de los constructores de cuantificación in-situ es el uso de predicados unarios tanto en rango como en término. Por ejemplo, no es posible tener proposiciones con términos que incluyan predicados binarios, conjunción de varios predicados o cuantificadores "anidados". Resolveremos estas limitaciones definiendo nuevos constructores.

3.2.1 Aplicación parcial de predicados binarios

Eliminaremos la limitación de no poder usar predicados binarios en la cuantificación in-situ definiendo un constructor que nos permita realizar aplicación parcial de predicados binarios con un individuo, generando así nuevos predicados unarios.

```
PartPred : Pred2 -> Ind -> Pred1 ;
```

Con este nuevo constructor podremos parsear frases como "todas las figuras rojas están arriba de A", donde la aplicación parcial del predicado binario `Arriba` genera el predicado unario que define el término de cuantificación.

La regla de linealización de `PartPred` hace uso de la función `mkAP` de la RGL, la cual tiene la signatura

```
mkAP : A2 -> NP -> AP
```

Un hecho importante en la linealización es que al definir predicados unarios en base a predicados binarios debemos propagar el tipo `CopC`, por las mismas razones que dimos cuando definimos el tipo de linealización de `Pred2`. La nueva categoría de linealización de `Pred1` será la siguiente

```
lincat Pred1 = {t : CopC ; s : AP} ;
```

y la regla de linealización de `PartPred` se encargará de propagar el tipo del predicado binario.

```
PartPred p i = {t = p.t ; s = mkAP p.s i} ;
```

La linealización de `APred1` será por casos y análoga a la linealización de `APred2`. Veamos algunos ejemplos de uso de aplicación parcial en cuantificación in-situ.

```
Logic> dc translate p -tr -lang=Spa ?0 | pt -tr -transfer=transfer |
       l -bind -lang=Sym
```

```
Logic> %translate "alguna figura roja está arriba de A"
ExistIS ?1 (ModKind Figura Rojo) (PartPred Arriba (IVar (VString "A")))
PExist ?1 (PAtom (APred1 Rojo (IVar ?1)))
          (PAtom (APred2 Arriba (IVar ?1) (IVar (VString "A"))))
# ⟨∃?1 : Rojo.?1 : arriba.?1.A⟩ .

Logic> %translate "cada figura a la izquierda de A es grande"
UnivIS ?1 (ModKind Figura (PartPred Izquierda (IVar (VString "A")))) Grande
PUniv ?1 (PAtom (APred2 Izquierda (IVar ?1) (IVar (VString "A"))))
         (PAtom (APred1 Grande (IVar ?1)))
# ⟨∀?1 : izq.?1.A : Grande.?1⟩ .
```

Recordemos que `?1` es una metavariable generada por GF. Un post-procesamiento debería encargarse de generar variables concretas como "x" e "y" a partir de estas metavariables. De aquí en adelante las metavariables serán transformadas en variables concretas para representar fórmulas cuantificadas.

Para proposiciones atómicas usaremos una función auxiliar `trAtom` que nos será útil para definir la función de transferencia. Veamos el caso de la función `trAtom` para aplicación parcial de predicados.

```
def
  transfer (PAtom pa) = trAtom (PAtom pa) ;
  transfer (PNegAtom pa) = trAtom (PNegAtom pa) ;

fun trPred2 : Pred2 -> Ind -> Ind -> Prop ;
def
  trPred2 p i1 i2 = trAtom (PAtom (APred2 p i1 i2)) ;

fun trAtom : Prop -> Prop ;
def
  trAtom (PAtom (APred1 (PartPred p i2) i1)) = trPred2 p i1 i2 ;
  trAtom (PNegAtom (APred1 (PartPred p i2) i1)) = PNeg (trPred2 p i1 i2) ;

  trAtom (PAtom pa) = PAtom pa ;
  trAtom (PNegAtom pa) = PNegAtom pa ;
```

La llamada recursiva de `trAtom` usando `trPred2` en los primeros dos casos de la función `trAtom` se debe a que la aplicación del predicado binario podría requerir una nueva transformación. Esto se hará más evidente cuando definamos conjunción de individuos, con lo que podremos aceptar frases como "alguna figura roja está arriba de B y C", donde será necesario usar la función de transferencia para transformar la aplicación parcial y la conjunción de individuos.

3.2.2 Conjunción de individuos

Como anticipamos, otra posible mejora a la gramática base es agregar conjunción de individuos. Esta conjunción nos permite referirnos con un mismo predicado a más de un individuo. Ejemplos en castellano de conjunción de individuos pueden observarse en frases como "A y B son rojos" o "A está arriba de B y C".

Recordemos que los únicos constructores de la categoría `Conj` son `CAnd` y `COr`, dejando afuera otros operadores binarios lógicos como la implicación o la equivalencia. La razón de esto es que usaremos a los constructores de `Conj` como operadores polimórficos. Hasta aquí solo hemos usado a estos constructores como conjunción de proposiciones, es decir, con tipo `Prop -> Prop -> Prop`. La conjunción de individuos nos permite pensar en una versión polimórfica de conjunción, cuyo tipo sea además `Ind -> Ind -> Ind`.

Definiremos un nuevo constructor, el cual nos permitirá parsear frases con conjunción de individuos. Para esto introducimos primero *listas de categorías*, un tipo de categorías de GF que no usamos aún. Las listas de categorías nos permiten definir, dada una categoría `C`, una categoría de tipo `[C]` que estará formada por elementos de tipo `C`. GF nos permite definir un número mínimo `n` de elementos que puede tener una lista de tipo `[C]` mediante la notación `cat [C] {n}`. Una categoría definida mediante `cat [C] {n}` nos provee automáticamente dos constructores, llamados `BaseC` y `ConsC` con la signatura

```
BaseC : C -> ... -> C -> [C]
ConsC : C -> [C] -> [C]
```

donde `BaseC` toma `n` elementos de tipo `C` para asegurarse construir una lista de al menos `n` elementos.

