Introducción a la Construcción de Compiladores

Alejandro Piad Morffis Juan Pablo Consuegra Ayala
Rocío Cruz Linares

Índice general

Introducción

Este documento tiene como objetivo presentar el diseño de un curso de Compilación para estudiantes de Ciencia de la Computación. Este documento no es en absoluto un libro de texto de la asignatura Compilación (ni pretende serlo). Se asume que los profesores y estudiantes cuentan con documentación adicional para profundizar en los conocimientos impartidos. Cuando se presenta una sección de contenido, es más con el objetivo de presentar una guía sobre qué contenido será impartido, ilustrar un posible enfoque para introducir dicho contenido, y quizá además para acotar ciertas cuestiones específicas que consideramos de especial importancia. En cualquier caso, los estudiantes pueden encontrar también en este documento respuestas a preguntas de su interés, no solo desde el punto de vista puramente técnico (preguntas sobre el contenido) sino también desde el punto de vista metodológico (preguntas sobre la forma, o preguntas sobre las preguntas). De esta manera pueden entender mejor qué podemos prometerles como instructores, que se espera de ellos en cambio, y de qué forma pueden sacar el máximo provecho de este curso.

Diseño del Curso

Este curso fue diseñado bajo una concepción más dinámica e interactiva, más integrada con el resto de las disciplinas, y más práctica, pero sin perder en formalidad, precisión y profundidad en el conocimiento impartido. Para desarrollar este curso, primeramente nos planteamos primerante la siguiente pregunta: *¿Cuáles son las cuestiones fundamentales que esta asignatura ayuda a responder y comprender?*

El objetivo de esta pregunta es cuestionarse primeramente el porqué de un curso de Compilación, antes de decidir qué contenidos específicos se deben incluir. Para ellos analizamos el lugar que ocupa la asignatura dentro del plan general de la carrera de Ciencia de la Computación, y los diversos conocimientos que enlaza. De esta forma pretendemos diseñar un curso más integrado y de mayor utilidad para la formación profesional.

En una primera mirada, el curso de Compilación ocupa un lugar privilegiado dentro de la carrera. Ocurre justo en un momento donde los estudiantes han obtenido la mayor parte de los conocimientos básicos (programación, diseño de algoritmos, lógica y matemática y discreta), y están a punto de comenzar a adentrarse en las cuestiones propias de la Ciencia de la Computación. Esta asignatura viene a ser, de un modo tal vez un poco arrogante, la asignatura más integradora de toda esta primera parte. En este punto los estudiantes

conocen cuestiones de muy alto nivel, tales como diseños de algoritmos y estructuras de datos, teoría de la computabilidad y la complejidad computacional, patrones de diseño y diversos paradigmas de programación, así como una creciente colección de lenguajes de programación particulares. Además conocen muchos detalles de bajo nivel, tales como la arquitectura de los procesadores, la memoria, los procesos y los sistemas operativos, lenguajes de máquina y ensambladores y modelos computacionales para la implementación de operaciones aritméticas eficientes.

Compilación viene a ser la asignatura que enlaza estos dos mundos, el de alto nivel, y el de bajo nivel. Por primera vez los estudiantes pueden hacerse una idea del proceso completo que sucede desde la concepción de una idea de un algoritmo o sistema, su análisis, su implementación en un lenguaje de alto nivel (o varios lenguajes, frameworks y bibliotecas), y como todo ello se traduce a un lenguaje de máquina, y se ejecuta encima de un dispositivo de cómputo concreto. Desde este punto de vista, Compilación permite integrar los conocimientos del resto de las disciplinas de esta primera parte de la carrera, y viene a cerrar lo que podemos llamar el perfil básico del Científico de la Computación. Es en cierta forma la última de las asignaturas básicas y la primera de las asignaturas avanzadas.

En una primera enumeración, podemos intentar definir habilidades y conocimientos concretos que los estudiantes pueden aspirar a obtener en esta asignatura. Un buen ejercicio consiste en preguntar a los propios estudiantes qué creen que deberían aprender en esta asignatura. Aunque realmente no tienen una idea clara de cuál es el campo de estudio que tienen delante, o dicho de otra forma, sobre qué *deberían* aprender; hemos encontrado que los estudiantes sí tienen expectativas bastante claras sobre qué *quieren* aprender. Por supuesto, encontrar un balance adecuado entre ambos intereses (los *nuestros* y los *suyos*) debe ser un objetivo a perseguir. Presentamos a continuación una posible e incompleta lista de estas habilidades, que hemos recopilado de varias sesiones de preguntas a estudiantes, edulcoradas con nuestras propias concepciones:

- Reconocer problemas típicos de compilación y teoría de lenguajes.
- Crear reconocedores y generadores de lenguajes.
- Entender el funcionamiento de un compilador.
- Saber cómo se implementan instrucciones de alto nivel.
- Diseñar lenguajes de dominio específico.
- Poder implementar intérpretes y compiladores de un lenguaje arbitrario.
- Entender las diferencias y similaridades entre lenguajes distintos.
- Conectar lenguajes de alto nivel con arquitecturas de máquina.
- Aprender técnicas de procesamiento de lenguaje natural.

Estas habilidades cubren desde los temas más prácticos sobre el diseño de lenguajes y compiladores hasta cuestions más filósoficas y abstractas relacionadas con otras áreas del conocimiento, desde la ingeniería de software hasta la inteligencia artificial. Los estudiantes quieren no sólo ser capaces de *hacer*, sino también, y tal vez más importante, quieren ser capaces de *entender* cómo funcionan los algoritmos, técnicas, estructuras de datos, patrones de diseño, modelos de razonamiento, que son empleados en esta ciencia.

Para dar forma a un curso que pueda ayudar a los estudiantes a obtener estas habilidades (y otras relacionadas), nos dimos entonces a la tarea de resumir las preguntas o cuestiones fundamentales que debe responder dicho curso. En una primera instancia, parece que una

pregunta tan clara cómo *¿qué es un compilador?* o incluso *¿cómo funciona un compilador?* puede servir de guía al contenido del curso. Sin embargo, en una mirada más profunda, podemos descubrir que hay cuestiones más primarias, a las cuáles un compilador es ya una respuesta, un medio más que un fin en sí mismo. De hecho, podemos cuestionarnos porqué surgió la necesidad de hacer compiladores en primer lugar, a qué problema intentaron dar respuesta, y tratar de escarbar entonces las preguntas más primarias que subyacen en esta historia.

Un poco de historia

Podemos comenzar esta historia más o menos así. Hay una gran distancia entre el nivel de razonamiento que ocurre en el cerebro y el nivel de razonamiento que ocurre en una computadora. Los problemas de cualquier dominio se resuelven pensando a un nivel de abstracción con un lenguaje que describe las reglas de ese dominio. Hubo una época en que estos niveles de abstracción tenía que conectarlos el programador. De hecho, en esta época, la diferencia entre analista y programador era justamente que el analista diseñaba la solución en su lenguaje, y el programador la traducía a un programa ejecutable en el lenguaje de máquina.

Por ejemplo, en 1952 Grace Hooper trabajaba en la simulación de trayectorias balísticas. Para dirigir un proyectil a su objetivo, los modelos físicos se describen en un lenguaje de ecuaciones diferenciales y mecánica newtoniana. Sin embargo, para poder implementar estos modelos en un dispositivo de cómputo hay que hablar en un lenguaje de registros, pilas e interrupciones. Esta diferencia es lo que hace que programar sea tan difícil, y hacía extremadamente lento el desarrollo de nuevos modelos porque a cada paso podían haber errores tanto en la modelación como en la codificación. ¿Cuando algo fallaba, de quién era la culpa? ¿Del analista o del programador? ¿O peor, del sistema de cómputo?

Entonces a Grace Hooper se le ocurrió una genial idea: viendo que el proceso de convertir las ecuaciones diferenciales a programas concretos era fundamentalmente mecánico, ¿por qué no dejar que la propia computadora hiciera esta conversión?

Esta idea genial tomaría varios años en perfeccionarse al punto de ser una realidad. El primer compilador de Grace Hooper para el lenguaje A-0 realmente era prácticamente un linker con algunas funciones básicas. Los primeros lenguajes de alto nivel en tener compiladores “serios” son FORTRAN (1957, John Backus), ALGOL (1958, Friedrich Bauer) y COBOL (1960, Grace Hooper). Una ventaja adicional además de reducir el tiempo de desarrollo, era la posibilidad de compilar el mismo programa para múltiples plataformas. En 1960 por primera vez se compiló el mismo programa de COBOL para dos máquinas distintas: UNIVAC II y RCA 501.

En este punto los lenguajes se volvieron suficientemente complicados, al punto que ya los compiladores no se podían escribir “a mano”. Entonces hizo falta volcarse a la teoría, y desarrollar una ciencia sobre qué tipos de lenguajes de programación se podían compilar, y con qué compiladores. Esto dio nacimiento, en 1960, a la ciencia que hoy conocemos como Compilación. Motivada no solo por un motivo práctico, sino también fundamentada en los principios teóricos más sólidos, la compilación vino a convertirse en una de las primeras justificaciones para que la computación se cuestionara problemas propios, y dejara

de ser una mera herramienta de cálculo. Problemas tan distantes como el procesamiento de lenguaje natural y la naturaleza de las funciones computables han caído bajo el diapasón de las problemáticas estudiadas en este campo. Hoy la compilación es una ciencia sólida, fundamentada en años de teoría formal y práctica ingenieril.

Escondida bajo todo este aparataje formal y toda la gama de experiencias y resultados teóricos y prácticos de los últimos 60 años, podemos encontrar una cuestión más fundamental, una pregunta que quizá retroceda hasta el propio Alan Turing, o incluso más allá, hasta Ada Lovelace y Charles Babbage con su máquina analítica. La cuestión es esta:

¿Cómo hablar con una computadora?

Esta pregunta es, a nuestro modo de ver, en última instancia la cuestión a responder en este curso. Todos los intentos de diseñar lenguajes, todos los algoritmos y técnicas descubiertos, todos los patrones de diseño y arquitecturas, están en última instancia ligados al deseo de poder hacer una *pregunta* a la computadora, y obtener una *respuesta* a cambio. No importa si la pregunta es calcular cierta trayectoria de proyectiles, o encontrar la secuencia de parámetros que minimizan cierta función. Todo programa es en cierto modo una conversación con la computadora, un canal de comunicación, que queremos que sea lo suficientemente poderoso como para poder expresar nuestras ideas más complejas, y lo suficientemente simple como para poder ser entendido por una máquina de Turing. Como veremos en este curso, hallar el balance adecuado es un problema sumamente interesante, e intentar responderlo nos llevará por un camino que nos planteará muchas otras interrogantes, entre ellas las siguientes:

- ¿Qué tipos de lenguajes es capaz de *entender* una computadora?
- ¿Cuánto de un lenguaje debe ser *entendido* para poder entablar una conversación?
- ¿Qué es *entender* un lenguaje?
- ¿Es igual de fácil o difícil *entender* que *hablar* un lenguaje?
- ¿Podemos caracterizar los lenguajes en términos computacionales según su complejidad para ser *entendidos* por una computadora?
- ¿Cómo se relacionan estos lenguajes con el lenguaje humano?
- ¿Qué podemos aprender sobre la naturaleza de las computadoras y los problemas computables, a partir de los lenguajes que son capaces de reconocer?
- ¿Qué podemos aprender sobre el lenguaje humano para hacer más inteligentes a las computadoras?
- ¿Qué podemos aprender sobre el lenguaje humano, y la propia naturaleza de nuestra inteligencia, a partir de estudiar los lenguajes entendibles por distintos tipos de máquinas?

Estas preguntas, aunque no serán directamente respondidas en los siguientes capítulos, forman la columna vertebral del contenido del curso, en el sentido en qué todo lo presentado es con la intención de, al menos, poder echar un poco de luz en estos temas. Esperamos que al finalizar el curso, los estudiantes sean capaces de discutir las implicaciones filosóficas de las posibles respuestas a estas preguntas, y no solo a las cuestiones técnicas o más prácticas que el curso ataca. Por este motivo, sí trataremos en la medida de lo posible de, además del contenido técnico, adicionar en ocasiones algunos comentarios o discusiones más filósoficas al respecto de estas preguntas y otras similares.

Anatomía de un Compilador

Comenzaremos este viaje diseccionando el sistema computacional canónico de la teoría de lenguajes formales: un compilador. A grandes razgos, un compilador no es más que un programa, cuya entrada y salida resultan ser también programas. La entrada es un programa en un lenguaje que llamaremos "de alto nivel", y la salida en un lenguaje de "bajo nivel", que es equivalente al primero. Exactamente qué es alto y bajo nivel dependerá de muchos factores, y no existe una definición formal. De forma general, un lenguaje de alto nivel es aquel que nos es cómodo a los programadores para expresar las operaciones que nos interesa ejecutar. Así mismo, un lenguaje de bajo nivel es aquel que un dispositivo de cómputo puede ejecutar de forma eficiente. Tal vez los ejemplos más típicos sean un lenguaje orientado a objetos y un lenguaje ensamblador respectivamente, pero existen muchas otras combinaciones de lenguaje de entrada y salida de interés.

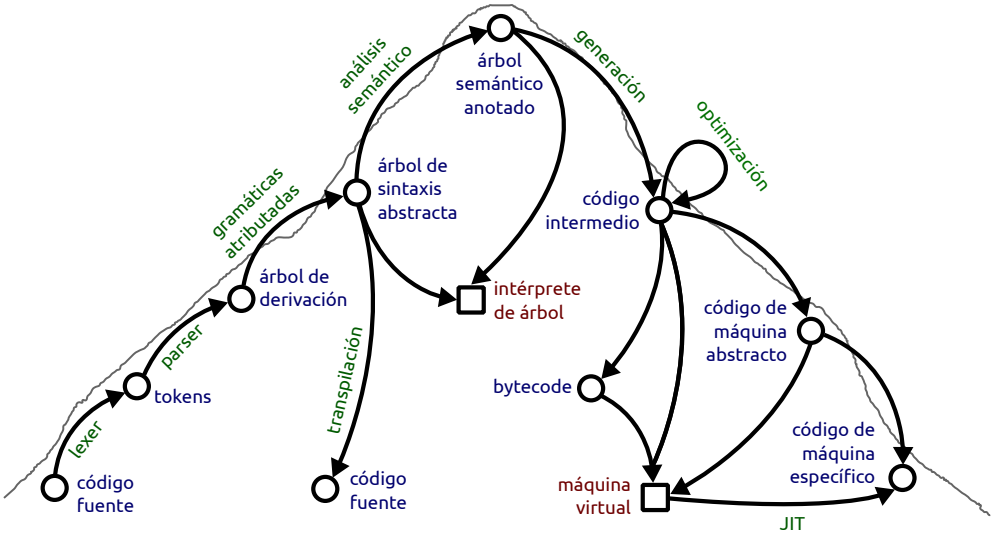


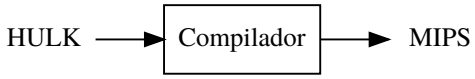
Figura 1: Representación abstracta del proceso de compilación.

Ahora bien, antes de zambullirnos de lleno en la anatomía de un compilador, es conveniente mencionar algunas sistemas de procesamiento de lenguajes relacionados. Podemos intentar categorizarlos según el "tipo" del lenguaje de entrada y salida. En primer lugar, el ejemplo clásico es cuando queremos convertir un lenguaje de alto nivel a otro de bajo nivel, y justamente llamamos a este sistema un **compilador**. El caso contrario, cuando queremos convertir de un lenguaje en bajo nivel a otro en alto nivel, podemos llamarle por analogía un **decompilador**. Este tipo de herramientas son útiles para analizar y realizar ingeniería inversa en programas para los que, tal vez, ya no tenemos el código fuente, y necesitamos entender o modificar. Los otros dos casos, de alto nivel a alto nivel y de bajo nivel a nivel son básicamente **traductores**; y en ocasiones se les llama también **transpiladores**. Por ejemplo,

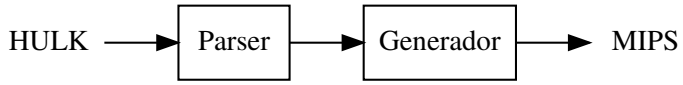
TypeScript es un lenguaje de alto nivel que se "transpila" a JavaScript, otro lenguaje también de alto nivel. Entre lenguajes de bajo nivel podemos tener también traductores. Un ejemplo son los llamados **compiladores JIT** (*just-in-time*), que se usan para traducir un programa compilado a un lenguaje de bajo nivel genérico (por ejemplo **IL**) a un lenguaje de máquina específico para la arquitectura donde se ejecuta.

Volvamos entonces al caso clásico, el **compilador**. o este curso vamos a usar como una guía didáctica el diseño de un compilador para el lenguaje HULK, que compilará a un lenguaje de máquina denominado MIPS. Los detalles de ambos lenguajes serán introducidos a medida que sea conveniente, pero por el momento cabe decir que HULK es un lenguaje orientado a objetos, con recolección automática de basura, herencia simple, polimorfismo, y un sistema de tipos unificado. MIPS es un lenguaje ensamblador de pila para una arquitectura de 32 bits con registros y operaciones aritméticas, lógicas y orientadas a cadenas.

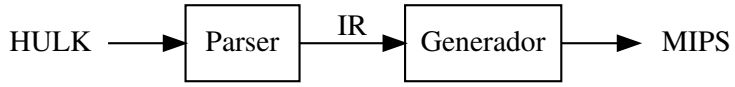
Intentemos entonces definir esta maquinaria paso a paso. De forma abstracta nuestro compilador es una "caja negra" que convierte programas escritos en HULK a programas escritos en MIPS:



Para comenzar a destapar esta caja negra, notemos que al menos tenemos dos componentes independientes: uno que opera en lenguaje COOL y otro que opera en lenguaje MIPS. Necesitamos ser capaces de "leer" un programa en COOL y "escribirlo" en MIPS. Al primer módulo, que "lee", le llamaremos *parser*, o analizador sintáctico, por motivos históricos que veremos más adelante. Al segundo componente le llamaremos simplemente el *generador*.



De aquí surge immediatamente una pregunta: ¿qué protocolo de comunicación tienen estos módulos? Es necesario diseñar una especie de lenguaje intermedio, un mecanismo de representación que no sea ni COOL ni MIPS, sino algo que esté "a medio camino" entre ambos. Es decir, hace falta traducir el programa en COOL a alguna forma de representación abstracta, independiente de la sintaxis, que luego pueda ser interpretada por el generador y escrita en MIPS. Llamésmole de momento *representación intermedia (IR)*.



Pasemos entonces a analizar qué forma debe tener esta representación intermedia. En

principio, debe ser totalmente independiente de COOL o de MIPS, en términos de sintaxis. A fin de cuentas, podemos estar generando para cualquier otra plataforma, no solo MIPS. Por otro lado, tiene que ser capaz de capturar todo lo qué es posible expresar en COOL. A este tipo de representación, independiente de la sintaxis, pero que captura todo el significado, le vamos a llamar indistintamente *representación semántica* en ocasiones, justamente por este motivo. ¿Qué va en una representación semántica? Pues todos los conceptos que son expresables en un programa, dígase clases, métodos, variables, expresiones, ciclos. ¿Qué no va? Pues todo lo que sea "superfluo" al significado. Por ejemplo, el hecho de que un método tiene un nombre, pertenece a una clase, y tiene ciertos argumentos de ciertos tipos, es importante semánticamente. Dos métodos se diferencian por alguno de estos elementos. Por otro lado, el hecho de que un método se escribe primero por su nombre, luego por los argumentos entre paréntesis seguidos por los nombres de sus tipos, es poco importante *en este momento*. Daría lo mismo que los tipos fueran delante o detrás de los nombres de los argumentos, ya en esta fase del procesamiento lo que nos interesa es de qué tipo es un argumento, y no si ese tipo se declara antes o después textualmente.