Definamos ahora sí la nueva categoría y el constructor `ConjInd` usando listas de categorías para aceptar frases con conjunción de al menos dos individuos

```
cat [Ind] {2} ;
fun ConjInd : Conj -> [Ind] -> Ind ;
```

Para la linealización de cada `ConjInd` usaremos la función `mkNP` de la RGL, esta vez con la siguiente signatura

```
mkNP : Conj -> [NP] -> NP
```

y la función `mkListNP` la cual nos permitirá crear listas de sintagmas nominales, con las siguientes signaturas:

```
mkListNP : NP -> NP -> [NP]
mkListNP : NP -> [NP] -> [NP]
```

Definamos ahora la linealización de `ConjInd`.

```
lincat [Ind] = [NP] ;
lin
  BaseInd = mkListNP ;
  ConsInd = mkListNP ;

  ConjInd = mkNP ;
```

Veamos algunos ejemplos de formalización de proposiciones usando conjunción de individuos:

```
Logic> %translate "A y B son rojos"
PAtom (APred1 Rojo (ConjInd CAnd (BaseInd (IVar (VString "A"))
  (IVar (VString "B")))))
PConj CAnd (PAtom (APred1 Rojo (IVar (VString "A"))))
           (PAtom (APred1 Rojo (IVar (VString "B"))))
# (Rojo.A ∧ Rojo.B)

Logic> %translate "A y B están a la derecha de C"
PAtom (APred2 Derecha (ConjInd CAnd (BaseInd (IVar (VString "A"))
  (IVar (VString "B")))) (IVar (VString "C")))
PConj CAnd (PAtom (APred2 Derecha (IVar (VString "A")) (IVar (VString "C"))))
           (PAtom (APred2 Derecha (IVar (VString "B")) (IVar (VString "C"))))
# (der.A.C ∧ der.B.C)

Logic> %translate "A está abajo de B o C"
PAtom (APred2 Abajo (IVar (VString "A")) (ConjInd COr (BaseInd
  (IVar (VString "B")) (IVar (VString "C")))))
PConj COr (PAtom (APred2 Abajo (IVar (VString "A")) (IVar (VString "B"))))
            (PAtom (APred2 Abajo (IVar (VString "A")) (IVar (VString "C"))))
# (abajo.A.B ∨ abajo.A.C)

Logic> %translate "A y B están arriba de C y D"
PAtom (APred2 Arriba (ConjInd CAnd (BaseInd (IVar (VString "A")) ...
PConj CAnd (PConj CAnd (PAtom (APred2 Arriba (IVar (VString "A")) ...
# ((arriba.A.C ∧ arriba.A.D) ∧ (arriba.B.C ∧ arriba.B.D))
```

Ahora, actualicemos la función de transferencia para conjunción de individuos

```
-- Aplicación de Pred1 con posible transformación
fun trPred1 : Pred1 -> Ind -> Prop ;
def
  trPred1 p i = trAtom (PAtom (APred1 p i)) ;

-- Aplicación de Pred1 a lista de individuos
fun trConjInd1 : Pred1 -> Conj -> [Ind] -> Prop ;
def
  trConjInd1 p c (BaseInd i1 i2) = PConj c (trPred1 p i1) (trPred1 p i2) ;
  trConjInd1 p c (ConsInd i li) = PConj c (trPred1 p i) (trConjInd1 p c li) ;

def
  trAtom (PAtom (APred1 p (ConjInd c li))) = trConjInd1 p c li ;
```

Los casos de negación de proposiciones con conjunción de individuos y los casos de aplicación de predicados binarios no son listados en la definición de `transAtom` pero siguen los siguientes criterios:

- La negación de una frase con un predicado unario p e individuos $i_1, i_2, \ldots, i_n$ unidos por una conjunción $C \in \{\wedge, \vee\}$ se traducirá en la siguiente fórmula (Suponiendo $C = \wedge$)

$$\bigwedge_{j=1}^{n} \neg p.i_j$$

  El uso de esta regla se puede ver en los siguientes ejemplos

```
Logic> %translate "A y B no son cuadrados"
PNegAtom (APred1 Cuadrado (ConjInd CAnd (BaseInd (IVar (VString "A"))
  (IVar (VString "B")))))
PConj CAnd (PNegAtom (APred1 Cuadrado (IVar (VString "A"))))
           (PNegAtom (APred1 Cuadrado (IVar (VString "B"))))
# (¬Cuad.A ∧ ¬Cuad.B)

Logic> %translate "A o B no es azul"
PNegAtom (APred1 Azul (ConjInd COr (BaseInd (IVar (VString "A"))
  (IVar (VString "B")))))
PConj COr (PNegAtom (APred1 Azul (IVar (VString "A"))))
          (PNegAtom (APred1 Azul (IVar (VString "B"))))
# (¬Azul.A ∨ ¬Azul.B)
```

- La aplicación de un predicado binario p sobre una lista de individuos $i_1, i_2, \ldots, i_n$ unidos por la conjunción C_1 y $i_{n+1}, i_{n+2}, \ldots, i_{n+m}$ unidos por la conjunción C_2 se traducirá en la siguiente fórmula (Suponiendo $C_1 = \vee$ y $C_2 = \wedge$)

$$\bigvee_{j=1}^{n}(\bigwedge_{k=n+1}^{n+m} p.i_j.i_k)$$

  El uso de esta regla se puede ver en los siguientes ejemplos

```
Logic> %translate "A o B está arriba de C y D"
# C_1 = ∨ y C_2 = ∧
# ((arriba.A.C ∧ arriba.A.D) ∨ (arriba.B.C ∧ arriba.B.D))

Logic> %translate "A y B están arriba de C y D"
# C_1 = ∧ y C_2 = ∧
# ((arriba.A.C ∧ arriba.A.D) ∧ (arriba.B.C ∧ arriba.B.D))
```

- La negación de la aplicación de un predicado binario p sobre una lista de individuos $i_1, i_2, \ldots, i_n$ unidos por la conjunción C_1 y $i_{n+1}, i_{n+2}, \ldots, i_{n+m}$ unidos por la conjunción C_2 se traducirá en la siguiente fórmula (Suponiendo $C_1 = \vee$ y $C_2 = \wedge$)

$$\bigvee_{j=1}^{n}\neg(\bigwedge_{k=n+1}^{n+m} p.i_j.i_k)$$

  El uso de esta regla se puede ver en los siguientes ejemplos

```
Logic> %translate "A y B no están arriba de C"
# (¬arriba.A.C ∧ ¬arriba.B.C)

Logic> %translate "A y B no están arriba de C y D"
# (¬(arriba.A.C ∧ arriba.A.D) ∧ ¬(arriba.B.C ∧ arriba.B.D))
```

3.2.3 Distributividad de Predicados binarios

Otro fenómeno encontrado en oraciones en castellano que involucra listas de individuos y predicados binarios puede observarse en frases como “Luis, Carlos y Mercedes son amigos”. En esta frase, “amigo” puede ser visto como un predicado binario y se distribuye a cada par de individuos, es decir, uno puede concluir que “Luis y Carlos son amigos”, “Luis y Merecedes son amigos” y “Carlos y Mercedes son amigos”.

Sin embargo, no siempre será posible construir frases de este estilo que tengan sentido. Por ejemplo, la frase "A, B y C están arriba entre sí" no tiene sentido. Pareciera que es necesario que se cumpla una condición de simetría, si pensamos a los predicados binarios como una relación binaria.