Definir exactamente qué es semánticamente importante en un lenguaje particular no es una tarea fácil, y veremos una vez llegados a ese punto algunas ideas para atacar este problema (qué es en última instancia un problema de diseño, y por lo tanto es más un arte que una ciencia, al menos en el sentido artístico de Donald Knuth). Lo que sí es interesante de momento, es analizar qué tipo de procesamiento es importante, o al menos conveniente, realizar sobre esta representación intermedia.

Cómo justificaremos más adelante, existen estructuras linguísticas que no son fáciles de reconocer, porque son dependientes del contexto. Por ejemplo, la expresión `x = y + z` es muy sencilla de reconocer sintácticamente, pero en un lenguaje con tipado estático esta expresión puede no pertenecer al lenguaje según los tipos de cada variable. Este es un problema clásico de dependencia del contexto, donde la expresión `x = y + z` es válida si existe en el contexto donde, por ejemplo existe, `int x, int y, int z` pero no donde el contexto es `int x, int y, object z`. Hay muchos problemas que son dependientes del contexto, entre ellos:

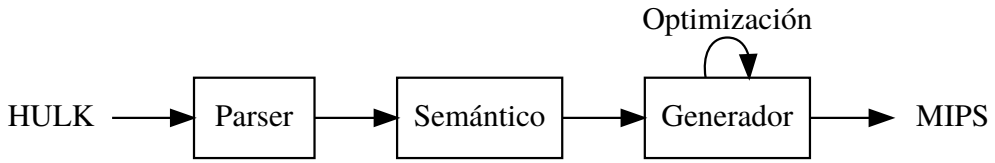
- Declaración de variables antes de su uso
- Consistencia en los tipos declarados y las operaciones realizadas
- Consistencia entre la declaración de una función y su invocación
- Retornos en todos los caminos de ejecución

La solución de estos problemas empleando las técnicas de la teoría de lenguajes solamente es en general no polinomial, y a veces no computable. Pero muchos de estos problemas se pueden resolver de forma más sencilla analizando la estructura computacional intermedia. ¿Por qué? Veremos más adelante que esta estructura tiene generalmente forma arbórea, y en una estructura arbórea es fácil analizar la consistencia de los tipos y problemas similares recorriendo cada uno de los nodos. De forma recursiva, el nodo raíz (o programa) estará correcto si cada hijo está correcto. Para realizar este tipo de procesamiento, introduciremos una nueva fase, que llamaremos *chequeo semántico*, y que opera justamente sobre la representación semántica del programa.

Finalmente, justo antes de generar el código ejecutable final, cabe preguntarse si existe algún tipo de procesamiento adicional conveniente. Descubriremos más adelante varios tipos de optimizaciones que se pueden aplicar en este punto, entre ellas:

- Eliminar código no alcanzable
- Expandir expresiones constantes
- Elimiar asignaciones superfluas
- Desenrrollar ciclos con extremos constantes

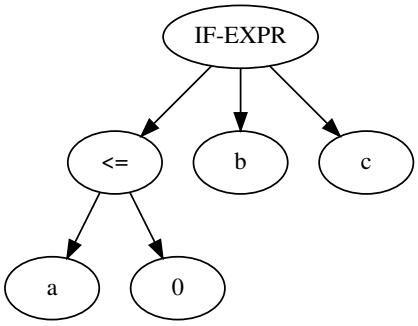
Estas optimizaciones a menudo son convenientes de realizar sobre una estructura del programa mucho más cercana al código de máquina que al código original. Son muchos los factores que influyen en esto y que veremos más adelante, pero de forma intuitiva, es fácil entender que optimizar un programa debe ser un proceso muy cercano al código de máquina, pues la propia naturaleza de la optimización requiere explotar características propias de la maquinaria donde será ejecutado el código. Introduciremos entonces una fase de optimización, que de momento visualizaremos de forma *paralela* al proceso de generación, pues en la práctica ambas fases comparten la misma representación y ambos procesos ocurren de forma más o menos simultánea.



Antes de terminar vamos a develar una fase adicional, justo antes del proceso de *parsing*. Para entender por qué, debemos introducirnos un poco más profundamente en este proceso. El proceso de *parsing* básicamente de lo que se encarga es convertir cadenas de texto (escritas en el lenguaje origen) a una estructura (arbórea) que captura la semántica del programa. Por ejemplo, supongamos que tenemos la siguiente expresión como parte de un programa en HULK:

```
if (a <= 0)
    b
else
    c
```

Esta expresión, desde el punto de vista semántico, podemos pensar que se transforma a una estructura como la siguiente:



De forma simplificada, en esta estructura hemos representado semánticamente el significado de la expresión sintáctica anterior, extrayendo los elementos importantes (el hecho de que una expresión `if` contiene tres elementos: condición, parte del cuerpo `true` y parte del cuerpo `false`) y obviando los detalles de sintaxis que ya no son imporantes (por ejemplo las palabras `then`, `else` y `fi` que solamente sirven para separar los bloques correspondientes).

La solución de este problema (convertir una secuencia de texto en una representación semántica) nos tomará la primera mitad del curso, pues es uno de los temas centrales en toda la teoría de lenguajes y la compilación en particular. Más adelante formalizaremos con exactitud este problema y veremos muchísimas estrategias de solución. Pero antes de llegar a ese punto, es necesario resolver un sub-problema de menor complejidad, que aún así nos dará suficiente trabajo como para desarrollar gran parte de la teoría de lenguajes en su solución. El problema en cuestión es el siguiente.

La cadena de texto de entrada, realmente está formada por una secuencia de caracteres. En el caso anterior, por ejemplo, tenemos la siguiente secuencia:

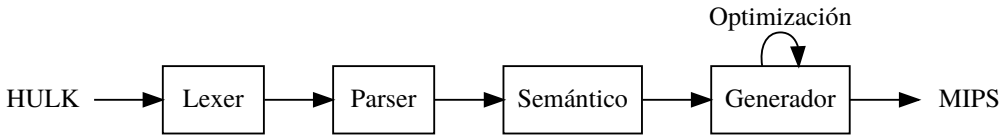
i	f	\s	(	a	<	=	0	)	\n	\t	b	\n	e	l	s	e	\n	\t	c	\n

En esta secuencia hemos representado cada caracter por separado, y hemos puesto explícitamente los espacios en blanco (`\s`), cambios de línea (`\n`) y espacios de tabulación (`\t`). Esta secuencia es lo que realmente "ve" el compilador en la entrada. Por lo tanto, desde el punto de vista puramente sintáctico, existen muchas secuencias que son exactamente el mismo programa. Por ejemplo, si existen varios espacios entre dos símbolos, o si no hay cambios de línea, o si los bloques están indentados con 4 espacios en vez con un caracter `\t`, realmente estamos en presencia del mismo programa.

Para simplificar la tarea del *parser*, es conveniente convertir primeramente esta secuencia de caracteres en una secuencia de **tokens**, que no son más que los elementos básicos que componen el lenguaje, lo que pudiéramos llamar las *palabras* o *símbolos* más básicos. Por ejemplo, en el caso anterior, quisiéramos obtener la siguiente secuencia de **tokens**:

if	(	a	<=	0	)	b	else	c

Introduciremos entonces una primera fase, que llamaremos indistintamente *tokenización*, o análisis lexicográfico, para transmitir la idea de que en esta fase solamente se procesan los elementos *léxicos*, es decir, las palabras, sin llegar todavía a tocar los elementos *sintácticos*, es decir, la estructura. Al mecanismo encargado de resolver este problema le llamaremos entonces *lexer*.



A modo de resumen, tenemos entonces cinco fases fundamentales:

Análisis Lexicográfico (*lexer*) donde se realiza la conversión de una secuencia de caracteres a una secuencia de *tokens*.

En esta fase se detectan los errores relacionados con la escritura incorrecta de símbolos, por ejemplo, un número decimal con más dígitos de los permitidos, una cadena de texto que no termina en `"`, un identificador que usa un caracter inválido (e.j. `$`).

Análisis Sintáctico (*parser*) donde se determina la estructura sintáctica del programa a partir de los *tokens* y se obtiene una estructura intermedia.

En esta fase se detectan los errores relacionados con la sintaxis, por ejemplo, un paréntesis no balanceado, una función que no tiene cuerpo, un `if-else` con la parte `else` vacía.

Análisis Semántico donde se verifican las condiciones semánticas del programa y se valida el uso correcto de todos los símbolos definidos.

En esta fase se determinan los errores relacionados con los símbolos y tipos, por ejemplo, variables no declaradas o usadas antes de su declaración o funciones invocadas con un número o un tipo incorrecto de los argumentos.

Optimización donde se eliminan o simplifican secciones del programa en función de la arquitectura de máquina hacian donde se vaya a compilar.

Generación donde se convierte finalmente la estructura semántica y optimizada hacia el lenguaje objetivo en la arquitectura donde será ejecutado.

Comenzaremos entonces a estudiar la fase de análisis lexicográfico, donde introduciremos los primeros elementos de la teoría de lenguajes formales.

Contenido

Para intentar responder algunas de las interrogantes planteadas arriba, hemos dividido el contenido del curso en 6 temas fundamentales, aunque existen muchos elementos que son

trans-temáticos y por tanto no quedarían bien ubicados en ningún lugar particular. Por este mismo motivo, estos temas no deben tomarse como particiones rígidas del contenido, sino más bien guías a grandes razgos que giran alrededor de un concepto común. De hecho, estos temas no son presentados estrictamente en orden, sino que se van entrelazando a medida que avanza el curso. Los motivos para esto quedarán más claros adelante.

Teoría de lenguajes: En este tema trataremos sobre los aspectos teóricos de la teoría de lenguajes formales, entre otras cuestiones:

- Definiciones matemáticas y computacionales
- Tipos de lenguajes y características
- Equivalencias y relaciones entre lenguajes
- Problemas interesantes sobre lenguajes
- Relación entre la teoría de lenguajes y la computabilidad

Mecanismos de generación: En este tema trataremos las diferentes clases de gramáticas, sus propiedades, los tipos de lenguajes que pueden generar, y algunos algoritmos de conversión entre ellas:

- Jerarquía de gramáticas de Chomsky
- Gramáticas atributadas
- Formas normales y algoritmos de conversión

Mecanismos de reconocimiento: En este tema trataremos sobre los autómatas como reconocedores de lenguajes, sus propiedades, y los algoritmos de construcción a partir de las gramáticas correspondientes:

- Tipos de autómatas según el tipo de lenguaje
- Conversión entre autómatas y expresiones regulares
- Autómatas para el problema de *parsing*

Análisis semántico: En este tema trataremos los problemas fundamentales de semántica en el contexto de los compiladores y las estrategias de modelación y solución:

- Representaciones semánticas y problemas típicos
- Solución de referencias y *aliases*
- Inferencia de tipos y expresiones
- Validación de pre- y post-condiciones e invariantes

Generación de código: En este tema trataremos las diferentes arquitecturas de código de máquina y los problemas asociados:

- Semánticas operacionales
- Arquitecturas de generación código
- Optimizaciones

Ejecución de código: En este tema trataremos los problemas que surgen o son resueltos posterior a la generación de código, y que dependen de operaciones especiales por parte del mecanismo de ejecución:

- Intérpretes y máquinas virtuales
- Manejo de excepciones

- Recolección de basura

La guía general del curso será orientada a la tarea canónica de construir un compilador. Para ello usaremos el lenguaje `COOL` diseñado específicamente para este propósito. Con esta guía en mente, iremos presentando los contenidos en el orden que sean necesarios para ir avanzando en el propósito de construir el compilador. Por este motivo, a diferencia de otros cursos, no daremos primero toda la teoría de lenguajes formales y luego los algoritmos de *parsing*, sino que iremos introduciendo elementos teóricos y prácticos según lo vaya requiriendo el compilador que estamos diseñando en clase.

El objetivo de esta forma de ordenar los contenidos es, en primer lugar, brindar una visión más unificada de todo el proceso de compilación y todas las técnicas, elementos teóricos y cuestiones de diseño. Por otro lado, aspiramos con esto a lograr que los estudiantes realmente obtengan las habilidades planteadas al inicio, al tener que poner desde el principio del curso todo el conocimiento en función de un proyecto práctico.

Evaluación y retroalimentación

Una vez que los estudiantes saben que habilidades obtendrán, entonces se plantea el problema de la evaluación. En este curso consideramos la evaluación como una necesidad y una ventaja para los estudiantes, no como una obligación o una arbitrariedad del profesor. Por eso hemos diseñado un sistema de evaluación que permita a los estudiantes reconocerlo como un mecanismo de retroalimentación que les permite entender su propio avance y optimizar su estudio. Claro que la evaluación tiene un valor externo, que le permite a la sociedad saber quienes están preparados o no. Pero las notas particulares obtenidas en una asignatura no significan nada fuera del sistema educacional. A nadie se le ofrece un trabajo por las notas que obtuvo. Lo que cuenta es graduarse o no. Entonces, ¿por qué se toma tanto trabajo en cuantificar la evaluación? La respuesta es que el número obtenido en una evaluación es una métrica de retroalimentación para que el estudiante sepa en qué grado fue capaz de cumplir las expectativas del profesor.

Teniendo esta máxima en cuenta, hemos planteado algunos principios básicos que creemos deben guiar el sistema de evaluación del curso. En primer lugar, la idea de que todos los estudiantes deben tener tantas oportunidades como sea posible para experimentar y aprender. Esto significa que deben permitirse fallar sin miedo a ser recriminados. Por tanto, no consideramos ninguna práctica que penalice a los estudiantes por fallar en una evaluación, en particular, todos los estudiantes tienen derecho a todos los exámenes independientemente de los resultados obtenidos anteriormente.

Por otro lado, queremos incentivar un estudio constante y consistente en vez de un maratón de última hora, por lo que sí aprobamos las prácticas que premien por el trabajo continuado, incluso teniendo en cuenta los posibles errores que hayan cometido. De modo que hemos dividido el curso en 3 grandes conjuntos de habilidades:

- Habilidades teóricas de modelación con lenguajes formales.
- Habilidades en el uso y diseño de algoritmos relacionados con lenguajes.
- Habilidades en el diseño de arquitecturas y patrones para sistemas de procesamiento de lenguajes.

De forma aproximada podemos incluir todo contenido dado en el curso en uno de estos conjuntos de habilidades. Consideramos en un estudiante que domine de forma efectiva estos tres conjuntos de habilidades, está preparado para enfrentarse a las preguntas y problemáticas típicas de este campo de estudio. Por el mismo motivo, consideramos que es imprescindible haber vencido las tres habilidades para poder aprobar el curso.

Por tanto, hemos concebido un sistema de evaluación que se compone de 3 exámenes parciales y tres proyectos opcionales, cada par respectivamente orientado a evaluar uno de los tres conjuntos de habilidades mencionados. Los estudiantes que obtienen resultados satisfactorios en cada caso se considera que vencieron dicha habilidad. Al finalizar el curso hay un exámen final que permite a los estudiantes complementar sus resultados hasta el momento, en caso de no haber cumplido todos los objetivos en las evaluaciones parciales.

Así mismo, se orientarán a menudo tareas de menor envergadura que pueden ayudar también a complementar la comprensión (y la consecuente evaluación) en cualquiera de los temas presentados.

Conclusiones

Este curso es un viaje, un viaje por una de las ramas más espectaculares de la historia de la computación, una rama que definió a la computación como ciencia, y que creó algunos de sus héroes más famosos. Es un viaje lleno de dificultades, pero detrás de cada obstáculo hay algo increíble que descubrir. Los que hemos pasado por este viaje les podemos prometer que vale la pena. Pero la mejor forma de experimentarlo no es como espectador, como un simple pasajero. La mejor forma de experimentarlo es coger el timón y decidir ustedes cuáles son los lugares que quieren explorar. Nosotros haremos el mejor esfuerzo por llevarlos allí, pero no les podemos mantener los ojos abiertos. Mirar, oler y tocar todo lo que puedan es responsabilidad de ustedes.

Capítulo 1

Análisis Lexicográfico

En este capítulo vamos a atacar el problema de implementar un *lexer*, y todos los elementos teóricos que nos ayudarán en esta tarea, y que nos servirán además para resolver problemas independientes pero relacionados con esta fase. Vamos a comenzar definiendo formalmente el problema que queremos resolver, y luego veremos como desarrollar una teoría que nos permita atacar su solución. Para ello:

- Definiremos el concepto de *token*.
- Diseñaremos autómatas finitos deterministas para reconocer lenguajes regulares.
- Caracterizaremos los lenguajes regulares.
- Conoceremos las expresiones regulares para describir esta clase de lenguajes.
- Veremos un algoritmo convertir expresiones regulares en autómatas.
- Aprenderemos sobre las operaciones entre lenguajes regulares.
- Y por si fuera poco, veremos finalmente los límites de los lenguajes regulares.

Los *tokens* de nuestro lenguaje

Comenzaremos por definir el concepto de *token*, básicamente, como una secuencia de caracteres con cumple con cierta estructura sintáctica. Más adelante impondremos restricciones sobre qué tipos de estructuras son válidas. De momento, pondremos algunos ejemplos:

- Las palabras claves de un lenguaje: **if**, **else**, **function**, **class**, etc.
- Los literales numéricos: `42`, `3.1415`, `0.123e-44`, etc.
- Los literales de cadena: `"hello world!"`, etc.
- Los literales booleanos: `True`, `False`.
- Los operadores: `+`, `-`, `*`, `<`, `<=`, etc.
- Identificadores: `x`, `value`, `someMethodName`, etc.

De modo que queremos idear un mecanismo que reciba como entrada una descripción del conjunto de *tokens* de nuestro lenguaje, y una cadena de caracteres, y nos devuelva la secuencia de *tokens* que aparecen en la cadena.

Un primer análisis nos puede llevar a pensar idea similar a dividir la cadena por espacios en blanco y quedarnos con cada elemento como un *token*, pero es muy sencillo ver por qué esta idea no es suficiente. En primer lugar, existen *tokens* que no necesariamente aparecen separados por espacio, como en el caso de `value<=0.001`. Además, existen *tokens* que son prefijos de otros *tokens*, como `<` y `<=`; o `class` que es una palabra clave y `class1` que es un identificador cualquiera. Esto quiere decir que necesitamos realizar un análisis un poco más complejo para determinar qué parte de una secuencia de caracteres corresponde a un *token* o a *otro*.

Veamos el problema entonces desde el siguiente punto de vista: supongamos que escaneamos la secuencia de caracteres de inicio a fin, un caracter a la vez, y lo que deseamos es en cada momento ser capaces de determinar si estamos en el medio de un *token*, o si acabamos de reconocer un *token* completo. Por ejemplo, tomemos nuevamente la cadena de ejemplo del capítulo anterior:

```
if (a<=0) b else c
```

Vamos a simbolizar con el símbolo `|` el punto que divide el fragmento de la cadena que hemos analizado, de la parte de la cadena que falta por analizar. Es decir, al inicio de nuestro procesamiento, estamos en el siguiente estado:

```
|if (a<=0) b else c
```

Que simboliza que aún no hemos visto ningún caracter de la cadena. Queremos entonces idear una maquinaria que "avance" por la cadena, caracter a caracter, y en cada instante sepa determinar si con los caracteres vistos hasta el momento ya tiene un *token* construido, o si necesita seguir avanzando. Veamos paso a paso cómo podría funcionar un mecanismo así. Comenzamos con el inicio de la cadena, y vamos además a almacenar el fragmento de *token* que estamos construyendo, y la lista de *tokens* que vamos reconociendo como salida:

```
CADENA: |if a<=0 then b else c
 TOKEN:
SALIDA:
```

El primer paso consiste en analizar el caracter `i`. Cómo acabamos de empezar, no tenemos nada almacenado, entonces la única posibilidad es que este caracter `i` sea parte del *token* que estamos intentando reconocer.

```
CADENA: i|f a<=0 then b else c
 TOKEN: i
SALIDA:
```

Ahora vemos el caracter `f`. Cómo viene immediatamente después de otro caracter, pudiera ser que estemos viendo un *token* de tipo identificador. Por otro lado, pudiera ser que estuviéramos viendo particularmente el *token* `if`, que es una palabra clave. De momento, lo tomamos.

```
CADENA: if| a<=0 then b else c
 TOKEN: if
SALIDA:
```

Nos encontramos entonces con un espacio en blanco, lo que nos permite decidir que efectivamente hemos encontrado el *token* `if`, por lo que podemos devolverlo en la secuencia de salida.

```
CADENA: if |a<=0 then b else c
 TOKEN:
SALIDA: [if]
```

Ahora nos encontramos un caracter `a`, que pudiera ser un identificador, pero todavía no podemos decir nada al respecto pues es posible que el identificador tenga más letras.

```
CADENA: if a|<=0 then b else c
 TOKEN: a
SALIDA: [if]
```

En este paso nos encontramos un caracter `<`, y como estábamos tratando de reconocer un identificador que empieza con `a`, y nuestras reglas dicen que `<` no puede ser parte del nombre de ningún identificador, entonces podemos concluir que el *token* anterior es justamente `a`, y comenzar un nuevo *token*:

```
CADENA: if a<|=0 then b else c
 TOKEN: <
SALIDA: [if] [a]
```

Del mismo modo, hemos comenzado a construir un *token* `<`, que pudiera ser el operador en sí, o pudiera ser el operador `<=`, todo depende de lo que le siga. Al ver a continuación el caracter `=`, concluimos que tenemos el *token* `<=` sin necesidad de seguir buscando, pues no existe ningún *token* con prefijo `<=`.

```
CADENA: if a<=|0 then b else c
 TOKEN:
SALIDA: [if] [a] [<=]
```

Es fácil ver como continúa este proceso, llevando finalmente a la secuencia de *tokens* correcta:

```
[if] [a] [<=] [0] [then] [b] [else] [c]
```

La pregunta consiste entonces en cómo implementamos este mecanismo computacionalmente. De modo general, la decisión que hemos tomado en cada paso depende únicamente de dos factores: el caracter que estamos examinando, y lo que hemos hecho hasta el paso anterior. Para modelar este segundo factor de "lo que hemos hecho hasta el paso anterior", analizemos primero de cuántas formas posibles podemos hablar de lo que hemos hecho hasta el paso anterior. Básicamente, tenemos un conjunto finito de clases de *tokens*, y lo que necesitamos saber es simplemente que clase de *token* estamos intentando construir, para saber si el nuevo caracter puede completar esta clase o no.

De modo que podemos pensar en una especie de **máquina de estados**, donde modelaremos con los estados el hecho de llevar la cuenta de qué clase de *token* estamos construyendo, y por ende, cuáles son los posibles caracteres a esperar a continuación. Esta máquina de estados es una clase particular de máquina de *Turing*, que tiene la característica de que solo puede moverse hacia la izquierda en la cinta y solo puede leer (no está permitido escribir). A las máquinas de estado que solo se mueven en una dirección en la cinta se

les llama **autómatas**. Distintos tipos de autómatas se diferencian entonces por los tipos de transiciones que le son permitidas, y los tipos de análisis que pueden realizar para tomar una decisión. En este caso particular, estaremos definiendo la clase más sencilla de autómata, un **autómata finito determinista**.

Autómatas Finitos Deterministas

Formalizando, un autómata finito determinista es un quíntuplo $A =< Q, q_0, V, F, f >$ con las siguientes características:

- Q es un conjunto finito de estados ($Q = \{q_0, \ldots, q_n\}$), de ahí el adjetivo de **finito**.
- $q_0 \in Q$ es el estado inicial.
- V es un conjunto finito de símbolos que pueden aparecer en la cinta.
- $F \subseteq Q$ es un subconjunto de estados que denominaremos *estados finales*.
- $f : Q \times V \to Q$ es una *función de transición*, que determina, para cada par posible de estados y símbolos, cuál es el estado de destino. Se denomina un autómata **determinista** justamente porque en un estado particular, para un símbolo particular, existe solamente un estado posible de destino (o ninguno), por lo tanto, siempre existe una única decisión que tomar.

El modo de funcionamiento de un autómata finito determinista es el siguiente. Tenemos una cadena de entrada, que no es más que una secuencia de símbolos de V, o sea, un elemento $\omega \in V^k$ para algún valor de k (que será la longitud de la cadena). El estado inicial del autómata es $q^{(0)} = q_0$. Por cada símbolo en $\omega_i \in \omega$, realizamos la operación $q^{(i)} = f(q^{(i-1)}, \omega_i)$. Al concluir, si $q^{(k)} \in V$, decimos que el autómata **acepta** o **reconoce** la cadena ω.

Si en un caso particular la función $f(q_i, a_i)$ no estuviera definida, entonces diremos que el autómata "se traba", y no reconoce la cadena. Este convenio lo tomamos para evitar tener que definir un autómata completamente cuando existen muchos estados "superfluos". Siempre es posible convertir un autómata con algunas transiciones no definidas (*parcialmente especificado*) a un autómata con todas las transiciones definidas (*completamente especificado*), si añadimos un estado adicional q_{error}, a dónde apuntan todas las transiciones faltantes, y de donde salen además para el propio estado q_{error} todas transiciones con todos los símbolos. Este estado sirve como especie de "sumidero".

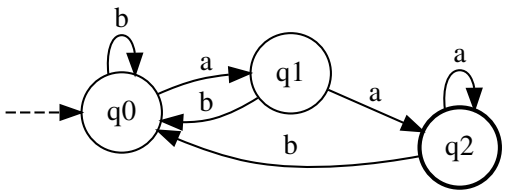
Veamos entonces un ejemplo. Para representar gráficamente un autómata, usaremos un grafo dirigido para visualizar la función de transición, donde el nodo i representa al estado q_i, y la artista (i, j) está etiquetada con el símbolo a_k si $f(q_i, a_k) = q_j$. Los estados finales se representan con un borde doble, y el resto de los estados con un borde normal.

Por ejemplo, si tenemos el siguiente autómata $A =< Q, q_0, V, F, f >$, donde:

- $Q = q_0, q_1, q_2$
- $V = \{a, b\}$
- $F = \{q_2\}$
- f se define por la siquiente tabla:

Estado	Símbolo	f
q_0	a	q_1
q_0	b	q_0
q_1	a	q_2
q_1	b	q_0
q_2	a	q_2
q_2	b	q_0

Podemos representar gráficamente el autómata anterior con el siguiente gráfico:



En este gráfico queda completamente representado el autómata, sin necesidad de especificar el resto, pues de la función de transición se pueden inferir los estados y el alfabeto, y se toma el convenio de que el estado q_0 siempre es el estado inicial.

Cabe preguntarnos entonces la siguiente interrogante:

> ¿Qué lenguajes es capaz de reconocer un autómata finito determinista?

A responder esta pregunta dedicaremos la mayor parte del resto de este capítulo. Para comenzar, debemos formalizar primeramente la pregunta planteada. Necesitamos definir qué es un **lenguaje**, y qué significa que un autómata **reconozca** cierto lenguaje.

Para definir un lenguaje, comenzaremos con la definición de **alfabeto**. Un alfabeto V no es más que un conjunto de símbolos, que son elementos básicos indivisibles. Por ejemplo, $V = \{a, b\}$ es el alfabeto que contiene los símbolos a y b. El "significado" de cada símbolo no nos interesa en la teoría de lenguajes formales, solamente nos interesa diferenciar un símbolo de otro. Dejaremos el problema del "significado" para cuando lleguemos a la fase semántica.

Una vez armados con el concepto de alfabeto, podemos definir el concepto de **cadena**. Una cadena ω es un elemento del conjunto V^k para algún valor de k arbitrario. Por ejemplo, la cadena *abba* es un elemento del conjunto $\{a, b\}^4$. Se llama *longitud* de la cadena justamente al valor k. Si $k = 0$, tenemos la *cadena vacía*, que se simboliza generalmente como ϵ independientemente del conjunto V.

Finalmente, definiremos un **lenguaje** L, como un conjunto de cadenas sobre un alfabeto particular, o alternativamente, como un subconjunto de la clausura del producto cartesiano de dicho de alfabeto. Es decir, $L \subseteq V^*$, donde:

$$V^* = \bigcup_{k=0}^{\infty} V^k$$

Vamos a introducir entonces algo de notación adicional. Sea A un autómata finito determinista, denominaremos L_A al lenguaje de todas las cadenas reconocidas por A. Es decir, $\omega \in L_A$ si y solo A reconoce ω. Cabe entonces preguntarnos qué tipos de lenguajes pueden ser reconocidos por este mecanismo. Podemos entonces definir la interrogante anterior de manera más precisa cómo:

> ¿Qué características tienen los lenguajes que son posibles de reconocer por algún autómata finito determinista?

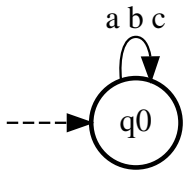
A este tipo de lenguajes les llamaremos **lenguajes regulares**, por motivos históricos que veremos más adelante. A modo definición:

> **Definición (Lenguaje Regular):** Sea L un lenguaje sobre el alfabeto V, se dice que L es **regular** si y solo si existe un autómata finito determinista $A =< Q, q_0, V, F, f >$ (sobre el mismo alfabeto), tal que $L = L_A$.

Pasemos entonces a describir algunos ejemplos de lenguajes regulares, que nos darán una idea intuitiva del poder de cómputo de estos autómatas.

Lenguajes Regulares

El más simple de todos los lenguajes que podemos definir sobre un alfabeto V es justamente V^*, el lenguaje de todas las cadenas que se pueden construir con los símbolos de V. A este lenguaje le llamamos *universo*. Para construir un autómata para este lenguaje, simplemente necesitamos un estado que será a la vez inicial y final, y una transición para cada símbolo del alfabeto desde este estado hacia el propio estado. Por ejemplo, si $V = \{a, b, c\}$, tenemos el siguiente autómata:

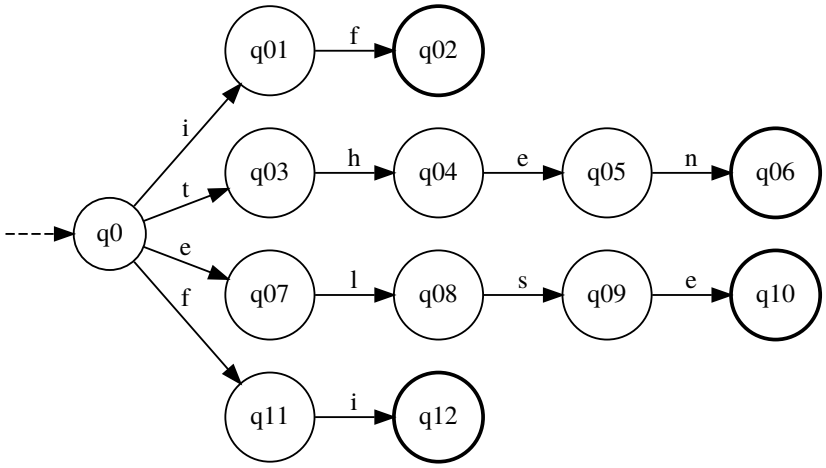


Con esta idea podemos demostrar nuestro primer teorema en la teoría de lenguajes formales:

> **Teorema:** El lenguaje *universo* $L = V^*$ es regular.

La demostración se realiza por construcción. Sea $V = \{a_1, \ldots, a_n\}$ el alfabeto, sea A el autómata $< Q, q_0, F, V, f >$, donde $Q = F = \{q_0\}$ y $f(q_0, a_i) = q_0$ para todo símbolo $a_i \in V$; entonces A reconoce el lenguaje V^*.

Yendo al extremo contrario, vamos a definir el lenguaje que contiene exactamente las palabras claves **if**, **then**, **else**, **fi**. Para este lenguaje también es fácil definir un autómata. Creamos dos estados, uno inicial y uno final, y por cada palabra del lenguaje creamos un camino entre el estado inicial y final con un estado intermedio para cada símbolo de la palabra correspondiente (menos el último, claro):



Este autómata nos da una idea para demostrar nuestro segundo teorema:

> **Teorema:** Sea $L = \{\omega_1, \ldots, \omega_n\}$ un lenguaje *finito*, entonces L es regular.

La demostración es muy sencilla, simplemente necesitamos construir un autómata que contenga un camino por cada cadena del lenguaje. Si existen cadenas con prefijos comunes, los estados correspondientes a los símbolos que participan en los prefijos se comparten (para mantener el determinismo, aunque veremos pronto que esto no es estrictamente necesario).

Para nuestro siguiente ejemplo, vamos a complejizar un poco el lenguaje a reconocer, y de paso veremos una estrategia general para la construcción de autómatas. El lenguaje que queremos reconocer es el lenguaje de las cadenas sobre el alfabeto $V = \{a, b\}$, con exactamente 3 letras a. Antes de lanzarnos a construir el autómata, tratemos de pensar en una solución algorítmica para este problema. De forma general, la estrategia de solución sería algo así:

```
def match(s: str) -> bool:
    count_a = 0

    for c in s:
        if c == 'a':
            count_a += 1

    return count_a == 3
```

Esta solución es correcta, pero padece de un problema que nos hace imposible convertirla

en un autómata finito. El problema es justamente que la cantidad de memoria que usa esta solución es potencialmente *infinita*, o al menos, proporcional al tamaño de la cadena. ¿Por qué? Pues, porque hemos utilizado un contador no acotado superiormente (`count_a`), por lo que, en principio, podemos necesitar hasta $log_2(n)$ bits de memoria para una cadena con longitud $|w| = n$. Claro que en la práctica sabemos que ese entero es de 32 bits, independientemente del tamaño de la cadena, pero la idea que queremos transmitir es que *no hemos acotado la cantidad de memoria*, por lo que no sabemos cuántos estados debería tener un autómata que haga lo mismo que este método.

Una solución mejor consiste en notar que solamente hay 5 posibles situaciones:

- Hemos visto 0 símbolos a.
- Hemos visto 1 símbolo a.
- Hemos visto 2 símbolos a.
- Hemos visto 3 símbolos a.
- Hemos visto 4 símbolos *a o más*.

Una vez que hayamos visto 4 veces una a en la cadena, ya sabemos que la cadena es incorrecta, y no es posible que "se arregle" más adelante pues la cantidad de a solo puede aumentar. Con esta idea en mente, cambiemos el método anterior para que solo use, a lo sumo, **5 valores diferentes** de la variable `count_a`:

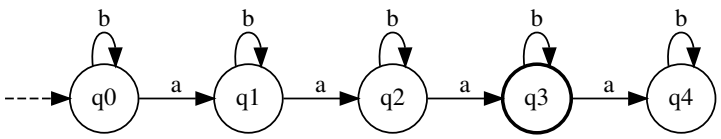
```
def match(s: str) -> bool:
    count_a = 0

    for c in s:     # esto es lo nuevo
        if c == 'a' and count_a <= 3:
            count_a += 1

    return count_a == 3
```

Parece que no hemos hecho mucho, pero hemos logrado garantizar que nuestro método solamente tiene a lo sumo **5 valores diferentes** de la variable `count_a`, a saber, $0 \ldots 4$. ¿De qué nos sirve esto? Pues, ahora podemos diseñar un autómata tranquilamente, con la siguiente idea: contamos la cantidad posible de valores que toman *todas las variables locales*, y por *cada combinación de valores* tendremos un estado. Si este número es finito, entonces tenemos un autómata finito (que puede ser o no determinista, ya llegaremos a ese problema).

En este ejemplo, tenemos entonces 5 posibles estados, desde q_0 hasta q_4. Para determinar las transiciones, veremos cómo cambian los valores en nuestro método. En todo momento, si tenemos un valor `count_a = qi`, en la siguiente iteración o bien `count_a = qi` o de lo contrario `count_a = qi + 1`, y esto solo ocurre si el símbolo recién "visto" es a. Esto nos dice entonces que tenemos entre los estados q_i y q_{i+1} siempre una transición con a, *excepto* en q_4, pues en ese caso no se cumple la condición `count_a <= 3`. Solo nos queda definir el estado final, y en este ejemplo no puede estar más claro, pues la línea `return count_a == 3` nos dice exactamente qué estado (valor de `count_a`) corresponde a un valor `True`. De modo que tenemos nuestro autómata:



Lo que hemos hecho es un ejemplo de lo que informalmente llamamos "programar con autómatas". La idea fundamental radica en pensar en una solución algorítmica al problema de reconocer una cadena de cierto lenguaje L, y luego extraer de esta solución un autómata. Para que funcione, informalmente, la solución algorítmica debe cumplir:

1. Consiste en un método que recibe una cadena y devuelve verdadero o falso.
2. El método consiste en **una sola iteración** sobre la cadena, que analiza cada símbolo exactamente una vez.
3. La memoria local de dicho método, para cualquier cadena, *tanto si pertenece o no* al lenguaje, debe alcanzar a lo sumo una cantidad **finita** de valores diferentes (estos serán los estados de nuestro autómata).

Para obtener el autómata correspondiente, simplemente contamos *todos* los posibles valores que pueden alcanzar todas las variables locales de nuestro método, y definimos un estado por cada uno. Luego, analizando cómo cambian estos valores en función del símbolo siguiente en la cadena, se definen las transiciones (asumamos que son deterministas de momento). Los estados finales estarán determinidos por las combinaciones de valores para los cuáles el método devuelve `True`.

De manera general, informalmente, podemos decir que los lenguajes regulares son todos aquellos cuyas cadenas pueden ser reconocidas mediante un método (función en un lenguaje de programación) que, usando *un solo recorrido* sobre la cadena, y una *cantidad finita de memoria (independiente del tamaño de la cadena)* es capaz de determinar *para toda cadena*, si pertenece o no al lenguaje. Si tal método existe, podremos construir un autómata equivalente.