Para determinar si un predicado binario es "distributivo" crearemos una nueva categoría dependiente `Distr` (usando la noción de tipos dependientes descrita en 2.6.1) de la categoría `Pred2`. Un elemento de tipo `Distr p`, para un predicado binario `p`, será una prueba de que `p` es un predicado que puede ser distribuido. Marcaremos como distributivos a los predicados `Igual` y `Diferente` ya que sí tiene sentido aceptar frases como "A, B y C son iguales" o "A, B y C son distintos", sin embargo, no marcaremos como distributivos a los predicados binarios de posición. Veamos como lograr esto en GF.

```
cat
  Distr Pred2 ;
fun
  distr_Igual : Distr Igual ;  -- Prueba de que Igual es "distributivo"
  distr_Dif : Distr Diferente ;  -- Prueba de que Diferente es "distributivo"
  APred2Distr : (p : Pred2) -> Distr p -> [Ind] -> Atom ;
```

Podemos ver que para poder usar un predicado binario `p` en `APred2Distr` debe existir un elemento de tipo `Distr p`.

La regla de linealización de `APred2Distr` es similar a la aplicación de predicados unarios sobre listas de individuos y utiliza la función `mkAP` con la siguiente signatura

```
mkAP : A2 -> AP
```

y además obliga a que los individuos esten enlazados por la conjunción "y".

```
APred2Distr p _ li = mkCl (mkNP and_Conj li) (mkAP p.s) ;
```

Veamos un ejemplo de formalización usando esta regla

```
Logic> %translate "A, B y C son iguales"
PAtom (APred2Distr Igual distr_Igual (ConsInd (IVar (VString "A"))
  (BaseInd (IVar (VString "B")) (IVar (VString "C")))))
PConj CAnd (PConj CAnd (PAtom (APred2 Igual (IVar (VString "A"))
  (IVar (VString "B")))) ...
# (((A = B) ∧ (A = C)) ∧ (B = C))
```

El caso correspondiente de `APred2Distr` en la función de transferencia debe encargarse de generar cada uno de los posibles pares de la lista de individuos.

```
-- Funciones para distribuir predicados binarios en una lista de individuos
fun distrBinPred : Pred2 -> Ind -> [Ind] -> Prop ;
fun distrBin : Pred2 -> [Ind] -> Prop ;

def
  distrBinPred p x (BaseInd i1 i2) = PConj CAnd (trPred2 p x i1) (trPred2 p x i2) ;
  distrBinPred p x (ConsInd i li) = PConj CAnd (trPred2 p x i) (distrBinPred p x li) ;

  distrBin p (BaseInd i1 i2) = trPred2 p i1 i2 ;
  distrBin p (ConsInd i li) = PConj CAnd (distrBinPred p i li) (distrBin p li) ;

  trAtom (PAtom (APred2Distr p _ li)) = distrBin p li ;
```

La definición de la función de transferencia para el caso `PNegAtom (APred2Distr p _ li))` no es mostrado, pero sigue el siguiente criterio. La negación de una frase con predicado p y lista de individuos $i_1, i_2, \ldots, i_n$ se traducirá en la fórmula

$$\bigwedge_{1 \leq j < k \leq n} \neg p.i_j.i_k$$

El uso de esta regla se puede ver en el siguiente ejemplo

```
Logic> %translate "A, B y C no son iguales"
# ((¬(A = B) ∧ ¬(A = C)) ∧ ¬(B = C))
```

3.2.4 Conjunción de predicados

Otra observación que podemos hacer sobre la gramática base es que para definir proposiciones con varios predicados sobre un mismo individuo debemos repetir el nombre del individuo una vez por cada predicado, por ejemplo, en frases como “A es rojo y A es grande”. Sin embargo, en español es mucho más común agrupar características de un individuo usando conjunciones, reemplazando la oración anterior por “A es rojo y grande” . Un ejemplo de este fenómeno puede verse en la siguiente frase del clásico cuento “Platero y yo”:

> Platero es pequeño, peludo, suave; tan blando por fuera, que se diría todo de algodón, que no lleva huesos.

La frase “Platero es pequeño, peludo y suave” agrupa tres características que describen a Platero. En cambio, para dar tres descripciones sobre un individuo con una fórmula lógica debe repetirse el individuo una vez por cada predicado, por ejemplo, con la fórmula $peq.Platero \wedge pel.Platero \wedge suave.Platero$

La conjunción de predicados, al igual que la conjunción de individuos, nos permitirá pensar a los elementos de `Conj` como operadores polimórficos, esta vez con el tipo `Pred1 -> Pred1 -> Pred1`.

Definamos el constructor y la nueva categoría para aceptar frases con conjunción de predicados

```
cat [Pred1] {2} ;
fun ConjPred1 : Conj -> [Pred1] -> Pred1 ;
```

donde establecemos que una lista de predicados tendrá al menos 2 elementos. Además, podemos notar que solo definimos listas de predicados unarios, lo que no es una restricción ya que anteriormente definimos aplicación parcial de predicados binarios.

Sin embargo, sí será un problema el hecho que estamos juntando predicados de distinto tipo en una misma lista. Recordemos que en la sintaxis concreta habíamos establecido tipos de predicados para poder linealizarlos de manera adecuada. Idealmente, quisiéramos tener frases de la pinta “A es rojo y grande” o “A está arriba de B y abajo de C”, donde cada frase contenga solo predicados del mismo tipo. Por ejemplo, podemos juntar los predicados `Rojo` e `Igual`, ya que ambos irán acompañados del verbo copulativo “ser”, parseando por ejemplo la frase “A es rojo e igual a B”. Para conseguir esto, debemos modificar la categoría de predicados en la sintaxis abstracta, convirtiéndola en un tipo dependiente en el valor de una nueva categoría que representará los posibles verbos copulativos “ser” y “estar”. Veamos como conseguir esto en GF.

```
cat
  Cop ;  -- Categoría que representa verbos copulativos
  Pred1 Cop ;  -- Tipo dependiente
  Pred2 Cop ;
data
  Ser, Estar : Cop ;
  ConjPred1Ser : Conj -> [Pred1 Ser] -> Pred1 Ser ;
  ConjPred1Estar : Conj -> [Pred1 Estar] -> Pred1 Estar ;
```

Luego de esta modificación debemos actualizar los constructores que utilicen predicados. Veamos la actualización de algunos constructores del lexicón:

```
Rojo, Azul, Verde : Pred1 Ser ;
Arriba, Abajo, Izquierda, Derecha : Pred2 Estar ;
```

y de otros constructores como `Igual`, `Diferente` y `PartPred`:

```
Igual, Diferente : Pred2 Ser ;
PartPred : (c : Cop) -> Pred2 c -> Ind -> Pred1 c ;
```

Para la linealización de cada `ConjPred1` usaremos nuevamente la función `mkAP` de la RGL, esta vez con la siguiente signatura

```
mkAP : Conj -> [AP] -> AP
```

y la función `mkListAP` la cual nos permitirá crear listas de adjetivos, con las siguientes signaturas

```
mkListAP : AP -> AP -> [AP]
mkListAP : AP -> [AP] -> [AP]
```

Definamos ahora sí la linealización de `ConjPred1Ser` y `ConjPred1Estar`.

```
lincat [Pred1] = [AP] ;
lin
  BasePred1 _ p1 p2 = mkListAP p1.s p2.s ;
  ConsPred1 _ p lp = mkListAP p.s lp ;

  ConjPred1Ser c lp = {t = SerC ; s = mkAP c lp}
  ConjPred1Estar c lp = {t = EstarC ; s = mkAP c lp}
```