Expresiones Regulares

Ahora que ya sabemos cómo construir autómatas para cualquier lenguaje regular, y sabemos (intuitivamente) que el lenguaje de los *tokens* de un lenguaje de programación se puede definir como un lenguaje regular, nos dedicamos a la pregunta de cómo es mejor definir ese lenguaje. Evidentemente, podemos diseñar un autómata, de forma manual, que reconozca los *tokens*, pero esta tarea, incluso para los lenguajes más sencillos, es muy engorrosa. El problema radica en los autómatas son un mecanismo de cómputo, podemos decir, un mecanismo *procedural* o *imperativo*, para definir lenguajes. Para definir un autómata hay que decidir una cantidad de estados, el conjunto de estados finales y las transiciones. Todo esto no parece intuitivo de antemano. Por otro lado, un autómata es muy difícil de "leer" para un humano. Al mirar un autómata, identificar rápidamente el lenguaje que reconoce no es una tarea sencilla, mucho menos intuitiva.

De forma alternativa, pudiéramos desear tener un mecanismo *declarativo* que nos permita definir lenguajes de forma más sencilla, y de dónde sea posible de forma automática obtener

el autómata correspondiente. Como sabemos de la experiencia anterior en otros lenguajes de programación, los humanos somos muchos mejores mientras mayor sea el nivel de abstracción, y las máquinas son mucho mejores mientras menor sea el nivel abstracción. ¡Justamente por esta diferencia es que surgió la ciencia de la compilación en sí misma! Estamos volviendo a enfrentarnos al mismo problema original, pero en una escala menor. Cabe preguntarnos entonces si podemos crear un mecanismo de "alto nivel" para definir lenguajes regulares, y una especie de "compilador" que transforme este mecanismo de alto nivel al autómata correspondiente.

Convirtiendo expresiones regulares a autómatas finitos

Operaciones entre lenguajes regulares

Los límites de los lenguajes regulares

Capítulo 2

Parsing Descendente (*Top-Down*)

Hasta el momento hemos visto el proceso de compilación a grandes razgos, y hemos definido que la primera fase consiste en el análisis sintáctico. Esta fase a su vez la hemos divido en 2 procesos secuenciales: el análisis lexicográfico (*tokenización* o *lexing*), y el análisis sintáctico en sí (*parsing*). En esta sección nos concentraremos en este segundo proceso.

Recordemos que el proceso de parsing consiste en analizar una secuencia de tokens, y producir un árbol de derivación, lo que es equivalente a producir una derivación extrema izquierda o derecha de la cadena a reconocer. Tomemos la siguiente gramática, que representa expresiones aritméticas (sin ambigüedad):

```
E = T
  | T + E

T = int * T
  | int
  | (E)
```

Tomemos la cadena siguiente:

```
2 * (3 + 5)
```

Que una vez procesada por la fase lexicográfica produce la siguiente secuencia de *tokens*.

```
int  *  (  int  +  int  )
```

Intentemos producir una derivación (en principio, extrema izquierda, por comodidad) de esta cadena. Como sabemos, si existe una derivación extrema izquierda, es porque se cumple que:

```
[1]  E -*-> int * ( int + int )
```

Preguntémonos entonces, ¿de cuántas formas pudiera `E` derivar en la cadena? Evidentemente, hay exactamente dos formas en que `E` es capaz de producir esta cadena, es decir,

hay solamente dos producciones posibles que pudieran seguir en la derivación extrema izquierda. O bien `E -> T` o bien `E -> T + E`. Probemos entonces con la primera de ellas:

```
[2]  E -*-> T -*-> int * ( int + int )
```

Como estamos produciendo una derivación extrema izquierda, tenemos que expandir a continuación el símbolo `T`. Nuevamente, hay varias opciones, probemos con la primera:

```
[3]  E -*-> int * T -*-> int * ( int + int )
```

De momento parece que vamos por buen camino, pues hemos logrado producir los primeros 2 *tokens* de la cadena. Expandimos entonces nuevamente el no-terminal más a la izquierda con la primera producción posible `T -> int * T`:

```
[4]  E -*-> int * int * T -*-> int * ( int + int )
```

En este punto, podemos darnos cuenta de que hemos tomado el camino equivocado. Como estamos produciendo una derivación extrema izquierda, y los terminales no derivan en ningún símbolo, sabemos que todo lo que esté a la izquierda del primer no-terminal no va a cambiar en el futuro. Luego, es evidentemente que ya no seremos capaces de generar la cadena, pues estos terminales (`int * int *`) no son prefijo de la cadena a reconocer. Deshagamos entonces la última producción (volviendo al paso 3) y probemos otro camino (`T -> int`):

```
[4]  E -> int * int -*-> int * ( int + int )
```

Nuevamente, hemos generado una secuencia de *tokens* que no es prefijo de la cadena. Probamos de nuevo:

```
[4]  E -*-> int * ( E ) -*-> int * ( int + int )
```

Parece que en este punto hemos hecho un avance, pues logramos reconocer tres *tokens* de prefijo. Expandimos entonces el nodo `E -> T` nuevamente, y de ahí probamos la próxima producción (`T -> int * T`), que también falla:

```
[5]  E -*-> int * ( T ) -*-> int * ( int + int )
[6]  E -*-> int * ( int * T ) -*-> int * ( int + int )
```

Probando con cualquiera de las producciones de `T` nunca podremos generar el *token* `+` que falta, por lo tanto eventualmente volveremos a probar con `E -> T + E`:

```
[5]  E -*-> int * ( T + E ) -*-> int * ( int + int )
```

Una vez llegados a este punto, ya podemos hacernos una idea de cómo funciona este proceso. Eventualmente, tendremos que derivar `T -> int` y `E -> T -> int` para lograr producir la cadena final. Finalmente obtenemos la derivación extrema izquierda siguiente:

```
E -> T
  -> int * T
  -> int * ( E )
  -> int * ( T + E )
  -> int * ( int + E )
  -> int * ( int + T )
  -> int * ( int + int )
```

Básicamente, lo que hemos hecho ha sido probar todas las posibles derivaciones extrema izquierda, de forma recursiva, podando inteligentemente cada vez que era evidente que habíamos producido una derivación incorrecta. Tratemos de formalizar este proceso.

Parsing Recursivo Descendente

De forma general, tenemos tres tipos de operaciones o situaciones que analizar:

- La expansión de un no-terminal en la forma oracional actual.
- La prueba recursiva de cada una de las producciones de este no-terminal.
- La comprobación de que un terminal generado coincide con el terminal esperado en la cadena.

Para cada una de estas situaciones, vamos a tener un conjunto de métodos. Para representar la cadena a reconocer, necesitamos mantener un estado global que indique la parte reconocida de la cadena. Diseñemos una clase para ello:

```
interface IParser {
    bool Parse(Token[] tokens);
}

class RecursiveParser : IParser {
    Token[] tokens;
    int nextToken;

    //...
}
```

Para reconocer un terminal, tendremos un método cuya función es a la vez decidir si se reconoce el terminal, y avanzar en la cadena:

```
bool Match(Token token) {
    return tokens[nextToken++] == token;
}
```

A cada no-terminal vamos a asociar un método recursivo cuya función es determinar si el no-terminal correspondiente genera una sub-cadena "adecuada" del lenguaje. Por ejemplo, para el caso de la gramática anterior, tenemos los siguientes métodos:

```
bool E() {
    // Parsea un no-terminal E
}

bool T() {
    // Parsea un no-terminal T
}
```

La semántica de cada uno de estos métodos es que devuelven `true` si y solo si el no-terminal correspondiente genera una parte de la cadena, comenzando en la posición `nextToken`.

Tratemos de escribir el código del método `E`. Para ello, recordemos que el símbolo `E` deriva en 2 producciones: `E -> T` y `E -> T + E`. Por tanto, de forma recursiva podemos decir que `E` genera esta cadena si y solo si la genera a partir de una de estas dos producciones. El primer caso es fácil: `E` genera la cadena a partir de derivar en `T` si y solo si `T` a su vez genera dicha cadena, y ya tenemos un método para eso:

```
bool E1() {
    // E -> T
    return T();
}
```

Para el segundo caso, notemos que la producción `E -> T + E` básicamente lo que dice es: necesitamos generar una cadena a partir de `E`, de forma tal que primero se genere una parte con `T`, luego se genere un `+` y luego se genere otra parte con `E`. Dado que ya tenemos todos los métodos necesarios:

```
bool E2() {
    // E -> T + E
    return T() && Match(Token.Plus) && E();
}
```

Aprovechamos el operador `&&` con cortocircuito para podar lo antes posible el intento de generar la cadena, de forma que el primero de estos tres métodos que falle ya nos permite salir de esa rama recursiva. Ahora que tenemos métodos para cada producción, podemos finalmente develar el cuerpo del método `E`:

```
bool E() {
    int currToken = nextToken;
    if (E1()) return true;

    nextToken = currToken;
    if (E2()) return true;

    return false;
}
```

Este método simplemente prueba cada una de las producciones en orden, teniendo cuidado de retornar `nextToken` a su valor original tras cada llamado recursivo fallido. Así mismo, podemos escribir el método asociado al símbolo `T`, basado en los métodos correspondientes a cada producción:

```
bool T1() {
    // T -> int * T
    return Match(Token.Int) && Match(Token.Times) && T();
}

bool T2() {
    // T -> int
    return Match(Token.Int);
}
```

```
bool T3() {
    // T -> (E)
    return Match(Token.Open) && E() && Match(Token.Closed)
}
```

Y el método general para `T` queda así:

```
bool T() {
    int currToken = nextToken;
    if (T1()) return true;

    nextToken = currToken;
    if (T2()) return true;

    nextToken = currToken;
    if (T3()) return true;

    return false;
}
```

Es posible hacer estos métodos más compactos introduciendo un nuevo método auxiliar:

```
bool Reset(int pos) {
    nextToken = pos;
    return true;
}
```

Este método nos permite reescribir cada método con una sola expresión, haciendo uso del operador `||` con cortocircuito:

```
bool E() {
    int n = nextToken;
    return E1() || Reset(n) && E2();
}

bool T() {
    int n = nextToken;
    return T1() || Reset(n) && T2() || Reset(n) && T3();
}
```

Finalmente, para reconocer la cadena completa, solo nos queda garantizar que se hayan consumido todos los *tokens*:

```
bool Parse(Token[] tokens) {
    this.tokens = tokens;
    nextToken = 0;

    return E() && nextToken == tokens.Length;
}
```

Usualmente por comodidad se asume que existe un terminal especial `$` generado directamente por el Lexer al final de toda cadena. Esto se hace para poder generalizar desde el punto de vista teórico, y no tener que especificar siempre si estamos al final de la cadena o no. Si nuestro Lexer produce este token, al que llamaremos `Token.EOF` en el código, entonces podemos reescribir el método anterior de la siguiente forma:

```
bool Parse(Token[] tokens) {
    this.tokens = tokens;
    nextToken = 0;

    return E() && Match(Token.EOF); // <-- Este es el cambio
}
```

Esta metodología para crear parsers recursivos descendentes puede ser aplicada fácilmente a cualquier gramática libre del contexto. Sin embargo, no todas las gramáticas pueden ser reconocidas de esta forma. Según la estructura de la gramática, es posible que el parser definido no funcione correctamente.

Por ejemplo, para gramáticas ambigüas, el parser (si termina) dará alguna de las derivaciones extrema izquierda posibles, en función del orden en que hayan sido definidas las producciones. Esto se debe al uso de operadores con cortocircuito. Es posible modificar este tipo de parsers fácilmente para generar no solo la primera sino todas las derivaciones extrema izquierda disponibles, simplemente reemplazando los operadores `||` más externos.

Consideremos ahora la siguiente gramática:

```
S -> Sa | b
```

Para esta gramática, el parser recursivo descendente queda de la siguiente forma (simplificada):

```
bool S() {
    int c = nextToken;
    return S() && Match(Token.A) || Reset(n) && Match(Token.B);
}
```

El problema evidente con este parser es que al intentar reconocer el símbolo `S` el algoritmo cae en una recursión infinita. Este tipo de gramáticas se denominan gramáticas con recursión izquierda, que definiremos así:

> **Definición:** Una gramática libre del contexto `G=<S,N,T,P>` se dice recursiva izquierda si y solo si `S -*-> Sw` (donde `w` es una forma oracional).

La forma más sencilla de las gramáticas recursivas izquierdas es cuando existe directamente una producción `S -> Sw`. A este caso le llamamos *recursión izquierda directa*. Para este caso, es posible eliminar la recursión izquierda de forma sencilla. Tomemos nuevamente la gramática anterior:

```
S -> Sa | b
```

Es fácil ver que esta gramática general el lenguaje `ba*`. Otra gramática que genera dicho lenguaje sin recursión izquierda es:

```
S -> bX
X -> aX | epsilon
```

Aún cuando `a` y `b` son formas oracionales en general, y no simplemente terminales, el patrón anterior es válido. De forma general, si una gramática tiene recursión izquierda de la forma:

```
S -> Sa1 | Sa2 | ... San | b1 | b2 | ... | bm
```

Es posible eliminar la recursión izquierda con la transformación:

```
S -> b1X | b2X | ... | bmX
X -> a1X | a2X | ... | anX | epsilon
```

Para el caso más general de recursión izquierda indirecta, también existe un algoritmo para su eliminación, pero de momento no lo presentaremos :(.

El algoritmo de parsing que hemos desarrollado resuelve, al menos de forma teórica, el problema de construir el árbol de derivación. Aunque el código presentado no construye explícitamente el árbol de derivación, es bastante fácil modificarlo al respecto. Sin embargo, aunque en principio el problema ha sido resuelto, el algoritmo recursivo descendente es extremadamente ineficiente. El problema es que, en principio, es necesario probar con todos los árboles de derivación posibles antes de encontrar el árbol correcto. De cierta forma, para resolver el problema de parsing lo que hemos hecho es buscar entre todos los posibles programas, cuál de ellos tiene una representación textual igual a la cadena deseada.

Parsing Predictivo Descendente

Idealmente, quisiéramos un algoritmo de parsing que construya el árbol de derivación con un costo lineal con respecto a la cadena de entrada. Para ello, necesitamos poder "adivinar", en cada método recursivo, cuál es la rama adecuada a la que descender. Con vistas a resolver este problema, consideremos nuevamente la gramática vista en la sección anterior:

```
E -> T | T + E
T -> int * T | int | (E)
```

Analicemos ahora nuevamente la cadena de *tokens* `int * ( int + int )`, y tratemos de "adivinar" en cada paso de una derivación extrema izquierda qué producción es necesario aplicar. Tengamos en cuenta que en cada paso del proceso de parsing, hay al menos un *token* que conocemos tiene que ser generado de inmediato (dado que la gramática no puede "intercambiar" *tokens* una vez generados). Por tanto, observando el siguiente *token* que es necesario generar (`nextToken` en nuestra implementación), tenemos una pista de cuáles producciones no son posibles. En el caso anterior, el primer *token* a generar es `int`. Por tanto, es evidente que ninguna producción que derive en `(E)` funciona, pues el `(` no coincidirá con el *token* `int`. La primera producción a aplicar tiene que derivar en una forma oracional que comience por `int`.

Desafortunadamente existen varias producciones que generan un `int` al inicio. Tanto `T -> int` como `T -> int * T` pudieran ser escogidas. Aún más, a cualquiera de estas dos producciones se llega tanto por `E -> T` como por `E -> T + E`. Por tanto, hay varios caminos por los cuáles se pudiera generar el primer *token* `int`. Intuitivamente, esto se debe a que la

gramática no está **factorizada**. Informalmente, llamaremos a una gramática **factorizada a la izquierda** si las producciones de cualquier símbolo, dos a dos, no comparten ningún prefijo.

En muchas ocasiones es fácil factorizar una gramática. Se introduce un no-terminal nuevo por cada grupo de producciones que compartan un prefijo común. Se reemplazan dichas producciones por una sola producción nueva que contiene el prefijo común, y el nuevo no-terminal se hace derivar en todos los posibles sufijos:

```
E -> T X
X -> + E | epsilon
T -> int Y | (E)
Y -> * T | epsilon
```

Por supuesto, es posible que la relación entre los prefijos sea más complicada, y una vez que se realice la transformación anterior aún queden producciones no factorizadas (e.j. `X -> abC | abD | aY`). Incluso en estos casos es posible factorizar la gramática aplicando varias veces el proceso de factorización anterior.

Esta modificación evidentemente no cambia el lenguaje, y ni siquiera cambia la ambigüedad o no de la gramática. Simplemente nos permite delegar la decisión de qué producción tomar un *token* hacia adelante. Si antes no sabíamos cuando venía `int` que producción tomar, porque podía ser `T -> int` o `T -> int * T`, ahora simplemente reconocemos el primer `int`, y delegamos la decisión de generar `epsilon` o `* T` a un nuevo no-terminal.

Desde el punto de vista del diseño de lenguajes, el beneficio de este cambio es discutible. Por un lado nos permite aplicar un algoritmo que de otra forma no funcionaría. Sin embargo, por otra parte, estamos provocando un cambio en el diseño de la gramática, que es una cuestión de "alto nivel", para poder usar un algoritmo particular, que es una cuestión de "bajo nivel". En otras palabras, estamos cambiando el diseño en función de la implementación. Este cambio puede tener efectos adversos. Por ejemplo, nuestra gramática para expresiones aritméticas es ahora más difícil de entender, pues contiene símbolos "extraños" que no significan nada desde el punto de vista semántico, solamente están ahí para simplificar la implementación. El árbol de derivación ahora es más complejo. Más adelante discutiremos esta problemática en mayor profundidad.