Veamos algunos ejemplos de formalización de proposiciones usando conjunción de predicados

```
Logic> %translate "A es rojo y grande"
PAtom (APred1 (ConjPred1Ser CAnd (BasePred1 Ser Rojo Grande))
  (IVar (VString "A")))
PConj CAnd (PAtom (APred1 Rojo (IVar (VString "A"))))
           (PAtom (APred1 Grande (IVar (VString "A"))))
# (Rojo.A ∧ Grande.A)

Logic> %translate "A es cuadrado y distinto de B"
PAtom (APred1 (ConjPred1Ser CAnd (BasePred1 Ser Cuadrado
  (PartPred Diferente (IVar (VString "B"))))) (IVar (VString "A")))
PConj CAnd (PAtom (APred1 Cuadrado (IVar (VString "A"))))
           (PAtom (APred2 Diferente (IVar (VString "A")) (IVar (VString "B"))))
# (Cuad.A ∧ ¬(A = B))
```

```
Logic> %translate "A es cuadrado, verde y grande"
PAtom (APred1 (ConjPred1Ser CAnd (ConsPred1 Ser Cuadrado
  (BasePred1 Verde Grande))) (IVar (VString "A")))
PConj CAnd (PAtom (APred1 Cuadrado (IVar (VString "A"))))
           (PConj CAnd (PAtom (APred1 Verde (IVar (VString "A"))))
                       (PAtom (APred1 Grande (IVar (VString "A")))))
# (Cuad.A ∧ (Verde.A ∧ Grande.A))

Logic> %translate "A está arriba de B o abajo de C"
PAtom (APred1 (ConjPred1 COr (BasePred1 (PartPred Arriba (IVar (VString "B")))
  (PartPred Abajo (IVar (VString "C"))))) (IVar (VString "A")))
PConj COr (PAtom (APred2 Arriba (IVar (VString "A")) (IVar (VString "B"))))
         (PAtom (APred2 Abajo (IVar (VString "A")) (IVar (VString "C"))))
# (arriba.A.B ∨ abajo.A.C)
```

La función de transferencia para conjunción de predicados es construido a continuación. Describimos una versión simplificada en la que los predicados no son dependientes de la categoría `Cop`.

```
fun trConjPred1 : Conj -> [Pred1] -> Ind -> Prop ;
def
  trConjPred1 c (BasePred1 p1 p2) i = PConj c (trPred1 p1 i) (trPred1 p2 i) ;
  trConjPred1 c (ConsPred1 p lp) i = PConj c (trPred1 p i) (trConjPred1 c lp i) ;

def
  trAtom (PAtom (APred1 (ConjPred1 c lp) i)) = trConjPred1 c lp i ;
```

Más extensiones...

A continuación describimos brevemente otras extensiones implementadas en el trabajo que, como las anteriores, también facilitan el parseo de una mayor cantidad de oraciones en castellano.

- Se introdujo la categoría `[Prop]` para representar listas de proposiciones. De esta manera es posible listar más de dos proposiciones en una misma oración sin tener que repetir la conjunción para unir cada proposición. En vez de escribir "A es azul y B es verde y C es cuadrado", podremos ahora escribir la frase "A es azul, B es verde y C es cuadrado".

  El constructor en la sintaxis abstracta utilizado para esto es el siguiente

```
fun PConjs : Conj -> [Prop] -> Prop ;
```

Veamos un ejemplo de traducción usando este constructor

```
Logic> %translate "A es azul, B es verde y C es cuadrado"
PConjs CAnd (ConsProp (PAtom (APred1 Azul (IVar (VString "A"))))
  (BaseProp (PAtom (APred1 Verde (IVar (VString "B"))))) ...
PConj CAnd (PAtom (APred1 Azul (IVar (VString "A")))) (PConj CAnd ...
# (Azul.A ∧ (Verde.B ∧ Cuad.C))
```

- Se agregó un constructor usado para representar la aplicación de un mismo individuo sobre un predicado binario. Esto nos permitirá aceptar frases como "A es igual a sí" o "A no está arriba de sí".

  El constructor en la sintaxis abstracta utilizado para esto es el siguiente

  ```
  fun APredRefl : Pred2 -> Ind -> Atom ;
  ```

  Veamos ejemplos de traducción usando este constructor

  ```
  Logic> %translate "A no está arriba de sí"
  PNegAtom (APredRefl Arriba (IVar (VString "A")))
  PNegAtom (APred2 Arriba (IVar (VString "A")) (IVar (VString "A")))
  # ¬arriba.A.A
  ```

  Una observación graciosa al parsear la frase anterior es que para nuestra gramática esta frase es ambigua, generando también la siguiente traducción

  ```
  Logic> %translate "A no está arriba de sí"
  PNegAtom (APred2 Arriba (IVar (VString "A")) (IVar (VString "sí")))
  PNegAtom (APred2 Arriba (IVar (VString "A")) (IVar (VString "sí")))
  # ¬arriba.A.sí
  ```

  que para el caso particular de SAT, en el cual los individuos solo están formados por una o dos letras mayúsculas, no es una fórmula válida.

- Se reemplazó el uso de la palabra "cada" en cuantificación universal in-situ por frases que usen la palabra "todas". Veamos algunos ejemplos que hacen uso de este reemplazo:

```
Logic> %translate "todas las figuras azules son circulares"
# ⟨∀x : Azul.x : Circ.x⟩
```

También es posible omitir la palabra "todas", aceptando frases como "las figuras chicas son cuadradas".

```
Logic> %translate "las figuras chicas son cuadradas"
# ⟨∀x : Chico.x : Cuad.x⟩
```

- Otra limitación de la gramática definida hasta el momento es que no puede parsear frases que se traduzcan en una fórmula cuantificada con rango o término negado. Esto se debe a que tanto el rango como término de una fórmula se genera a partir de predicados unarios. Para resolver esto, se agregó una nueva categoría `Pol` en la sintaxis abstracta con constructores `P` y `N`, para denotar polaridad positiva y negativa respectivamente. Los predicados unarios serán dependientes de esta nueva categoría, por lo que podemos pensar que ahora existen predicados positivos y negativos.

  Con predicados negativos podremos parsear frases con rango y término negado y además, al tener conjunción de predicados negativos aceptaremos frases que utilicen la conjunción "ni", por ejemplo, en frases como "A no es rojo, azul, ni grande". Veamos algunos ejemplos usando predicados negativos

  ```
  Logic> %translate "A no es rojo, azul ni grande"
  # (¬ Rojo.A ∧ (¬ Azul.A ∧ ¬ Grande.A))
  ```

  ```
  Logic> %translate "todas las figuras que no son rojas son grandes"
  # ⟨∀x : ¬Rojo.x : Grande.x⟩
  ```

  ```
  Logic> %translate "las figuras que no son rojas ni verdes son azules"
  # ⟨∀x : (¬Rojo.x ∧ ¬Verde.x) : Azul.x⟩
  ```

  ```
  Logic> %translate "alguna figura que está arriba de A no es chica"
  # ⟨∃x : arriba.x.A : ¬Chico.x⟩
  ```

  Podemos observar que en proposiciones de cuantificación in-situ tanto la negación como el uso de predicados binarios con el verbo "estar" en el rango introducen una oración subordinada adjetiva precedida por la palabra "que".