Consideremos ahora nuevamente el proceso de parsing, y tratemos de ver si es posible adivinar en todo caso cuál producción aplicar. Tomemos como ejemplo la cadena `int * ( int + int )`, y tratemos de generar la derivación extrema izquierda:

```
E -*-> int * ( int + int )
```

La primera producción tiene que ser necesariamente `E -> T X` pues es la única disponible:

```
E -*-> T X -*-> int * ( int + int )
```

Ahora, tenemos que expandir el primer símbolo `T`. Afortunadamente, sabemos que esta expansión obligatoriamente genera un *token* a continuación, ya sea `int` o `(`. Por tanto es trivial escoger la única producción posible:

```
E -*-> int Y X -*-> int * ( int + int )
```

El nuevo símbolo a expandir es `Y`, y ahora se nos complica un poco el análisis. `Y` bien pudiera desaparecer (`Y -> epsilon`) o generar un `* T`. Sabemos que la producción `* T` nos genera el token que queremos. Pero la pregunta es, ¿cómo sabemos que la producción `Y -> epsilon` no pudiera redundar en que eventualmente aparezca ese `*` por otro lado? Observando la gramática, intuitivamente, podemos ver que si `Y` desaparece, `X` nunca podrá poner un `*` justamente en esa posición, ya que `X` tiene que generar primero un `+` (`X -> + E`) antes de que aparezca otro no-terminal que pudiera generar el `*`. Por tanto, la única producción posible es `Y -> * T`:

```
E -*-> int * T X -*-> int * ( int + int )
```

En este punto es fácil ver que la única solución es derivar `T -> ( E )`, pues `T` nunca desaparece:

```
E -*-> int * ( E ) X -*-> int * ( int + int )
```

Ahora volvemos al punto inicial:

```
E -*-> int * ( T X ) X -*-> int * ( int + int )
```

Nuevamente `T` tiene que generar un `int`:

```
E -*-> int * ( int Y X ) X -*-> int * ( int + int )
```

Evidentemente `Y` no puede generar el token `+` que hace falta, así que solo puede desaparecer:

```
E -*-> int * ( int X ) X -*-> int * ( int + int )
```

Volvemos entonces a la situación complicada anterior. Es cierto que `X -> + E` nos sirve, pero ¿cómo sabemos que es la única opción? ¿Es posible que de `X -> epsilon` se logre en algún momento que aparezca un `+`. Si miramos la forma oracional generada hasta el momento, vemos que no nos queda otra `X` dentro los paréntesis que pueda poner el `+` que falta. Sin embargo, este análisis no lo puede hacer nuestro algoritmo, que solamente conoce el no-terminal actual, y el siguiente token que es necesario generar. Tratemos de hacer un razonamiento un poco más generalizable. La pregunta que estamos haciendo aquí básicamente es si es conveniente eliminar `X` con la esperanza de que aparezca un `+` de lo que sea que venga detrás. Más adelante formalizaremos este concepto, pero por ahora baste decir que, intuitivamente, podemos ver que detrás de una `X` solamente puede venir o bien un `)` o bien el fin de la cadena. Por tanto, no queda otra opción que derivar `X -> + E`:

```
E -*-> int * ( int + E ) X -*-> int * ( int + int )
```

Generar el siguiente `int` es fácil:

```
E -*-> int * ( int + T X ) X -*-> int * ( int + int )
E -*-> int * ( int + int Y X ) X -*-> int * ( int + int )
```

Finalmente nos queda por generar un `)`. Claro que si miramos la forma oracional generada, sabemos que es necesario eliminar `Y` y `X`, pero recordemos que nuestro algoritmo no puede ver tan hacia adelante. De todas formas, no hace falta mirar más, ni `Y` ni `X` son capaces de generar nunca un `)`, así que ambos símbolos desaparecen:

```
E -*-> int * ( int + int X ) X -*-> int * ( int + int )
E -*-> int * ( int + int ) X -*-> int * ( int + int )
```

Y finalmente la última `X` debe desaparecer también pues se ha generado toda la cadena. Finalmente nos queda:

```
E -> T X
  -> int Y X
  -> int * T X
  -> int * ( E ) X
  -> int * ( T X ) X
  -> int * ( int Y X ) X
  -> int * ( int X ) X
  -> int * ( int + E ) X
  -> int * ( int + T X ) X
  -> int * ( int + int Y X ) X
  -> int * ( int + int X ) X
  -> int * ( int + int ) X
  -> int * ( int + int )
```

La derivación extrema izquierda producida es considerablemente mayor con esta gramática factorizada, dado que existen más producciones. Sin embargo, ganamos en un factor exponencial al eliminar el *backtrack* por completo. Intuitivamente, el largo de esta derivación debe ser lineal con respecto a la longitud de la cadena de entrada, pues a lo sumo en cada paso o bien generamos un nuevo *token* o derivamos el símbolo más izquierdo en nuevos símbolos. Dado que no tenemos recursión izquierda, estas operaciones con cada símbolo no pueden ser "recursivas". Es decir, si el no-terminal más izquierdo es `Y`, y empezamos a derivarlo, eventualmente terminaremos con ese `Y`, ya sea produciendo un terminal o derivando en `epsilon`. De forma general, el costo está acotado superiormente por `|w| * |N|`, pues no es posible que para generar un *token* sea necesario usar `|N| + 1` no terminales, ya que en ese caso tendríamos un no-terminal derivando en sí mismo (al menos de forma indirecta), lo que contradice que la gramática no tenga recursión izquierda.

Tratemos ahora de formalizar este proceso de "adivinación" de qué producción aplicar en cada caso. De forma general nos hemos enfrentado a dos interrogantes fundamentalmente distintas:

- Saber si alguna de las producciones de X puede derivar en una forma oracional cuyo primer símbolo sea el terminal que toca generar.
- Si `X -> epsilon`, saber si esta derivación puede potencialmente redundar en que "lo que sea que venga detrás" de `X` genere el terminal que toca.

Si para las preguntas anteriores obtenemos una sola producción como respuesta, entonces podemos estar seguros de la decisión a tomar. En caso de que obtengamos más de una respuesta, nuestro algoritmo no podrá decidir qué producción tomar, y será inevitable el *backtrack*. Para encontrar una respuesta a estas preguntas, intentemos formalizar estos conceptos de "`X` puede derivar en..." y "lo que venga detrás de `X`...".

Llamaremos `First(W)` al conjunto de todos los terminales que pueden ser generados por `W` como primer elemento (siendo `W` una forma oracional cualquiera, no solamente un no-terminal). Formalmente:

> **Definición**: Sea $G =< S, N, T, P >$ una gramática libre del contexto, $W \in$

> $N \cup T^*$ una forma oracional, y $x \in T$ un terminal. Decimos que $x \in First(W)$ si y solo si $W \rightarrow^* xZ$ (donde $Z \in \{N \cup T\}^*$ es otra forma oracional).

Este concepto captura formalmente la noción de "comenzar por". De forma intuitiva, si logramos computar el conjunto `First(W)` para todas las producciones `X -> W` de nuestra gramática, y cada uno de estos conjuntos de las producciones del mismo símbolo son disjuntos dos a dos, entonces podremos decir inequívocamente qué producción aplicar para generar el terminal que toca (o cuando no es posible generarlo). Notemos que fue necesario definir `First(W)` no solo para un no-terminal, sino para una forma oracional en general, pues necesitamos computarlo en toda parte derecha de una producción.

Por otro lado, la noción de "lo que viene detrás" se formaliza en un concepto similar, denominado `Follow(X)`. En este caso solo necesitamos definirlo para un no-terminal, pues solo nos interesan las producciones `X -> epsilon`. Informalmente diremos que el `Follow(X)` son todos aquellos terminales que pueden aparecer en cualquier forma oracional, detrás de un no-terminal `X`. Formalmente:

> **Definición**: Sea $G =< S, N, T, P >$ una gramática libre del contexto, $X \in N$ un no-terminal, y $x \in T$ un terminal. Decimos que $x \in Follow(X)$ si y solo si $S - * - > WXxZ$ (donde $W, Z \in \{N \cup T\}^*$ son formas oracionales cualesquiera).

La definición de `Follow(X)` básicamente nos dice que si en algún momento el terminal que queremos generar `x` está justo detrás del no-terminal `X` que toca expandir, entonces `X -> epsilon` es una producción válida a aplicar, porque existe la posibilidad de que otro no-terminal genere a `x` justo en esa posición (aunque en la cadena particular que se está reconociendo puede que esto no sea posible).

Supongamos entonces que tenemos todos estos conjuntos calculados (o potencialmente calculables en cualquier momento). ¿Cómo podemos utilizarlos para guiar la búsqueda del árbol de derivación correcto? Veamos como podría quedar el método recursivo descendente para generar `T` en esta nueva gramática:

```
bool T() {
    // T -> int Y
    if (tokens[currToken] == Token.Int)
        return Match(Token.Int) && Y();

    // T -> ( E )
    else if (tokens[currToken] == Token.Open)
        return Match(Token.Open) && E() && Match(Token.Close);

    return false;
}
```

Como `T` siempre genera un token, es fácil decidir qué camino escoger. Por otro lado, en la expansión de `X`, es posible que sea necesario escoger `X -> epsilon`. En este caso, el método recursivo sería:

```
bool X() {
    // X -> + E
```

```
    if (tokens[nextToken] == Token.Plus)
        return Match(Token.Plus) && E()

    // X -> epsilon
    else if (Follow("X").Contains(tokens[currToken]))
        return true;

    return false;
}
```

En el caso de `X -> epsilon`, simplemente retornamos `true` de inmediato y no consumimos el terminal correspondiente.

De forma general, podemos escribir cualquier método recursivo descendente de la siguiente forma (asumimos algunos métodos y clases utilitarios que no presentaremos formalmente):

```
bool Expand(NonTerminal N) {
    foreach(var p in N.Productions) {
        if (!p.IsEpsilon && First(p).Contains(tokens[nextToken]))
            return MatchProduction(p);

        if (p.IsEpsilon && Follow(N).Contains(tokens[nextToken]))
            return true;
    }

    return false;
}
```

El método `MatchProduction` puede a grandes razgos implementarse de la siguiente forma:

```
bool MatchProduction(Production p) {
    foreach(var symbol in p.Symbols) {
        if (symbol.IsTerminal && !Match(symbol as Token))
            return false;

        if (!symbol.IsTerminal && !Expand(symbol as NonTerminal))
            return false;
    }

    return true;
}
```

En todos estos casos hemos asumido que la primera producción aplicable era la única posible. Para ello deben cumplirse ciertas restricciones entre los conjuntos `First` y `Follow` que formalizaremos a continuación.

Gramáticas LL(1)

Llamaremos gramáticas LL(1) justamente a aquellas gramáticas para las cuales el proceso de cómputo de `First` y `Follow` descrito informalmente en la sección anterior nos permite construir un parser que nunca tenga que hacer *backtrack*. El nombre LL(1) significa *left-to-right left-derivation look-ahead 1*. Es decir, la cadena se analiza de izquierda a derecha, se construye una derivación extrema izquierda, y se analiza un solo *token* para decidir que producción aplicar. De forma general, existen las gramáticas LL(k), donde son necesarios k *tokens* para poder predecir que producción aplicar. Aunque los principios son los mismos, el proceso de construcción de estos conjuntos es más complejo, y por lo tanto no analizaremos estas gramáticas por el momento :(.

Para poder formalizar este concepto, será conveniente primero encontrar algoritmos explícitos para computar los conjuntos `First` y `Follow`. Comencemos por el `First` ;). Veamos primero algunos hechos interesantes que se cumplen en este conjunto, y luego veremos cómo se diseña un algoritmo para su cómputo. No presentaremos demostración para estos hechos, pues la mayoría son intuitivos.

- Si $X \to W1|W2|\ldots|Wn$ entonces por definición, $First(X) = \cup First(W_i)$.
- Si $X \to^* \epsilon$ entonces $\epsilon \in First(X)$.
- Si $W = xZ$ donde x es un terminal, entonces trivialmente $First(W) = x$.
- Si $W = YZ$ donde Y es un no-terminal y Z una forma oracional, entonces $First(Y) \subseteq First(W)$.
- Si $W = YZ$ y $Y \to^* \epsilon$ entonces $First(Z) \subseteq First(W)$.

Las observaciones anteriores nos permiten diseñar un algoritmo para calcular todos los conjuntos `First(X)` para cada no-terminal `X`. Como de forma general pueden existir producciones recursivas, calcularemos todos los conjuntos `First` a la vez, aplicando cada una de las "reglas" anteriores, hasta que no se modifique ninguno de los conjuntos `First`. Nuevamente abusaremos de la imaginación y creatividad para introducir métodos y clases utilitarias sin definirlos de manera formal.

```
Firsts CalculateFirsts(Grammar G) {
    var Firsts = new Firsts(); // Parecido a un Diccionario

    // Calculamos el First de cada terminal
    foreach(var t in G.Terminals) {
        Firsts[t] = new FirstSet() { t };
    }
    foreach(var T in G.NonTerminals) {
        Firsts[T] = new FirstSet(); // Parecido a un HashSet
    }

    bool changed;

    do {
        changed = false;
```

```
        // Vamos por cada producción
        foreach(var p in G.Productions) {
            // X -> W
            var X = p.Left;
            var W = p.Right;

            if (p.IsEpsilon) { // X -> epsilon
                changed = Firsts[X].Add(epsilon);
            }
            else {
                bool allEpsilon = true;

                foreach(var s in W) {
                    // Agregamos todo en el First(s)
                    changed = Firsts[X].AddAll(Firsts[s]);

                    // Si s_i deriva en epsilon,
                    // agregamos también el First(s_i+1)
                    if (!Firsts[s].Contains(epsilon)) {
                        allEpsilon = false;
                        break;
                    }
                }

                // Si todos los s_i derivan en epsilon
                // entonces epsilon pertenece al First(X)
                if (allEpsilon) {
                    changed = Firsts[X].Add(epsilon);
                }
            }
        }
    } while (changed);

    return Firsts;
}
```

El algoritmo anterior computa todos los conjuntos `First` de todos los terminales y no-terminales a la vez. Hemos supuesto la existencia de estructuras de datos `Firsts` y `FirstSet` con operaciones convenientes para ello. Estas estructuras se implementan fácilmente usando diccionarios y conjuntos. Una vez obtenidos todos los `First` anteriores, podemos calcular fácilmente el `First` de cualquier forma oracional.

```
FirstSet CalculateFirst(Symbol[] p, Firsts firsts) {
    FirstSet result = new FirstSet();
    bool allEpsilon = true;

    foreach(var s in p) {
```

```
            result.AddAll(Firsts[s]);

            if (!Firsts[s].Contains(epsilon)) {
                allEpsilon = false;
                break;
            }
        }

        if (allEpsilon) {
            result.Add(epsilon);
        }

        return result;
    }
```

Básicamente este nuevo algoritmo consiste en repetir la parte más interna del algoritmo anterior, así que no son necesarias más explicaciones.

Pasemos entonces a calcular el conjunto `Follow` de cada no-terminal. Para ello, veamos también algunos hechos que nos ayudarán a entender como está formado este conjunto:

- `$` pertenece al `Follow(S)`.
- Por definición `epsilon` nunca pertenece al `Follow(X)` para todo `X`.
- Si `X -> WAZ` siendo `W` y `Z` formas oracionales, y `A` un no-terminal cualquiera, entonces `First(Z) - { epsilon } \subseteq Follow(A)`.
- Si `X -> WAZ` y `Z -*-> epsilon` (o igualmente `epsilon` está en el `First(Z)`), entonces `Follow(X) \subseteq Follow(A)`.

De la misma forma que en el caso del conjunto `First`, dado que las relaciones entre los `Follow` de cualquier par de no-terminales pueden ser recursivas, diseñaremos un algoritmo que los computa a todos a la misma vez:

```
Follows CalculateFollows(Grammar G, Firsts firsts) {
    var Follows = new Follows();

    Follows[S] = new FollowSet() { Token.EOF };

    foreach(var X in G.NonTerminals) {
        Follows[X] = new FollowSet();
    }

    bool changed;

    do {
        changed = false;

        foreach(var p in G.Productions) {
            // X -> W
            var X = p.First;
```

```
            var W = p.Last;

            for(int i=0; i < W.Length; i++) {
                var S = W[i];

                if (S.IsTerminal)
                    continue;

                var first = CalculateFirst(W.Sufix(i+1));
                changed = Follows[S].AddAll(first.Remove(epsilon));

                if (first.Contains(epsilon) || i == W.Length - 1) {
                    changed = Follows[S].AddAll(Follows[X]);
                }
            }
        }

    } while (changed);
}
```

Una vez tenemos todos los conjuntos `First` y `Follow` calculados, podemos decir formalmente en qué consiste una gramática LL(1). Para ello, construiremos una tabla `T`, donde asociaremos a cada par no-terminal `X` / token `t` una producción (a lo sumo). Dicha producción es la única que tiene sentido aplicar si se debe expandir el no-terminal `X` y el token actual es `t`.

Las reglas generales para generar esta tabla son las siguientes:

1. Si `X -> W` y `t` pertenece al `First(W)` entonces `T[X,t] = X -> W`.
2. Si `X -> epsilon` y `t` pertenece al `Follow(X)` entonces `T[X,t] = X -> epsilon`.

Si al aplicar estas reglas, en cada posición `T[X,t]` obtenemos a lo sumo una producción, entonces decimos que una gramática es LL(1). En caso contrario, tenemos al menos un conflicto, pues hay más de una producción que tiene sentido utilizar en algún caso. Formalmente:

> **Definición:** Sea $G =< S, N, T, P >$ una gramática libre del contexto. G es LL(1) si y solo si para todo no-terminal $X \in N$, tal que $X \leftarrow W_1|W_2||W_n$ se cumple que:
>
> - $First(Wi) \bigcap First(Wj) = \emptyset \ \forall i \neq j$
> - $\epsilon \in First(Wi) => First(Wj) \bigcap Follow(X) = \emptyset \ \forall j \neq i$

Esta definición nos garantiza que en toda entrada de la tabla LL(1) exista a lo sumo una producción a aplicar. Ahora podemos demostrar que la gramática anterior para expresiones, una vez factorizada, es LL(1):

```
E -> T X
X -> + E | epsilon
T -> int Y | ( E )
Y -> * T | epsilon
```

Comencemos por calcular todos los conjuntos `First`. Para los terminales es trivial:

```
First( int ) = { int }
First( + )   = { + }
First( * )   = { * }
First( ( )   = { ( }
First( ) )   = { ) }
```

Ahora calculemos los `First` de cada no-terminal:

```
First(E) = { (, int }
First(X) = { +, epsilon }
First(T) = { (, int }
First(Y) = { *, epsilon }
```

Luego podemos calcular los `First` de cada parte derecha de cada producción:

```
First( T X ) = { (, int }
First( + E ) = { + }
First(int Y) = { int }
First( (E) ) = { ( }
First( * T ) = { * }
```

Calculemos finalmente los `Follow` de cada no-terminal:

```
Follow(E) = { $, ) }
Follow(X) = { $, ) }
Follow(T) = { +, $, ) }
Follow(Y) = { +, $, ) }
```

Ahora podemos llenar la tabla LL(1). Comencemos por la fila correspondiente a `E`. Para ello analizamos la producción `E -> T X`. Por la regla (1) podemos decir que esta producción se aplica solo para los terminales `(` y `int`.

	int	+	*	(	)	$
E	T X			T X		

Veamos entonces la fila asociada a `T`. La producción `T -> int Y` solamente se aplica para el token `int` mientas que la producción `T -> (E)` se aplica solamente para `(`.

	int	+	*	(	)	$
T	int Y			(E)		

Ahora veamos las producciones de `X`. Para `X -> + E` la única entrada importante es con el token `+`. Por otro lado, la producción `X -> epsilon` se aplica en todos los tokens que pertenezcan al `Follow(X)`, es decir, `$` y `)`.

	int	+	*	(	)	$
X		+ E			ϵ	ϵ

Finalmente para el no-terminal `Y`, la producción `Y -> * T` es trivial, y la producción `Y -> \epilon` se aplica para `+`, `$` y `)`.

	int	+	*	(	)	$
Y		ϵ	* T		ϵ	ϵ

Finalmente, nos queda la tabla completa. Dado que no encontrarmos conflictos al construirla, podemos concluir que la gramática es LL(1):

	int	+	*	(	)	$
E	T X			T X		
T	int Y			(E)		
X		+ E			ϵ	ϵ
Y		ϵ	* T		ϵ	ϵ

Parsing Descendente No Recursivo

Una vez obtenida la tabla LL(1) podemos escribir un algoritmo de parsing descendente no recursivo. La idea general consiste en emplear una pila de símbolos, donde iremos construyendo la forma oracional que eventualmente derivará en la cadena a reconocer. Si leemos la pila desde el tope hasta el fondo, en todo momento tendremos una forma oracional que debe generar la parte de la cadena no reconocida.

El símbolo en el tope de la pila representa el terminal o no-terminal a analizar. En caso de ser un terminal, debe coincidir con el token analizado. En caso de ser un no-terminal, se consulta la tabla LL(1) y se ejecuta la producción correspondiente, insertando en la pila (en orden inverso) la forma oracional en que deriva el no-terminal extraído:

```
bool NonRecursiveParse(Grammar G, Token[] tokens) {
    Stack<Symbol> stack = new Stack<Symbol>() { G.Start };
    int nextToken = 0;
    LLTable table = BuildLLTable(G);

    while (stack.Count > 0 && nextToken < tokens.Length) {
        var symbol = stack.Pop();

        if (symbol.IsTerminal && tokens[nextToken++] != symbol.Value) {
            return false;
        }
```

```
        else if (symbol.IsNonTerminal) {
            var prod = LLTable[symbol, tokens[nextToken]];

            if (prod == null)
                return false;

            for(var s in prod.Reverse()) {
                stack.Push(s);
            }
        }
    }

    return stack.Count == 0 && nextToken == tokens.Length;
}
```

En la práctica la mayoría de las gramáticas interesantes no son LL(1). Sin embargo, con suficiente esfuerzo pueden lograrse gramáticas LL(k) para un valor k > 1, que también pueden ser parseadas con técnicas similares. Por otro lado, para gramáticas suficientemente simples (como las expresiones aritméticas) este parser es muy eficiente, y fácil de implementar.

Finalmente, en el caso que la gramática no sea LL(1), este análisis nos permite reducir al mínimo necesario la cantidad de producciones a probar en cada terminal. La tabla LL(1) en estos casos pudiera tener más de una producción en cada entrada, y en esos casos implementaríamos un parser recursivo que solamente probara aquellas producciones listadas en la tabla. De esta forma, podemos obtener el parser (descendiente) más eficiente posible, sin perder en expresividad.

De todas formas las gramáticas LL(1) son un conjunto estrictamente menor que las gramáticas libres del contexto. Más adelante veremos estrategias de parsing basadas en principios similares que permiten reconocer lenguajes y gramáticas más expresivas.

Capítulo 3

Parsing Ascendente (*Bottom-Up*)

En la sección anterior vimos las técnicas de parsing descendente, y aprendimos algunas de las limitaciones más importantes que tienen. Dado que el objetivo es poder predecir exactamente que producción es necesario ejecutar en cada momento, las gramáticas LL(1) tienen fuertes restricciones. En particular, deben estar factorizadas, y no pueden tener recursión izquierda (criterios que son necesarios pero no suficientes). Por este motivo, para convertir una gramática "natural" en una gramática LL(1) es necesario adicionar no-terminales para factorizar y eliminar la recursión, que luego no tienen ningún significado semántico. Los árboles de derivación de estas gramáticas son por tanto más complejos, y tienen menos relación con el árbol de sintaxis abstracta que queremos obtener finalmente (aunque aún no hemos definido este concepto formalmente).

Intuitivamente, el problema con los parsers descendentes, es que son demasiado exigentes. En cada momento, se quiere saber qué producción hay que aplicar para obtener la porción de cadena que sigue. En otras palabras, a partir de una forma oracional, tenemos que decidir cómo expandir el no-terminal más a la izquierda, de modo que la siguiente forma oracional esté "más cerca" de generar la cadena. Por este motivo se llama parsing descendente.

¿Qué pasa si pensamos el problema de forma inversa? Comenzamos con la cadena completa, y vamos a intentar reducir fragmentos de la cadena, aplicando producciones "a la inversa" hasta lograr reducir toda la cadena a `S`. En vez de intentar adivinar que producción aplicar "de ahora en adelante", intentaremos deducir, dado un prefijo de la cadena analizado, qué producción se puede "desaplicar" para reducir ese prefijo a una forma oracional que esté "más cerca" del símbolo inicial. Si pensamos el problema de forma inversa, puede que encontremos una estrategia de parsing que sea más permisiva con las gramáticas.

Veamos un ejemplo. Recordemos la gramática "natural" no ambigua para expresiones aritméticas:

```
E = T + E | T
T = int * T | int | ( E )
```

Y la cadena de siempre: `int * ( int + int )`. Tratemos ahora de construir una derivación

"de abajo hacia arriba", tratando de reducir esta cadena al símbolo inicial `E` aplicando producciones a la inversa. Vamos a representar con una barra vertical `|` el punto que divide el fragmento de cadena que hemos analizado del resto. De modo que empezamos por:

```
|int * ( int + int )
```

Miramos entonces el primer token:

```
 int|* ( int + int )
```

El primer token de la cadena es `int`, que se puede reducir aplicando `T -> int` a la inversa. Sin embargo, esta reducción no es conveniente. ¿Por qué? El problema es que queremos lograr reducir hasta `E`, por tanto hay que tener un poco de "luz larga" y aplicar reducciones que, en principio, dejen la posibilidad de seguir reduciendo hasta `E`. Como no existe ninguna producción que derive en `T * w`, si reducimos `T -> int` ahora no podremos seguir reduciendo en el futuro. Más adelante formalizaremos esta idea. Seguimos entonces buscando hacia la derecha en la cadena:

```
 int *|( int + int )
 int * (|int + int )
 int * ( int|+ int )
```

En este punto podemos ver que sí es conveniente reducir `T -> int`, porque luego viene un `+` y tenemos, en principio, la posibilidad de seguir reduciendo aplicando `E -> T + E` en el futuro:

```
 int * ( T|+ int )
```

Avanzamos hacia el siguiente token reducible:

```
 int * ( T +|int )
 int * ( T + int|)
```

Aquí nuevamente podemos aplicar la reducción `T -> int`:

```
 int * ( T + T|)
```

Antes de continuar, dado que tenemos justo delante de la barra (`|`) un sufijo `T + T`, deberíamos darnos cuenta que es conveniente reducir `E -> T` para luego poder reducir `E -> T + E`:

```
 int * ( T + E|)
 int * ( E|)
```

En este punto, no hay reducciones evidentes que realizar, así que seguimos avanzando:

```
 int * ( E )|
```

Hemos encontrado entonces un sufijo `( E )` que podemos reducir con `T -> (E)`:

```
 int * T|
```

Luego reducimos `T -> int * T`:

```
 T|
```

Y finalmente reducimos `E -> T`:

```
E
```

En este punto hemos logrado reducir al símbolo inicial toda la cadena. Veamos la secuencia de formas oracionales que hemos obtenido:

```
int * ( int + int )
int * ( T + int )
int * ( T + T )
int * ( T + E )
int * ( E )
int * T
T
E
```

Si observamos esta secuencia en orden inverso, veremos que es una derivación extrema derecha de `E -*-> int * ( int + int )`. Justamente, un parser ascendente se caracteriza porque construye una derivación extrema derecha en orden inverso, desde la cadena hacia el símbolo inicial. Tratemos ahora de formalizar este proceso. Para ello, notemos primero algunas propiedades interesante que cumple todo parser ascendente. Partimos del hecho que hemos dado como definición de un parser bottom-up:

> Un parser bottom-up construye una derivación extrema derecha de $S \to^* \omega$.

A partir de este hecho, que hemos dado como definición, podemos deducir una consecuencia muy interesante:

> Sea $\alpha\beta\omega$ una forma oracional en un paso intermedio de un parser ascendente. Sea $X \to \beta$ la siguiente reducción a realizar. Entonces ω es una cadena de terminales (formalmente $\omega \in T^*$).

Para ver por qué esto es cierto, basta notar que si la derivación que construiremos es extrema derecha, la aplicación de $X \to \beta$ en este paso solamente puede ocurrir si X es el no-terminal más a la derecha. O sea, si $\alpha\beta\omega$ es el paso correspondiente, y reducimos por $X \to \beta$, entonces el siguiente paso es la forma oracional $\alpha X\omega$, donde X es el no-terminal más a la derecha, debido justamente a que estamos construyendo una derivación extrema derecha.

Esta propiedad nos permite entonces entender que en todo paso de un parser ascendente, cada vez que sea conveniente reducir $X \to \beta$, es porque existe una posición (que hemos marcado con `|`), tal que $\alpha\beta|\omega$ es la forma oracional, donde $\alpha\beta \in \{N \cup T\}^*$ y $\omega \in T^*$. Tenemos entonces dos tipos de operaciones que podemos realizar, que llamaremos **shift** y **reduce**. La operación **shift** nos permite mover la barra `|` un token hacia la derecha, lo que equivale a decir que analizamos el siguiente token. La operación **reduce** nos permite coger un sufijo de la forma oracional que está antes de la barra `|` y reducirla a un no-terminal, aplicando una producción a la inversa (o "desaplicando" la producción). Veamos nuevamente la secuencia de operaciones que hemos realizado, notando las que fueron **shift** y las que fueron **reduce**:

```
|int * ( int + int )         | shift
 int|* ( int + int )         | shift
 int *|( int + int )         | shift
```

```
int * (|int + int )          | shift
int * ( int|+ int )          | reduce T -> int
int * ( T|+ int )            | shift
int * ( T +|int )            | shift
int * ( T + int|)            | reduce T -> int
int * ( T + T|)              | reduce E -> T
int * ( T + E|)              | reduce E -> T + E
int * ( E|)                  | shift
int * ( E )|                 | reduce T -> ( E )
int * T|                     | reduce T -> int * T
T|                           | reduce E -> T
E                            | OK
```

Debido a estas operaciones, llamaremos a este tipo de mecanismos *parsers shift-reduce*. Veamos de forma general como implementar este tipo de parsers.

Parsers *Shift-Reduce*

Notemos que la parte a la izquierda de la barra siempre cambia porque un sufijo es parte derecha de una producción, y se reduce a un no-terminal. La parte derecha solo cambia cuando un terminal "cruza" la barra y se convierte en parte del sufijo que será reducido en el futuro. De forma que la barra que la parte izquierda se comporta como una pila, ya que solamente se introducen terminales por un extremo, y se extraen símbolos (terminales o no-terminales) por el mismo extremo. La parte derecha es simplemente una secuencia de tokens, que se introducen en la pila uno a uno. Formalicemos entonces el funcionamiento de un parser *shift-reduce*.

Un parser *shift-reduce* es un mecanismo de parsing que cuenta con las siguientes estructuras:

- Una pila de símbolos `S`.
- Una secuencia de terminales `T`.

Y las operaciones siguientes:

- **shift**: Si $S = \alpha|$ es el contenido de la pila, y $T = c\omega\$$ la secuencia de terminales, entonces tras aplicar una operación **shift** se tiene en la pila $S' = \alpha c|$, y la secuencia de terminales ahora es $T' = \omega\$$. Es decir, se mete en la pila el token c.
- **reduce**: Si $S = \alpha\beta|$ el contenido de la pila, y $X \to \beta$ es una producción, entonces tras aplicar una operación **reduce** $T \to \beta$ se tiene en la pila $S' = \alpha X|$. La secuencia de terminales no se modifica. Es decir, se extraen de la pila $|\beta|$ símbolos y se introduce el símbolo X correspondiente.

Podemos definir entonces el proceso de parsing como:

> Sea $S = \emptyset$ la pila inicial, $T = \omega\$$ la cadena a reconocer, y E el símbolo inicial, un parser shift-reduce reconoce esta cadena si y solo si existe una secuencia de operaciones **shift** y **reduce** tal que tras aplicarlas se obtiene $S = E$ y $T = \$$.

Es decir, un parser shitf-reduce básicamente tiene que aplicar operaciones *convenientemente*

hasta que en la pila solamente quede el símbolo inicial, y se hayan consumido todos los tokens de la cadena de entrada. En este punto, se ha logrado construir una derivación extrema derecha de la cadena correspondiente. Por supuesto, existe un grado importante de no determinismo en esta definición, porque en principio puede haber muchas secuencias de operaciones shift-reduce que permitan llegar al símbolo inicial. Si asumimos que la gramática no es ambigua, y por tanto solo existe una derivación extrema derecha, podemos intuir que debe ser posible construir un parser que encuentre la secuencia de shift-reduce que produce esa derivación. Desgraciadamente esto no es posible para todo tipo de gramáticas libre del contexto, pero existen gramáticas más restringidas para las que sí es posible decidir de forma determinista en todo momento si la operación correcta es **shift** o **reduce**, y en el segundo caso a qué símbolo reducir.

Para simplificar la notación, en ocasiones identificaremos el estado de un parser shift-reduce en la forma $\alpha|\omega$, sobreentendiendo que el estado de la pila es $S = \alpha|$ y la cadena de entrada es $\omega\$$. Diremos además que un estado $\alpha|\omega$ es válido, si y solo si la cadena pertenece al lenguaje, y este estado forma parte de los estados necesarios para completar el parsing de forma correcta.

Este tipo de parsers son en la práctica los más usados, pues permiten reconocer una cadena (y construir la derivación) con un costo lineal en la longitud de la cadena (la misma eficiencia que los parsers LL), y permiten parsear gramáticas mucho más poderosas y expresivas que las gramáticas LL. De hecho, la mayoría de los compiladores modernos usan alguna variante de un parser shift-reduce. La diferencia entre todos ellos radica justamente en cómo se decide en cada paso qué operación aplicar. Formalicemos entonces el problema de decisión planteado. Tomemos de nuevo la gramática anterior, y recordemos que en el paso:

```
int|* ( int + int )
```

Habíamos dicho que aunque era posible reducir `T -> int`, no era convieniente hacerlo, porque caeríamos en una forma oracional que no puede ser reducida a `E`. En particular, en este caso caeríamos en:

```
T|* ( int + int )
```

Y sabemos intuitivamente que esta forma oracional no es reducible a `E`, porque no existe ninguna producción que comience por `T *`, o dicho de otra forma, `*` no pertenece al `Follow(T)`. Tratemos de formalizar entonces este concepto de "momento donde es conveniente reducir". Para ello introduciremos una definición que formaliza esta intuición.

> Sea $S \to^* \alpha X \omega \to \alpha\beta\omega$ una derivación extrema derecha de la forma oracional $\alpha\beta\omega$, y $X \to \beta$ una producción, decimos que $\alpha\beta$ es un **handle** de $\alpha\beta\omega$.

Intuitivamente, un **handle** nos representa un estado en la pila donde es conveniente reducir, porque sabemos que existen reducciones futuras que nos permiten llegar al símbolo inicial. En la definición anterior la pila sería justamente $\alpha\beta|$, y la cadena de terminales sería $\omega\$$. Sabemos que es posible seguir reduciendo, justamente porque hemos definido un **handle** a partir de conocer que existe una derivación extrema derecha donde aparece ese prefijo. De modo que justamente lo que queremos es identificar cuando tenemos un **handle** en la pila, y en ese momento sabemos que es conveniente reducir.

El problema que nos queda es que hemos definido el concepto de **handle** pero no tenemos

una forma evidente de reconocerlos. Resulta que, desgraciadamente no se conoce ningún algoritmo para identificar un **handle** unívocamente en cualquier gramática libre del contexto. Sin embargo, existen algunas heurísticas que nos permiten reconocer algunos **handle** en ciertas ocasiones, y afortunadamente existen gramáticas donde estas heurísticas son suficientes para reconocer todos los **handle** de forma determinista. En última instancia, la diferencia real entre todos los parsers shitf-reduce radica en la estrategia que usen para reconocer los **handle**. Comenzaremos por la más simple.

Reconociendo Handles

La forma en la que hemos definido el concepto de **handle** nos permite demostrar una propiedad interesante:

> En un parser shift-reduce, los **handles** aparecen solo en el tope de la pila, nunca en su interior.

Podemos esbozar una idea de demostración a partir de una inducción fuerte en la cantidad de operaciones **reduce** realizadas. Al inicio, la pila está vacía, y por tanto la hipótesis es trivialmente cierta. Tomemos entonces un estado intermedio de la pila $\alpha\beta|$ que es un **handle**. Además, es el único **handle** por hipótesis de inducción fuerte, ya que de lo contrario tendríamos un **handle** en el interior de la pila. Al reducir, el no-terminal más a la derecha queda en el tope de la pila, ya que es una derivación extrema derecha. Por tanto tendremos un nuevo estado en la pila $\alpha X|$. Ahora pueden suceder 2 cosas, o bien este estado es un **handle** también (y se cumple la hipótesis), o en caso contrario el siguiente **handle** aparecerá tras alguna secuencia solamente de operaciones **shift**. Este nuevo **handle** tiene que aparecer también en el tope de la pila, pues si apareciera en el interior de la pila, tendría que haber estado antes de X (lo que es falso por hipótesis de inducción), o tendría que haber aparecido antes del último terminal al que se le hizo **shift**, pero en tal caso deberíamos haber hecho **reduce** en ese **handle**, lo que contradice el hecho de que solo han sucedido operaciones **shift** desde el último **reduce**.

Este teorema nos permite, en primer lugar, formalizar la intuición de que solamente hacen falta movimientos **shift** a la izquierda. Es decir, una vez un terminal ha entrado en la pila, o bien será reducido en algún momento, o bien la cadena es inválida, pero nunca hará falta sacarlo de la pila y volverlo a colocar en la cadena de entrada.

Por otro lado, este teorema nos describe la estructura de la pila, lo que será fundamental para desarrollar un algoritmo de reconocimiento de **handles**. Dado que los **handles** siempre aparecen en el tope de la pila, en todo momento tendremos, en principio, un prefijo de un **handle**. De modo que una idea útil para reconocer **handles** es intentar reconocer cuales son los prefijos de un **handle**. En general, llamaremos *prefijo viable* a toda forma oracional α que puede aparecer en la pila durante un reconocimiento válido de una cadena del lenguaje. Formalmente:

> Sea $\alpha|\omega$ un estado válido de un parser shift-reduce durante el reconocimiento de una cadena, entonces decimos que α es un prefijo viable.

Intuitivamente, un prefijo viable es un estado en el cual todavía no se ha identificado un

error de parsing, por lo que, hasta donde se sabe, la cadena todavía pudiera ser reducida al símbolo inicial. Si podemos reconocer el lenguaje de todos los prefijos viables, en principio siempre sabremos si la pila actual representa un estado válido. Además podemos intuir que esto nos debería ayudar a decidir si hacer un **shift** o un **reduce**, según cual de las dos operaciones nos mantenga el contenido de la pila siendo un prefijo viable. De modo que hemos reducido el problema de identificar **handles** (de forma aproximada) al problema de identificar prefijos viables.

Si analizamos todos los posibles estados válidos de la pila (los posibles prefijos viables), notaremos una propiedad interesante que nos ayudará a reconocer estos prefijos. Supongamos que tenemos un estado $\alpha\beta|\omega$ que es un **handle** para $X \to \beta$. Entonces por definición $\alpha\beta$ es también un prefijo viable. Además, una vez aplicada la reducción, tendremos el estado $\alpha X|\omega$. Por tanto αX también es un prefijo viable, porque de lo contrario esta reducción sería inválida, contradiciendo el hecho de que hemos reducido correctamente en un **handle**. Por tanto o bien αX es un **handle** en sí, o es un prefijo de un **handle**. En el segundo caso, entonces hay una producción $Y \to \theta X\phi$, tal que $\alpha = \delta\theta$. Es decir, hay un sufjo de αX que tiene que ser prefijo de la parte derecha de esa producción.