- Se agregó un nuevo constructor para aceptar proposiciones cuantificadas que comiencen con la palabra "Ninguna". Estas proposiciones serán traducidas en fórmulas generadas por la negación de un cuantificador existencial. Veamos esto con unos ejemplos

```
Logic> %translate "ninguna figura verde es chica"
# ¬⟨∃x : Verde.x : Chico.x⟩
```

```
Logic> %translate "ninguna figura grande está arriba de A"
# ¬⟨∃x : Grande.x : arriba.x.A⟩
```

- Agregamos nuevos constructores que nos permitirán que proposiciones cuantificadas comiencen sin las palabras "todas", "alguna" o "ninguna". Estas frases comenzarán con la aplicación de un predicado binario sobre un individuo y luego sobre una frase cuantificada. Ejemplo de una frase con esta estructura es "A está arriba de todos las figuras cuadradas". Veamos algunas traducciones de frases con esta estructura

```
Logic> %translate "A está arriba de todas las figuras cuadradas"
# ⟨∀x : Cuad.x : arriba.A.x⟩
```

```
Logic> %translate "A está a la derecha de alguna figura roja"
# ⟨∃x : Rojo.x : der.A.x⟩
```

Las frases quen usen la palabra "alguna" solo irán acompañadas de aplicación positiva del predicado binario.

```
Logic> %translate "A no está abajo de ninguna figura roja"
# ¬⟨∃x : Rojo.x : abajo.A.x⟩
```

Mientras que las frases que usen la palabra "ninguna" solo irán acompañadas de aplicación negativa del predicado binario.

- Por último, agregamos constructores que nos permitirán aceptar frases que se traduzcan en fórmulas con cuantificadores "anidados". Veamos ejemplos de traducción para estas frases

```
Logic> %translate "las figuras rojas están arriba de las figuras verdes"
# ⟨∀x : Rojo.x : ⟨∀y : Verde.y : arriba.x.y⟩⟩
```

```
Logic> %translate "alguna figura azul está abajo de alguna figura grande"
# ⟨∃x : Azul.x : ⟨∃y : Grande.y : abajo.x.y⟩⟩

Logic> %translate "alguna figura mediana está abajo de todas
                   las figuras grandes"
# ⟨∃x : Mediano.x : ⟨∀y : Grande.y : abajo.x.y⟩⟩
```

3.3 Estructura de frases que no se traducen

Si bien extendimos la gramática base logrando aceptar frases cada vez más naturales y complejas, existen aún una gran cantidad de frases con estructura muy distinta que no son aceptadas. Describiremos a continuación fenómenos del lenguaje natural y estructuras de frases que en algún momento de este trabajo fueron consideradas pero que en la gramática final no son aceptadas.

- Si bien definimos todos los predicados unarios como adjetivos, una posible mejora sería definir a los predicados relacionados a la forma de las figuras geométricas como sustantivos. De esta manera podríamos aceptar frases como "A es un círculo" en vez de "A es circular" o "todos los triángulos son grandes" en vez de "todas las figuras triangulares son grandes".

- Uno de los enfoques de este trabajo fue trabajar con cuantificación in-situ en lenguaje natural, sin embargo, también podría ser útil describir proposiciones cuantificadas en lenguaje natural usando variables. De esta manera podríamos aceptar frases como "para toda figura f y g, si f está arriba de g entonces f es un cuadrado y g es un círculo" que se traduce en la fórmula

  $$\langle \forall f :: \langle \forall g : arriba.f.g : Cuad.f \wedge Circ.g \rangle \rangle$$

  la cual no puede ser generada con la gramática final de este trabajo.

  Es más, se puede analizar que con la gramática final de este trabajo, la cual no acepta frases en castellano que mencionen variables, las fórmulas generadas con cuantificadores "anidados" son tales que la variable ligada por el primer cuantificador solo puede aparecer en el término del segundo.

- Se podrían aceptar frases que mencionen la cantidad de elementos que cumplen una cierta propiedad. Por ejemplo, se podrían aceptar frases como "Hay 2 cuadrados rojos arriba de A" o "Hay más de 3 círculos".

- También se podrían aceptar frases de cuantificación existencial in-situ que usen las palabras “existe” o “hay”. Por ejemplo, “existe un cuadrado rojo arriba de A” o “hay un cuadrado rojo arriba de A” en vez de “algún cuadrado rojo está arriba de A”.

Otros fenómenos más complejos para los cuales se requiere un enfoque teórico más elaborado son los siguientes

Resolución de anáforas Donde anáfora refiere al uso de una expresión cuya interpretación depende de otra expresión en el mismo contexto. La resolución de anáforas es un problema conocido para el procesamiento de lenguaje natural y se encarga de dar una interpretación correcta a las anáforas. Por ejemplo, en la frase “Ana y María son amigas. Ellas se conocen hace mucho tiempo” se debería interpretar que el pronombre “Ellas” refiere a “Ana y María”. En el artículo (Ranta, 2004) se presenta una resolución minimalista de anáforas usando teoría de tipos en GF.

Tratamiento correcto de elipsis Donde elipsis refiere a la omisión de palabras que no son necesarias para completar el sentido de una oración. Un ejemplo de elipsis de verbo puede observarse en la frase “A es rojo y B azul”, donde se suprime el verbo “es”. La frase sin elipsis debería ser “A es rojo y B es azul”. Otro ejemplo de elipsis se da en la frase “A es rojo y está arriba de B”, en la cual se elimina al sujeto “A” luego de la conjunción. La frase sin elipsis debería ser “A es rojo y A está arriba de B”.

Capítulo 4

Análisis empírico de la gramática

En el capítulo anterior describimos la gramática abstracta y sus linealizaciones junto con las funciones de transferencia. Una vez que tuvimos una gramática que cubriera los aspectos más importantes nos pareció interesante analizarla empíricamente utilizando como entrada oraciones redactadas por otras personas. Para la recolección de ejemplos se difundió un formulario[1] el cual fue completado por un total de 34 personas. Si bien el formulario es anónimo (no hay información de las personas que lo respondieron) se supone que el mismo fue completado en gran parte por estudiantes y profesores de FaMAF.

4.1 Recolección de ejemplos

El formulario diseñado para la recolección de ejemplos consta de dos partes. En la primer parte se presentan tres "mundos" de SAT, y para cada uno se le pide a la persona que completa el formulario que describa mediante frases en castellano predicados que se satisfagan en dicho mundos.