¿Por qué?, pues porque como hemos reducido en un **handle**, esto quiere decir que sabemos que es posible en principio seguir reduciendo, por tanto tiene que haber alguna secuencia de tokens ϕ, que pudiera o no venir en ω (aún no sabemos), que complete la parte derecha $\theta X\phi$. Es decir, como sabemos que potencialmente podríamos seguir reduciendo, entonces lo que tenemos en la pila ahora tiene que ser prefijo de la parte derecha de alguna producción. Si no lo fuera, ya en este punto podríamos decir que será imposible seguir reduciendo en el futuro, puesto que solamente introduciremos nuevos tokens en la pila, y nunca tocaremos el interior de la pila (excepto a través de reducciones, que siempre modifican el tope de la pila).

Esta intuición nos dice algo muy importante sobre el contenido de la pila:

> En todo estado válido $\alpha|\omega$ de un parser shift-reduce, la forma oracional α es una secuencia $\alpha = \beta_1\beta_2 \dots \beta_n$ donde para cada β_i se cumple que existe una producción $X \to \beta_i\theta$.

Es decir, todo estado válido de la pila es una concatenación de prefijos de partes derechas de alguna producción. En caso contrario, tendríamos una subcadena en la pila que no forma parte de ninguna producción, por tanto no importa lo que pase en el futuro, esta subcadena nunca sería parte de un **reduce**, y por tanto la cadena a reconocer tiene que ser inválida. Más aún, podemos decir exactamente de cuales producciones tienen que ser prefijo esas subcadenas. Dado que en última instancia tenemos que reducir al símbolo inicial S, entonces en la pila tenemos necesariamente que encontrar prefijos de todas las producciones que participan en la derivación extrema derecha que estamos construyendo. Formalmente:

> Sea $S \to^* \alpha\delta \to^* \omega$ la única derivación extrema derecha de ω, sea $\alpha|\delta$ un estado de un parser shift-reduce que construye esta derivación, sea $X_1 \to \theta_1, \dots, X_n \to \theta_n$ la secuencia de producciones a aplicar tal que $S \to^* \alpha\delta$, entonces $\alpha = \beta_1 \dots \beta_n$, donde β_i es prefijo de θ_i.

Es decir, en todo momento en la pila lo que tenemos es una concatenación de prefijos de todas las producciones que quedan por reducir. Notemos intuitivamente que esto debe ser

cierto, porque el parser va a construir esta derivación al revés. Por tanto en el estado $\alpha|\delta$, que corresponde a la forma oracional $\alpha\delta$ en la derivación, el parser ya ha reconstruido todas las producciones finales, que hacen que $\alpha\delta \rightarrow^* \omega$ (de atrás hacia adelante), y le falta por reconstruir las producciones que hacen que $S \rightarrow^* \alpha\delta$. Luego, lo que está en la pila tiene que reducirse a S, y como solo puede pasar que se metan nuevos terminales de δ, todo lo que está en α tiene que de algún modo poderse encontrar en alguna de las producciones que faltan por reducir. De lo contrario, esta reducción sería imposible.

Por supuesto, muchos de los prefijos β_i pueden ser ϵ, porque todavía no han aparecido ninguno de los símbolos que forman la producción en la pila (dependen de que reducciones siguientes introduzcan un no-terminal, o de que **shifts** siguientes introduzcan un terminal). De esta forma podemos entender que incluso la pila vacía es una concatenación de prefijos de producciones, todos ϵ. Lo que no puede pasar es que tengamos una subcadena β_k que no forme parte de ningún prefijo de ninguna producción, porque entonces nunca podremos reducir totalmente al símbolo inicial. De modo que un prefijo viable no es nada más que una concatenación de prefijos de las producciones que participan en la derivación extrema derecha que queremos construir.

Para ver un ejemplo tomemos nuevamente nuestra gramática favorita:

```
E -> T + E | T
T -> int * T | int | (E)
```

Y veamos la cadena `( int )`. La derivación extrema derecha que nos genera esta cadena es:

```
E -> T -> ( E ) -> ( T ) -> ( int )
```

Para esta cadena `( E | )` es un estado válido, pues en el siguiente **shift** aparecerá el token `)` que permite reducir (en dos pasos `T -> ( E )` y `E -> T`) al símbolo inicial. Por tanto, `( E` es un prefijo viable. Veamos cómo este prefijo es una concatenación de prefijos de las dos producciones que faltan por reducir. Evidentemente `( E` es prefijo de `T -> ( E )`, y además, ϵ es prefijo de `E -> T`.

Esta idea es la pieza fundamental que nos permitirá deducir un algoritmo para reconocer prefijos viables. Como un prefijo viable no es más que una concatenación de prefijos de partes derechas de producciones, simplemente tenemos que reconocer *el lenguaje de todas las posibles concatenaciones de prefijos de partes derechas de producciones, que pudieran potencialmente aparecer en una derivación extrema derecha*. Parece una definición complicada, pero dado que conocemos la gramática que queremos reconocer, es de hecho bastante fácil. Notemos que hemos dicho, *que pudieran aparecer en una derivación*, lo cual nos debe dar una idea de cómo construir estas cadenas. Simplemente empezaremos en el símbolo inicial S, y veremos todas las posibles maneras de derivar, e iremos rastreando los prefijos que se forman. Para esto nos auxiliaremos de un resultado teórico impresionante, que fundamenta toda esta teoría de parsing bottom-up:

> El lenguaje de todos los prefijos viables de una gramática libre del contexto es regular.

Aunque parece un resultado caído del cielo, de momento podemos comentar lo siguiente. En principio, el lenguaje de todos los posibles prefijos de cada producción es regular (es finito). Y la concatenación de lenguajes regulares es regular. Por tanto, de alguna forma

podríamos intuir que este lenguaje de todas las posibles concatenaciones de prefijos debería ser regular. Por tanto debería ser posible construir un autómata finito determinista que lo reconozca. Claro, queda la parte de que no son *todas* las concatenaciones posibles, sino solo aquellas que aparecen en alguna derivación extrema derecha. Tratemos entonces de construir dicho autómata, y a la vez estaremos con esto demostrando que efectivamente este lenguaje es regular. Recordemos que en última instancia lo que queremos es un autómata que lea el contenido de la pila, y nos diga si es un prefijo viable o no.

Para entender como luce este autómata, introduciremos primero un concepto nuevo. Llamaremos **item** a una cadena de la forma $X \to \alpha\beta$. Es decir, simplemente tomamos una producción y le ponemos un punto (`.`) en cualquier lugar en su parte derecha. Este **item** formaliza la idea de ver los posibles prefijos de todas las producciones. Por cada producción $X \to \delta$, tenemos $|\delta| + 1$ posibles **items**. Por ejemplo, en la gramática anterior, tenemos los siguientes **items**:

```
E -> .T + E
E ->  T.+ E
E ->  T +.E
E ->  T + E.

E -> .T
E ->  T.

T -> .int * T
T ->  int.* T
T ->  int *.T
T ->  int * T.

T -> .int
T ->  int.

T -> .( E )
T ->  (.E )
T ->  ( E.)
T ->  ( E ).
```

Cada uno de estos **items** nos representa un posible prefijo de una producción. Pero además, cada **item** nos permite también rastrear que esperamos ver a continuación de dicha producción, si es que realmente esa producción fuera la que tocara aplicar a continuación. Veamos entonces qué podemos decir de cómo estos **items** se relacionan entre sí. Tomemos por ejemplo el **item** `E -> T.+ E`. Este **item** nos dice que ya hemos visto algo en la cadena que se reconoce como un `T`, y que esperamos ver a continuación un `+`, si resulta que esta es la producción que realmente tocaba aplicar. El **item** `E -> .T + E` nos dice que si realmente esta es la producción correcta, entonces lo que viene en la cadena debería ser reconocible como un `T`, y luego debería vernir un `+`, y luego algo que se reconozca como un `E`. Por último, un **item** como `T -> ( E ).` nos dice que ya hemos visto toda la parte derecha de esta producción, y por tanto intuitivamente deberíamos poder reducir. De modo que estos **items** nos están diciendo además si es conviente hacer **shift** o hacer **reduce**.

Vamos a utilizar ahora estos **items** para construir el autómata finito no determinista que nos dirá que es lo que puede venir el pila. Cada estado de este autómata es uno de los **items**. Vamos a decir que el estado asociado asociado al **item** $X \to \alpha\beta$ representa que en el tope de la pila tenemos el prefijo α de esta producción, o en general, algo que es generado por este prefijo α. Por tanto, todos los estados son estados finales, puesto que cada estado corresponde a un prefijo de alguna producción. Lo que tenemos que hacer es definir entonces un conjunto de transiciones que solamente reconozcan aquellas secuencias de prefijos que consituyen prefijos viables.

Suponamos ahora que tenemos cierto estado de un parser shift-reduce, y queremos saber si es un estado válido. Hagamos a nuestro autómata leer esta la pila desde el fondo hacia el tope, como si fuera una cadena de símbolos. Supongamos entonces que durante esta lectura nos encontramos en cierto estado del autómata, asociado por ejemplo al **item** `E -> .T`, y hemos leído ya una parte del fondo de la pila, siguiendo las transiciones que aún no hemos definido del todo. La pregunta entonces es qué puede venir a continuación en la pila, justo encima del último símbolo que analizamos. Evidentemente, en la pila podría venir un no-terminal `T` directamente, que haya aparecido por alguna reducción hecha anteriormente. Si este fuera el caso, entonces todavía tendríamos un prefijo viable. Entonces podemos añadir una transición del estado `E -> .T` al estado `E -> T.`.

Por otro lado, incluso si no viniera directamente un `T`, de todas formas todavía es posible que tengamos un prefijo viable. ¿Cómo? Supongamos que el no-terminal `T` todavía no ha aparecido porque esa reducción aún no ha ocurrido. Entonces lo que debería venir a continuación en la pila es algo que sea prefijo de alguna producción de `T`, de modo que un **reduce** futuro nos ponga ese `T` en la pila. En ese caso, todavía estaríamos en un prefijo viable, porque tendríamos un prefijo de `E -> T`, y luego un prefijo de algo que se genera con `T`. ¿Cómo reconocer entonces cualquier prefijo de cualquier producción que sale de `T`? Pues afortunadamente tenemos estados que hacen justamente eso, dígase `T -> .int` y `T -> .int * T`, es decir, los **items** iniciales de las producciones de `T`. Dado que estamos construyendo un autómata no-determinista, tenemos la libertad de añadir transiciones ϵ a estos dos estados. De modo que el estado `E -> .T` tiene tres transiciones, con un `T` se mueve a `E -> T.`, y con ϵ se mueve a `T -> .int` y a `T -> .int * T`.

Por otro lado, si estuviéramos en el estado `E -> T.+ E`, lo único que podemos esperar que venga en la pila es un terminal `+`. En cualquier otro ya no tendríamos un prefijo viable, pues estábamos esperando tener un prefijo de `E -> T + E`, y ya hemos visto en la pila un `T`. Por tanto si fuera cierto que esta pila es un prefijo viable, tendría que venir algo que continuara este prefijo o empezara un nuevo prefijo. Pero dado que en la producción que estamos esperando lo que viene es un terminal, no existe forma de un **reduce** futuro nos ponga en esa posición a dicho terminal (los **reduce** siempre introducen un no-terminal en la pila). Luego, si no viene exactamente un `+` en la pila, ya podemos estar seguros que este prefijo no es viable (claro, como estamos en un autómata no-determinista puede que existan otros caminos donde sí se reconoce un prefijo viable).

De forma general tenemos las siguientes reglas:

- Si tenemos un estado $X \to \alpha c \beta$ donde c es un terminal, añadimos una transición con c al estado $X \to \alpha c \beta$.

- Si tenemos un estado $X \to \alpha Y \beta$ donde Y es un no-terminal, añadimos una transición con Y al estado $X \to \alpha Y \beta$, y además por cada producción $Y \to \delta$ añadimos una transición con ϵ al estado $Y \to \delta$.

Apliquemos entonces estas reglas al conjunto completo de **items** que hemos obtenido anteriormente. Primero definiremos un estado por cada **item**, y luego iremos adicionando las transiciones:

```
[ 1]  E -> .T + E     {  }
[ 2]  E ->  T.+ E     {  }
[ 3]  E ->  T +.E     {  }
[ 4]  E ->  T + E.    {  }
[ 5]  E -> .T         {  }
[ 6]  E ->  T.        {  }
[ 7]  T -> .int * T   {  }
[ 8]  T ->  int.* T   {  }
[ 9]  T ->  int *.T   {  }
[10]  T ->  int * T.  {  }
[11]  T -> .int       {  }
[12]  T ->  int.      {  }
[13]  T -> .( E )     {  }
[14]  T ->  (.E )     {  }
[15]  T ->  ( E.)     {  }
[16]  T ->  ( E ).    {  }
```

Tomemos entonces el estado `E -> .T + E`. Primero ponemos la transición con `T` hacia `E -> T.+ E`:

```
[ 1]  E -> .T + E     { T:2 }
```

Y luego, dado que `T` es un no-terminal, adicionamos las transiciones ϵ correspondientes a los estados `T -> .int` y `T -> .int * T`:

```
[ 1]  E -> .T + E     { T:2, e:7, e:11 }
```

Por otro lado, para `E -> T.+ E` la única transición válida es con `+`, hacia el estado `E -> T +.E`:

```
[ 2]  E ->  T.+ E     { +:3 }
```

Para el estado `E -> T+.E` igualmente tenemos una transición con `E` y dos transiciones con ϵ:

```
[ 3]  E ->  T +.E     { E:4, e:1, e:5 }
```

El estado `E -> T + E.` no tiene transiciones salientes, pues representa que se ha reconocido toda la producción. Es responsabilidad de otros estados continuar reconociendo (de forma no-determinista) los prefijos que puedan quedar en la pila.

El estado `E -> .T` se parece mucho al estado `E -> .T + E`. De hecho, tiene las mismas transiciones:

```
[ 5]  E -> .T         { T:2, e:7, e:11 }
```

Finalmente el estado `E -> T.` tampoco transiciones salientes. Ya hemos dicho que todos los estados son finales, pues como las transiciones siempre nos mueven de un prefijo viable a otro, en cualquier momento en que se acabe la pila tenemos un prefijo viable reconocido. Solo queda definir el estado inicial. En principio, deberíamos empezar de forma no-determinista por cualquiera de los estados iniciales de las producciones de `E`. Afortunadamente, en un autómata no-determinista tenemos un recurso para simular esta situación en la que queremos 2 estados iniciales. Simplemente añadimos un estado "dummy", con transiciones ϵ a cada uno de los estados iniciales que deseamos. Desde el punto de la gramática, esto es equivalente a añadir un símbolo nuevo `E'` con la única producción `E' -> E` y convertirlo en el símbolo inicial. Para esta producción tenemos dos nuevos **items**: `E' -> .E` y `E' -> E.`. El estado `E' -> .E` se convertirá en el estado inicial de nuestro autómata.

El estado `E' -> E.` también es convieniente, pues nos permite reconocer que hemos logrado reducir al símbolo inicial, y deberíamos haber terminado de consumir toda la cadena. De modo que este estado "especial" nos permitirá además saber cuando aceptar la cadena. No podemos simplemente aceptar la cadena en cualquier estado donde se reduzca a `E`, porque es posible que estuviéramos reduciendo a un `E` intermedio, por ejemplo, al `E` que luego de ser reducido en `T -> (E)`.

Ahora que hemos visto como se construyen estas transiciones, veamos directamente el autómata completo.

```
[ 0] E' -> .E         { E: 17, e:1, e:5 }
[ 1]  E -> .T + E     { T:2, e:7, e:11 }
[ 2]  E ->  T.+ E     { +:3 }
[ 3]  E ->  T +.E     { E:4, e:1, e:5 }
[ 4]  E ->  T + E.    {  }
[ 5]  E -> .T         { T:2, e:7, e:11 }
[ 6]  E ->  T.        {  }
[ 7]  T -> .int * T   { int:8 }
[ 8]  T ->  int.* T   { *:9 }
[ 9]  T ->  int *.T   { T:10, e:7, e:11 }
[10]  T ->  int * T.  {  }
[11]  T -> .int       { int:12 }
[12]  T ->  int.      {  }
[13]  T -> .( E )     { (:14 }
[14]  T ->  (.E )     { E:15, e:1, e:5 }
[15]  T ->  ( E.)     { ):16 }
[16]  T ->  ( E ).    {  }
[17] E' ->  E.
```

A estos **items** se les denomina también **items LR(0)**, que significa *left-to-right rightmost-derivation look-ahead 0*.

Autómata LR(0)

Hemos construido finalmente un autómata finito no-determinista que reconoce exactamente el lenguaje de los prefijos viables. Sabemos que existe un autómata finito determinista que reconoce exactamente el mismo lenguaje. Aplicando el algoritmo de conversión de NFA a DFA podemos obtener dicho autómata. Sin embargo, hay una forma más directa de obtener el autómata finito determinista, que consiste en construir los estados aplicando el algoritmo de conversión a medida que vamos analizando las producciones y obteniendo los **items**.

Recordemos que el algoritmo de conversión de NFA a DFA básicamente construye un estado por cada subconjunto de los estados del NFA, siguiendo primero todas las transiciones con el mismo terminal, y luego computando la ϵ-clausura del conjunto de estados resultante. Para cada uno de estos "super-estados" Q_i, la nueva transición con un terminal concreto c va hacia el super-estado que representa exactamente al conjunto clausura de todos los estados originales a los que se llegaba desde algún estado $q_j \in Q_i$.

Vamos ahora a reescribir este algoritmo, pero teniendo en cuenta directamente que los estados del NFA son **items**. Por tanto, los super-estados del DFA serán conjuntos de **items**, que son justamente la ϵ-clausura de los **items** a los que se puede llegar desde otro conjunto de **items** siguiendo un símbolo concreto X (terminal o no-terminal). Definiremos entonces dos tipos de **items** para simplicar:

- Un **item kernel** es aquel de la forma $E' \to E$ si E' es nuevo símbolo inicial, o cualquier item de la forma $X \to \alpha\beta$ con $|\alpha| > 1$.
- Un **item no kernel** es aquel de la forma $X \to \beta$ excepto el **item** $E' \to E$.

Hemos hecho estas definiciones, porque de cierta forma los **items kernel** son los realmente importantes. De hecho, podemos definir dado un conjunto de **items kernel**, el conjunto clausura, que simplemente añade todos los **items no kernel** que se derivan de este conjunto.

> Sea I un conjunto de **items** (kernel o no), el conjunto clausura de I se define como $CL(I) = I \cup \{X \to \beta\}$ tales que $Y \to \alpha X \delta \in CL(I)$.

Es decir, el conjunto clausura no es más que la formalización de la operación mediante la cuál añadimos todos los **items** no-kernel que puedan obtenerse de cualquier **item** en I. Nótese que la definición es recursiva, es decir, el conjunto clausura de I se define a partir del propio conjunto clausura de I. Para computarlo, simplemente partimos de $CL(I) = I$ y añadimos todos los items no-kernel que podamos, mientras cambie el conjunto. Por ejemplo, computemos el conjunto clausura del item asociado estado inicial $E' \to E$. Partimos del conjunto singleton que solo contiene a este item:

```
I = { E' -> .E }
```

Ahora buscamos todas las producciones de `E` y añadimos sus items iniciales:

```
I = { E' -> .E,
      E -> .T,
      E -> .T + E }
```

Ahora buscamos todas las producciones de `T` y añadimos sus items iniciales:

```
I = { E' -> .E,
      E -> .T,
      E -> .T + E,
      T -> .int,
      T -> .int * T,
      T -> .( E ) }
```

Como no hemos añadido ningún item que tenga un punto delante de un no-terminal nuevo, este es el conjunto final. Notemos que esta definición no es nada más que la definición de ϵ-clausura usada en la conversión de un NFA a un DFA, solo que la hemos definido en función de los **items** directamente. Si aplicamos la ϵ-clausura al estado q_0 de nuestro NFA definido anteriormente, llegaremos exactamente al mismo conjunto de **items**.