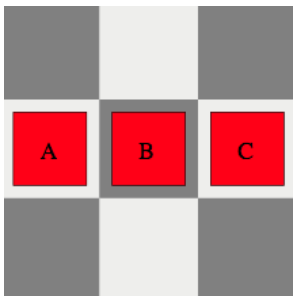
En la Figura 4.1 se muestran cada uno de los mundos presentes en el formulario. Para el primer mundo, por ejemplo, se espera que la persona que completa el formulario escriba frases del estilo "A, B y C son rojos" o "A está a la izquierda de B".

En la segunda parte del formulario se presentan 6 fórmulas lógicas de primer orden. Para cada una de estas fórmulas se le pide a la persona que completa el formulario que escriba como expresaría dicha fórmula en castellano.

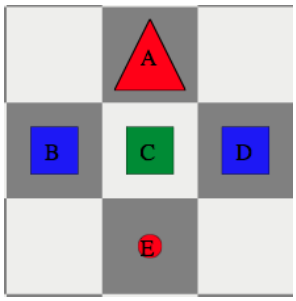
Las fórmulas presentes en el formulario son las siguientes:

- *Fórmula 1*: $Rojo.A \wedge Rojo.B \wedge Rojo.C$

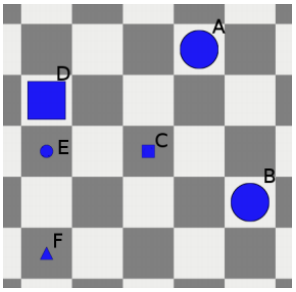
[1] https://goo.gl/forms/PPpxlYn84aq5WUeB2



(a) Mundo 0.



(b) Mundo 1.



(c) Mundo 2.

Figura 4.1: Mundos del formulario.

- *Fórmula 2*: $Rojo.A \wedge Cuad.A \wedge Rojo.B \wedge Cuad.B$
- *Fórmula 3*: $\langle \exists x : arriba.x.A : Azul.x \rangle$
- *Fórmula 4*: $\langle \forall x : Cuad.x \wedge Azul.x : arriba.A.x \rangle$
- *Fórmula 5*: $\langle \forall x : Verde.x : \langle \exists y : Azul.y : abajo.x.y \rangle \rangle$
- *Fórmula 6*: $\langle \exists x : Cuad.x : \langle \forall y : Circ.y : der.x.y \rangle \rangle$

Hemos separado el formulario en dos partes para analizar diferentes aspectos. La primera parte nos permitirá analizar y aproximar de qué manera se expresaría en lenguaje natural un usuario de SAT. Para completar esta parte no se requiere un conocimiento del lenguaje simbólico de la lógica de primer orden sino que solamente se requiere conocer los predicados propios de SAT. La segunda parte, en cambio, nos permitirá analizar de qué forma piensan las personas que completan el formulario que debería expresarse una fórmula lógica con una frase en castellano. De todas formas, como el foco de este trabajo está puesto en la precisión de la traducción y no tanto en la robustez, es esperable que muchas de las frases recibidas del formulario no sean aceptadas por la gramática. Algunas frases, por ejemplo, contendrán errores ortográficos que la gramática no podrá resolver. Otras frases, en cambio, usarán palabras desconocidas para la gramática y por lo tanto tampoco serán aceptadas.

4.2 Clasificación de ejemplos

Mostraremos a continuación algunos de los ejemplos recolectados y los clasificaremos según si fueron o no aceptados. Para los ejemplos aceptados mostraremos en qué fórmula lógica fueron traducidos y para los que no, intentaremos dar la razón por la cual no fueron aceptados.

Como pre-procesamiento se transforma el string recibido desde el formulario usando el comando de GF `ps -lexcode`, el cual nos servirá para separar listas de individuos y predicados unidos por comas.

Mundo 0

En total se recolectaron 117 frases de ejemplo para el Mundo 0 (Figura 4.1a). De estas 117 frases, 29 fueron aceptadas por la gramática mientras que 88 no fueron aceptadas. En la Tabla 4.1 se muestran algunas de las frases aceptadas para el Mundo 0 y su correspondiente formalización.

Frase	**Formalización**
B está a la derecha de A	$der.B.A$
C es rojo	$Rojo.C$
todas las figuras son rojas	$\langle \forall x : True : Rojo.x \rangle$
alguna figura es roja	$\langle \exists x : True : Rojo.x \rangle$
ninguna figura es verde	$\neg \langle \exists x : True : Verde.x \rangle$
todas las figuras son cuadradas y rojas	$\langle \forall x : True : (Cuad.x \wedge Rojo.x) \rangle$
B está a la derecha de A y a la izquierda de C	$(der.B.A \wedge izq.B.C)$
A, B y C son cuadrados	$(Cuad.A \wedge (Cuad.B \wedge Cuad.C))$
ninguna figura es azul	$\neg \langle \exists x : True : Azul.x \rangle$
A está a la izquierda de B y C	$(izq.A.B \wedge izq.A.C)$

Tabla 4.1: Algunas frases aceptadas del Mundo 0.

Además, existen dos frases que fueron aceptadas pero que su formalización no es la esperada. Una de ellas es la frase "C está a la derecha de todo" cuya traducción es la fórmula `der.C.todo`, mientras que la otra es la frase "ninguno es verde" que se traduce en la fórmula `Verde.ninguno`. De todas maneras, ninguna de estas fórmulas es válida en SAT, ya que los nombres de individuos deben ser letras mayúsculas.

En la Tabla 4.2 listamos algunas de las frases que no fueron aceptadas junto con la explicación de por qué no pueden ser parseadas por la gramática. En la Tabla 4.3 mostramos la cantidad de frases que no son aceptadas por alguna de estas razones, ordenadas de manera decreciente.

Frase	**Razón**
A y B son cuadrados rojos	La forma de las figuras aparece como sustantivo
C está a la derecha de un cuadrado grande	Cuantificación existencial usando la palabra “un”
el cuadrado B está a la derecha del cuadrado A	Referirse a una figura por su forma y nombre
A está a la izquierda de B y C a su derecha	Uso de anáforas
No hay ninguna figura abajo de los cuadrados	Cuantificación existencial usando la palabra “hay”
B esta a la derecha de A y a la izquierda de C.	Falta la tilde en la palabra “esta”
todos los cuadrados son de color rojo	Uso de la palabra “color”
existe al menos un cuadrado	Uso de la palabra “al menos” en cuantificación existencial
existen sólo 3 cuadrados	Menciona la cantidad de figuras que cumplen una condición
B está entre A y C	Uso de la palabra “entre”
A, B y C están en la fila del medio	Referencia a fila/columna
todos los cuadrados son del mismo tamaño	Uso de la palabra “tamaño”
B tiene un cuadrado rojo a derecha e izquierda	Uso de la palabra “tiene”

Tabla 4.2: Algunas frases no aceptadas del Mundo 0.

Motivo de rechazo	**Cant.**	**Ejemplo**
Forma como sustantivo	42	todos los cuadrados son rojos
Falta de tildes	8	A esta a la izquierda de B
Uso de la palabra "hay" (cuant. existencial)	7	no hay cuadrados medianos
Uso de la palabra "entre"	5	B está entre A y C
Cantidad de figuras que cumplen una condición	4	hay 3 cuadrados rojos
Referencia de una figura por forma y nombre	4	el cuadrado B está a la derecha del cuadrado A
Uso de la palabra "existe" (cuant. existencial)	3	no existen triángulos ni círculos
Uso de las palabra "al menos" (cuant. existencial)	2	existe al menos un cuadrado
Referencia a fila y columna	2	todas las figuras están en la misma fila
Uso de la palabra "tamaño"	2	todos los cuadrados son del mismo tamaño
Uso de anáforas y elipsis	1	A está a la izquierda de B y C a su derecha
Uso de la palabra "color"	1	todos los cuadrados son de color rojo

Tabla 4.3: Cantidad de frases no aceptadas por razón.

Podemos medir la precisión y cobertura al evaluar las frases recolectadas del Mundo 0. Definimos *precisión* como el conjunto de frases bien traducidas sobre el conjunto de frases parseadas y *cobertura* como el conjunto de frases parseadas sobre el total de frases a evaluar.

Precisión Mundo 0:

$$\frac{\#bien\_traducidas}{\#parseadas} = \frac{27}{29} = 0{,}93$$

Cobertura Mundo 0:

$$\frac{\#parseadas}{\#total} = \frac{29}{117} = 0{,}24$$

Donde podemos observar que la precisión es alta, es decir, cuando se parsea una oración suele estar bien traducida, pero la cobertura es baja, es decir, no es tan probable que dada una frase escrita por una persona (a la que suponemos bien escrita) sea parseada con la gramática definida.

Mundo 1 y Mundo 2

Para estos mundos no se realizó un análisis sobre la cantidad de frases parseadas y no parseadas, sino que solamente se extrajeron algunos ejemplos no aceptados. En las Tablas 4.4 y 4.5 se listan estas frases junto con las razones por las que no son aceptadas.

Frase	**Razón por la cual no es aceptada**
B y C son cuadrados medianos, pero E es un círculo pequeño	Uso de conjunción "pero"
ninguna figura es chica excepto E	Uso de la palabra "excepto"
E es la figura más chica	Uso de la palabra "más"
E es pequeño	Uso de la palabra "pequeño"

Tabla 4.4: Frases no aceptadas del Mundo 1.

Frase	**Razón por la cual no es aceptada**
C es el único cuadrado chico	Uso de la palabra "único" para expresar cantidad
C y F estan sobre la misma diagonal	Noción de diagonales en el tablero
Hay más círculos que cuadrados	Más figuras que otras (cantidad)
Los triángulos están abajo de las demás figuras	Uso de la palabra "demás"

Tabla 4.5: Frases no aceptadas del Mundo 2.

Fórmulas

Para cada fórmula presentada en el formulario analizaremos lo siguiente: cuáles son las respuestas más frecuentes y cuántas y cuáles son las frases que se parsean. Además haremos algunas observaciones sobre las frases que no se parsean.

Fórmula 1: $Rojo.A \wedge Rojo.B \wedge Rojo.C$

Para esta fórmula se obtuvieron un total de 12 resultados distintos, de los cuales 2 fueron parseados y 10 no. En la Tabla 4.6 podemos ver los resultados más frecuentes para esta fórmula. Un dato a tener en cuenta es que si bien solo 2 frases de las 12 distintas fueron parseadas, la más repetida con 18 apariciones en los resultados sí es aceptada.

Frase	Cantidad	Aceptada
A, B y C son rojos	18	Sí
A, B y C son figuras rojas	3	No
A, B y C son rojas	2	No
las figuras A, B y C son rojas	2	No

Tabla 4.6: Resultados más frecuentes para la fórmula 1.

Se puede observar el uso de la palabra "figura" en la aplicación de predicados, tanto en la frase "A, B y C son figuras rojas" como en la frase "las figuras A, B y C son rojas".

Fórmula 2: $Rojo.A \wedge Cuad.A \wedge Rojo.B \wedge Cuad.B$

Para esta fórmula se obtuvieron un total de 14 resultados distintos, de los cuales 3 fueron parseados y 11 no. En la Tabla 4.7 podemos ver los resultados más frecuentes para esta fórmula.

Frase	Cantidad	Aceptada
A y B son cuadrados rojos	18	No
A y B son cuadrados y rojos	5	Sí
las figuras A y B son cuadrados rojos	2	No

Tabla 4.7: Resultados más frecuentes para la fórmula 2.

Aquí podemos observar que las frases que usan sustantivos para describir la forma de las figuras son las más frecuentes y que la segunda frase más frecuente sí es aceptada por la gramática.

Fórmula 3: $\langle \exists x : arriba.x.A : Azul.x \rangle$

Para esta fórmula se obtuvieron un total de 21 resultados distintos, de los cuales uno fue parseado y 20 no. En la Tabla 4.8 podemos ver los resultados más frecuentes para esta fórmula.

Frase	Cantidad	Aceptada
Hay una figura azul arriba de A	10	No
Hay un elemento azul arriba de A	4	No
Hay una figura que está arriba de A y es azul	2	No

Tabla 4.8: Resultados más frecuentes para la fórmula 3.

En esta tabla podemos observar que los resultados más frecuentes comienzan con la palabra "hay", por lo que no son aceptados. De las frases no aceptadas podemos hacer las siguientes observaciones: 13 comienzan con la palabra "hay", 4 comienzan con la palabra "existe" y 3 comienzan con la palabra "algún" o "alguna". También aparecen en resultados no aceptados las palabras "sobre" o "debajo" en vez de "arriba de" o "abajo de", respectivamente.

Fórmula 4: $\langle \forall x : Cuad.x \wedge Azul.x : arriba.A.x \rangle$

Para esta fórmula se obtuvieron un total de 25 resultados distintos, de los cuales uno fue parseado y 24 no. En la Tabla 4.9 podemos ver los resultados más frecuentes para esta fórmula.

Frase	Cantidad	Aceptada
todos los cuadrados azules están arriba de A	6	No
todos los cuadrados azules están debajo de A	4	No
A está arriba de todos los cuadrados azules	2	No

Tabla 4.9: Resultados más frecuentes para la fórmula 4.

Es importante observar que la frase más frecuente en los resultados no es una interpretación correcta de la fórmula, ya que la variable ligada por el cuantificador aparece como segundo argumento del predicado *arriba*. Esto evidencia la dificultad de expresar en lenguaje natural fórmulas cuantificadas, aun para personas con conocimiento de lógica.

Nuevamente podemos observar que los resultados más frecuentes usan sustantivos para representar la forma de las figuras. La frase aceptada para esta fórmula es "todas las figuras cuadradas y azules están abajo de A", la cual no se encuentra entre las más frecuentes.

Fórmula 5: $\langle \forall x : Verde.x : \langle \exists y : Azul.y : abajo.x.y \rangle \rangle$

Para esta fórmula se obtuvieron un total de 33 resultados distintos, de los cuales uno fue parseado y 32 no. Es importante destacar que de los 33 resultados solo una frase se repite más de una vez. Esta frase es "Todas las figuras verdes están abajo de alguna figura azul" con 2 repeticiones, y sí es aceptada por la gramática.

De las frases que no son aceptadas 16 comienzan con la palabra "todos" o "todas", a veces en singular y 8 comienzan con las palabras "para todo" o "para cada". Como ejemplo, la frase "para toda figura verde hay una figura azul arriba de ella".

Fórmula 6: $\langle \exists x : Cuad.x : \langle \forall y : Circ.y : der.x.y \rangle \rangle$

Para esta fórmula se obtuvieron un total de 29 resultados distintos, de los cuales uno fue parseado y 28 no. En la Tabla 4.10 podemos ver los resultados más frecuentes para esta fórmula.

Frase	**Cantidad**	**Aceptada**
Existe un cuadrado que tiene todos los círculos a la derecha	2	No
Hay un cuadrado a la derecha de todos los círculos	2	No
Hay un cuadrado que está a la derecha de todos los círculos	2	No

Tabla 4.10: Resultados más frecuentes para la fórmula 6.

De las 29 frases 25 usan sustantivos para representar las formas de figuras, por lo que no son aceptadas. Además 13 comienzan con la palabra "existe" y 11 con la palabra "hay".

Capítulo 5

Conclusión

En esta tesis analizamos cómo definir una traducción del castellano al lenguaje simbólico de fórmulas lógicas de primer orden. Para esto, presentamos la herramienta Grammatical Framework, la cual puede ser pensada como un lenguaje de programación con propósito específico en la construcción de gramáticas multilingües. En el capítulo 2 describimos muchas de las características de esta herramienta y en el capítulo 3 vimos la utilidad de las mismas en el contexto de este trabajo. A continuación describimos las dos que consideramos que tuvieron mayor influencia.

En primer lugar, la división de la gramática en sintaxis abstracta y sintaxis concreta nos provee una separación adecuada de cada lenguaje concreto y nos permite trabajar a nivel semántico con los árboles de sintaxis abstractos. Por otra parte, para este trabajo fue de gran utilidad el uso de la librería Resource Grammar Library, la cual provee una interfaz de una gramática del español, permitiendo poner el foco en la definición de nuevos constructores en la sintaxis abstracta sin tener que definir nuestras propias reglas sintácticas del español.

La gramática final de este trabajo es capaz de capturar frases con cierto grado de aceptabilidad dentro del dominio de fórmulas lógicas de SAT, ya que si bien priorizamos la precisión de la traducción, intentamos aumentar de manera iterativa la cantidad y calidad de frases aceptadas.

Por último, un análisis empírico sobre datos recolectados nos muestra que la gramática no es muy robusta, en el sentido de que muchas de las frases recolectadas no son parseadas. Esto no nos sorprende, ya que no es una tarea sencilla definir un sistema de traducción que maximice la precisión y la cantidad de frases aceptadas al mismo tiempo. Se cree que para muchas de las frases no aceptadas se podría mejorar la gramática sin la necesidad de mucho esfuerzo. Para otras, se requiere un estudio teórico más complejo, como es el caso de frases que involucren resolución de anáforas y elipsis.

Tomando como base el trabajo de esta tesis podemos listar los siguientes posibles trabajos futuros:

1. Analizar en mayor profundidad las frases no aceptadas por la gramática, identificando los cambios necesarios en la misma para que puedan ser aceptadas.

2. Implementar estos cambios en la gramática.

3. Definir una fase de pre-procesamiento para mejorar la robustez del sistema, por ejemplo, para aceptar frases con errores ortográficos. Se piensa que para realizar este pre-procesamiento posiblemente sea más conveniente usar herramientas basadas en métodos estadísticos.

4. Integración de la gramática en Sat.

5. Medir precisión y cobertura para los Mundos 1 y 2.

6. Pensar en otras maneras de medir precisión y cobertura del sistema, por ejemplo, evaluando la performance de la traducción sobre un conjunto de sentencias obtenidas por un generador de expresiones referenciales.

Bibliografía

Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *EACL 2009, 12th Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, Athens, Greece, March 30 - April 3, 2009*, pages 69–76, 2009. URL `http://www.aclweb.org/anthology/E09-1009`.

David Barker-Plummer, Jon Barwise, and John Etchemendy. *Language, Proof, and Logic: Second Edition*. Center for the Study of Language and Information/SRI, 2nd edition, 2011. ISBN 1575866323, 9781575866321.

Sergio Barrionuevo. Diseño de objetos interactivos para la enseñanza de la lógica. *VII Jornadas de Investigación en Filosofía para profesores, graduados y alumnos*, 2008. URL `http://jornadasfilo.fahce.unlp.edu.ar/vii-jornadas/ponencias/BARRIONUEVO%20Sergio.pdf`.

Noam Chomsky. *Syntactic Structures*. Mouton and Co., The Hague, 1957.

Miguel Pagano Emmanuel Gunther, Alejandro Gadea. Sat. URL `https://github.com/manugunther/sat`.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden, 2004.

Franco M. Luque. El lenguaje natural como lenguaje formal. *CoRR*, abs/1703.04417, 2017. URL `http://arxiv.org/abs/1703.04417`.

Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford Univ. Press, New York, 1998.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf type theory: an introduction*. Clarendon, 1990.

Carlos Oller. Lógica formal y argumentación. *Revista de filosofía y teoría política*, (37):83–91, 2013. ISSN 2314-2553. URL http://www.rfytp.fahce.unlp.edu.ar/article/view/RFyTPn37a04.

Aarne Ranta. Grammatical framework webpage. a. URL http://www.grammaticalframework.org/.

Aarne Ranta. Aarne ranta homepage. b. URL http://www.cse.chalmers.se/~aarne/.

Aarne Ranta. Computational semantics in type theory. *Mathématiques et sciences humaines. Mathematics and social sciences*, (165), 2004.

Aarne Ranta. Translating between language and logic: What is easy and what is difficult. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 5–25, 2011a. doi: 10.1007/978-3-642-22438-6_3. URL https://doi.org/10.1007/978-3-642-22438-6_3.

Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011b. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Hiroyuki Seki, Ryuichi Nakanishi, Yuichi Kaji, Sachiko Ando, and Tadao Kasami. Parallel multiple context-free grammars, finite-state translation systems, and polynomial-time recognizable subclasses of lexical-functional grammars. In *Proceedings of the 31st Annual Meeting on Association for Computational Linguistics*, ACL '93, pages 130–139, Stroudsburg, PA, USA, 1993. Association for Computational Linguistics. doi: 10.3115/981574.981592. URL https://doi.org/10.3115/981574.981592.

S.M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343, 1985. URL http://www.eecs.harvard.edu/shieber/Biblio/Papers/shieber85.pdf.