Una vez que tenemos este conjunto clausura de **items**, podemos definir entonces cómo añadir transiciones. Para ello definiremos la función $Goto(I, X) = J$, que nos mapea un conjunto de items a otro conjunto de items a partir de un símbolo X, de la siguiente forma:

$$Goto(I, X) = CL(\{Y \to \alpha X \beta | Y \to \alpha X \beta \in I\})$$

La función $Goto(I, X)$ simplemente busca todos los items en I donde aparece un punto delante del símbolo X, crea un nuevo conjunto donde el punto aparece detrás del símbolo X, y luego calcula la clausura de este conjunto. Básicamente lo que estamos es formalizando la misma operación *Goto* que usábamos en la conversión de NFA a DFA, pero esta vez escrita en función de los **items**. Por ejemplo, si I es el conjunto calculado anteriormente, entonces:

```
Goto(I,T) = { E -> T., E -> T. + E }
```

Dado que no existe ningún punto delante de un no-terminal, no es necesario computar la clausura.

Una vez que tenemos estas dos definiciones, podemos dar un algoritmo para construir el autómata finito determinista que reconoce los prefijos viables. El estado inicial de nuestro autómata será justamente $CL(E' \to E)$. Luego, repetimos la siguiente operación mientras sea necesario: por cada estado I y cada símbolo X, añadimos el estado $Goto(I, X)$ si no existe, y añadimos la transición $I \to^X J$. El algoritmo termina cuando no hay cambios en el autómata.

Apliquemos entonces este algoritmo a nuestra gramática para expresiones. Partimos del estado I_0 ya computado:

```
I0 = { E' -> .E,
       E -> .T,
       E -> .T + E,
       T -> .int,
       T -> .int * T,
       T -> .( E ) }
```

Calculemos ahora $Goto(I_0, E)$:

```
I1 = { E' -> E. }
```

Como no hay ningún punto delante de un no-terminal, la clausura se mantiene igual. Calculemos entonces $Goto(I_0, T)$:

```
I2 = { E -> T.,
       E -> T.+ E }
```

Igualmente la clausura no añade items. Calculemos ahora $Goto(I_0, int)$:

```
I3 = { T -> int.,
       T -> int.* T }
```

Y ahora $Goto(I_0, ()$:

```
I4 = { T -> (.E ) }
```

A este estado si tenemos que calcularle su clausura:

```
I4 = { T ->  (.E ),
       E -> .T,
       E -> .T + E,
       T -> .int,
       T -> .int * T,
       T -> .( E ) }
```

De modo que ya terminamos con I_0. Dado que en I_1 no hay símbolos tras un punto, calculemos entonces $Goto(I_2, +)$, y aplicamos la clausura directamente:

```
I5 = { E ->  T +.E,
       E -> .T,
       E -> .T + E,
       T -> .int,
       T -> .int * T,
       T -> .( E ) }
```

Calculamos $Goto(I_3, *)$:

```
I6 = { T ->  int *.T,
       T -> .int,
       T -> .int * T,
       T -> .( E ) }
```

Calculamos $Goto(I_4, E)$:

```
I7 = { T -> ( E.) }
```

Si ahora calculamos $Goto(I_4, T)$, y nos daremos cuenta que es justamente I_2. Por otro lado, afortunadamente $Goto(I_4, int) = I_3$. Y finalmente $Goto(I_4, ()$ es el propio estado I_4! Por otro lado, $Goto(I_5, E)$ es:

```
I8 = { E -> T + E. }
```

Mientras que $Goto(I_5, T)$ nos lleva de regreso a I_3, y $Goto(I_5, ()$ a I_4. Saltamos entonces para $Goto(I_6, T)$ que introduce un estado nuevo:

```
I9 = { T -> int * T. }
```

Por otro lado, $Goto(I_6, int) = I_3$ nuevamente, mientras que $Goto(I_6, ()$ nos regresa nuevamente a I_4. Finalmente, $Goto(I_7,))$ nos da el siguiente, y último estado del autómata (ya que I_8 e I_9 no tienen transiciones salientes):

```
I10 = { T -> ( E ). }
```

Para agilizar este algoritmo, podemos notar que, como dijimos anteriormente, solamente los item kernel son importantes. De hecho, podemos probar fácilmente que dos estados son iguales sí y solo si sus item kernel son iguales, dado que las operaciones de clausura sobre conjuntos de items kernel iguales nos darán el mismo conjunto final. Por lo tanto, en una implementación computacional (o un cómputo manual), si distinguimos los items kernel del resto de los items, cuando computamos un nuevo estado a partir de la función *Goto*, antes de computar su clausura vemos si su conjunto de items kernel coincide con el kernel de otro estado ya creado. En caso contrario, hemos descubierto un nuevo estado y pasamos a computar su clausura.

Parsing LR(0)

Una vez construido el autómata, podemos finalmente diseñar un algoritmo de parsing bottom-up. Este algoritmo se basa en la idea de verificar en cada momento si el estado de la pila es un prefijo viable, y luego, según el terminal que corresponda en ω, decidimos si la operación a realizar es **shift** o **reduce**. Para determinar si la pila es un prefijo viable, simplemente corremos el autómata construido en el contenido de la pila. Supongamos que este autómata se detiene en el estado I. Vamos que nos dicen los items de este estado sobre la operación más conveniente a realizar.

Si en este estado tenemos un item $X \leftarrow \alpha c \beta$, y $c\omega$ es la cadena de entrada (es decir, c es el próximo terminal a analizar), entonces es evidente que una operación de **shift** me seguirá manteniendo en la pila un prefijo viable. ¿Por qué? Pues porque al hacer **shift** el contenido de la pila ahora crece en c, y si vuelvo a correr el autómata desde el inicio de la pila, llegaré nuevamente al estado I justo antes de analizar c. Pero sé que desde I hay una transición con c a cierto estado J, que es justamente $Goto(I, c)$, por lo tanto terminaré en el estado J habiendo leído toda la pila. Luego, por definición de prefijo viable, como he podido reconocer el contenido de la pila, todo está bien.

Por otro lado, si en el estado I tengo un item de la forma $X \leftarrow \beta$, entonces es conveniente hacer una operación de **reduce** justamente en la producción $X \rightarrow \beta$. Para ver por qué esta operación me sigue manteniendo en la pila un prefijo viable, notemos que $X \rightarrow \beta$ quiere decir que hemos reconocido en la pila toda la parte derecha de esta producción. Entonces en la pila lo que tenemos en un **handle**, y por su propia definición reducir en un **handle** siempre es correcto.

De modo que tenemos un algoritmo. En cada iteración, corremos el autómata en el contenido de la pila, y analizamos cuál de las estrategias anteriores es válida según el contenido del estado en que termina el autómata. Si en algún momento el autómata no tiene una transición válida, tiene que ser con el último terminal que acabamos de hacer **shift** (ya que de lo contrario se hubiera detectado en una iteración anterior). Luego, este algoritmo reconoce los errores sintácticos lo antes posible. Es decir, nunca realiza una reducción innecesaria.

Por otro lado, puede suceder que en un estado del autómata tenga items que nos sugieran operaciones contradictorias. Llamaremos a estas situaciones, **conflictos**. En general, podemos tener 2 tipos de conflictos:

- Conflicto **shift-reduce** si ocurre que tengo un item que me sugiere hacer **shift** y otro que me sugiere hacer **reduce**.
- Conflicto **reduce-reduce** si ocurre que tengo dos items que me sugieren hacer **reduce** a producciones distintas.

En cualquiera de estos casos, tenemos una fuente de no-determinismo, pues no sabemos por cuál de estas operaciones se pudiera reconocer la cadena. Este no-determinismo se debe a que en el autómata no-determinista había más de un camino posible que reconocía la cadena, y al convertirlo a determinista, estos caminos se expresan como items contradictorios en el mismo estado. En estos casos, decimos que la gramática no es LR(0). Luego:

> Sea $G =< S, N, T, P >$ una gramática libre del contexto, G es LR(0) si y solo si en el autómata LR(0) asociado no existen conflictos **shift-reduce** ni conflictos **reduce-reduce**.

Notemos que no es posible que tengamos conflictos **shift-shift**, pues solamente hay un caracter c en la cadena ω, y por tanto hay un solo estado hacia donde hacer **shift**.

Desgraciadamente nuestra gramática favorita de expresiones no es LR(0). Sin ir más lejos, en el estado I_3 tenemos un conflicto **shift-reduce**. Podemos reducir `T -> int`, o hacer **shift** si viene un terminal `*`. Intuitivamente el problema es que la operación de **reduce** es demasiado permisiva. Donde quiera que encontremos un item **reduce** diremos que es conveniente reducir en esa producción, aunque sabemos que esto no siempre es cierto. De hecho, ya hemos tenido que lidiar con este problema anteriormente, en el algoritmo de parsing LL.

Parsing SLR(1)

Recordemos que en el parsing LL teníamos la duda de cuando era conveniente aplicar una producción $X \to \epsilon$, y definimos para ello el conjunto $Follow(X)$, que justamente nos decía donde era conveniente eliminar X. Pues en este caso, este conjunto también nos ayudará. Intuitivamente, si tenemos $X \to \beta$, solamente tiene sentido reducir si en el $Follow(X)$ aparece el terminal que estamos analizando. ¿Por qué? Pues porque de lo contrario no es posible que lo que nos quede en la pila sea un prefijo viable.

Supongamos que c es el terminal a analizar, $c \notin Follow(X)$ y hacemos la reducción. Entonces en el próximo **shift** tendremos en el tope de la pila la forma oracional Xc. Pero esta forma oracional no puede aparecer en niguna derivación extrema derecha, porque de lo contrario c sería parte del $Follow(X)$. Por tanto, si esta forma oracional no es válida, entonces ningún **handle** puede tener este prefijo. Por tanto ya no tenemos un prefijo viable. Incluso si lo siguiente que hacemos tras reducir en X no es **shift** sino otra secuencia de operaciones **reduce**, en cualquier caso si lo que queda una vez hagamos **shift** es un prefijo viable, entonces es porque $c \in Follow(X)$ (intuitivamente, aplicando las producciones en las que redujimos hasta que vuelva a aparecer X, obtendremos la forma oracional Xc nuevamente).

Justamente a esta estrategia denominaremos SLR(1), o *Simple LR look-ahead 1*, dado que

usamos un terminal de look-ahead para decidir si vale la pena reducir. Con esta estrategia, podemos comprobar que ya en el estado I_2 no hay conflicto, pues $* \notin Follow(T)$, porque cuando viene un terminal $*$ solo tiene sentido hacer **shift**, nunca **reduce**.

De forma análoga llamamos gramáticas SLR(1) a aquellas gramáticas donde, bajo estas reglas, no existen conflictos.

Intentemos entonces reconocer la cadena `int * ( int + int )` con nuestro parser SLR(1). Comenzamos por el estado inicial:

```
|int * ( int + int )
```

Como la pila está vacía, el autómata termina en el estado I_0. Dado que viene un terminal `int`, buscamos la transición correspondiente, que es justamente hacia el estado $Goto(I_0, int) = I_3$. Por tanto, como existe esta transición, significa que la acción a realizar es **shift**.

```
int|* ( int + int )
```

Ahora corremos nuevamente el autómata, ya sabemos que caerá en el estado I_3. Ahora podemos potencialmente reducir o hacer **shift**. Calculamos el $Follow(T)$

```
Follow(T) = { +, ), $ }
```

Por tanto, como `*` no está incluido en el `Follow(T)`, no hay conflicto, solamente no queda hacer **shift**, en este caso al estado I_6.

```
int *|( int + int )
```

Corremos de nuevo y sabemos que acabaremos en I_6. Aquí no hay reducciones, así que solo queda hacer **shift** hacia el estado I_4:

```
int * (|int + int )
```

En I_4 tampoco hay reducciones, así que hacemos **shift** hacia el estado I_3:

```
int * ( int|+ int )
```

Ahora interesantemente si tenemos que `+` pertenece al `Follow(X)`, por tanto la reducción aplica. Afortunadamente no hay transiciones en este estado con `+`, por lo que no hay conflicto. Aplicamos entonces la reducción:

```
int * ( T|+ int )
```

Ahora corremos el autómata nuevamente desde el inicio, siguiendo las transiciones (recordemos que mienstras estamos leyendo el contenido de la pila no nos importan los items). Terminamos en el estado I_2. En este estado podemos reducir a `E` o hacer **shift**. Pero resulta que `Follow(E)` no contiene al terminal `+`, por lo que la reducción no tiene sentido. Hacemos **shift** entonces:

```
int * ( T +|int )
```

Ahora el autómata termina en el estado I_5. En este estado, viniendo `int`, solamente tiene sentido hacer **shift** hacia el estado I_3:

```
int * ( T + int|)
```

Ahora estamos en una situación conocida. Pero en este caso, `)` sí está en el `Follow(T)`, y no hay transiciones con este símbolo, luego lo que queda es reducir:

```
int * ( T + T|)
```

Al correr el autómata, en vez de I_3 como en la última vez, ahora de I_5 pasaríamos directamente a I_2, donde nuevamente estamos en territorio conocido. Sin embargo, de nuevo en este caso `)` sí está en el `Follow(E)`, luego podemos reducir (y no hay transiciones con `)`):

```
int * ( T + E|)
```

Ahora volvemos a correr el autómata, pero en vez de el estado I_2, terminaríamos en el estado I_8, donde la única opción es reducir (una vez comprobamos el `Follow(E)`):

```
int * ( E|)
```

En este caso, al correr el autómata desde el inicio, terminamos en I_7, que nos dice **shift**:

```
int * ( E )|
```

Ahora de I_7 saltaríamos para I_{10}, que nos indica la reducción (dado que `$` sí está en el `Follow(X)`):

```
int * T|
```

En este caso, rápidamente caeremos en el estado I_9, que nos indica reducir:

```
T|
```

El autómata con esta pila termina en el estado I_2 nuevamente, pero ahora `$` es el terminal a analizar, por lo que hacemos la reducción:

```
E|
```

Y finalmente, en esta entrada el autómata nos deja en el estado I_1, que nos permite reducir por completo al símbolo especial `E'` y aceptar la cadena:

```
E'|
```

De esta forma, el algoritmo de parsing SLR(1) ha logrado obtener una derivación extrema derecha de nuestra cadena favorita, pero empleando una gramática mucho más expresiva y "natural" que la gramática LL correspondiente.

De todas formas, muchas gramáticas medianamente complicadas no son SLR(1), por lo que necesitaremos un parser de mayor potencia. Para ello, tendremos que refinar aún más el criterio con el cuál se producen los **reduce**.

Parsing LR(1)

Veamos a continuación un ejemplo de una gramática clásica que no es SLR(1):

```
S -> E
E -> A = A | i
A -> i + A | i
```

Esta gramática representa un subconjunto del lenguaje de las ecuaciones algebraicas, donde tanto en la parte derecha como en la izquierda del token = podemos tener una expresión aritmética cualquiera. Veamos que sucede al construir el autómata SLR(1):

```
I0 = {
    S -> .E
    E -> .A = A
    E -> .i
    A -> .i + A
    A -> .i
}
```

Viendo los items de este estado, ya podemos intuir dónde podría haber problemas. Al hacer `Goto(I0, i)` aparecerán dos items **reduce** con parte izquierda distinta:

```
Goto(I0, i) = {
    E -> i.
    A -> i.+ A
    A -> i.
}
```

En este estado aperece entonces un conflicto **reduce-reduce**, ya que $Follow(E) = \{\$\}$, y $\$ \in Follow(A)$, puesto que `A` aparece como parte derecha de una producción de `E`. Por tanto esta gramática no es SLR(1). Sin embargo, la gramática no es ambigua, y esto es fácil de demostrar. Intuitivamente, la única cadena donde pudiera haber ambiguedad es justamente la cadena `i` (es el único token que es generado por más de un no-terminal). Sin embargo, para esta cadena, la única derivación posible es `S -> E -> i`. Aunque `A -> i` es una producción, la forma oracional `i` no es **handle** de `A`. Si solo existe un `i` en la pila, este tiene que ser generado por el no-terminal `E`, pues de lo contrario no sería posible reducir a `S`.

Sin embargo, nuestro parser SLR(1) no es suficientemente inteligente para determinar esto. Al encontrarse con la forma oracional `i` en la pila, en principio, el autómata dice que `A -> i` es una reducción posible. Sin embargo, sabemos que esta reducción es inválida, porque luego quedaría `A` en la pila, que no es una forma oracional válida en ninguna derivación extrema derecha. De la producción `E -> A = A` podemos ver que esta gramática nunca genera una `A` sola. En otras palabras, nuestra heurística SLR(1) para detectar **handles** (reducir en `X` para todo terminal en el `Follow(X)`) es demasiado débil para manejar esta situación, y produce un falso positivo, al determinar que la forma oracional `i$` es un **handle** de `A`, cuando realmente no lo es.

La pregunta es entonces, ¿por qué surge este conflicto? Qué falla en la heurística SLR(1) que produce estos falsos positivos? Evidentemente el conjunto Follow es en ocasiones demasiado grande, y contiene tokens para los cuáles no es válida una operación **reduce** en ese estado particular. Tratemos de rastrear, durante la construcción del autómata, dónde es que se introducen estos tokens inválidos.

Comenzamos por el estado inicial nuevamente, pero viéndolo paso a paso a medida que se computa la clausura. Comenzamos por el *kernel*:

```
I0 = {
```

```
    S -> .E
    ...
}
```

En este punto, el único item de este estado indica que esperamos encontrar en la cadena una forma oracional que se reduzca a `E`. Por tanto, añadimos las producciones de `E`:

```
I0 = {
    S -> .E
    E -> .A = A
    E -> .i
    ...
}
```

Hasta aquí no hay problemas, pues ni siquiera hay dos items con la misma parte derecha. Entonces tenemos que adicionar las producciones de `A`:

```
I0 = {
    S -> .E
    E -> .A = A
    E -> .i
    A -> .i + A
    A -> .i
}
```

Y aquí es donde podemos tener la primera pista de que viene un conflicto. Tenemos dos items que tienen la misma parte derecha (`E -> .i` y `A -> .i`). Por tanto, tras el próximo **shift** llegaremos a un estado con dos **reduce** a no-terminales distintos. Ahora, recordemos que el conflicto va a suceder porque `$` está en la intersección de los Follow. Sin embargo, veamos por qué motivo aparece `A -> .i` en este estado. Justamente, es por culpa de la producción `E -> .A = A` que tenemos que expandir los items de `A`. Recordemos entonces qué significa este item para nosotros: básicamente representa que a partir de este punto en la cadena de entrada, estamos esperando que aparezca *algo que se pueda reducir* a la forma oracional `A = A`. Por tanto, como esta forma oracional empieza con `A`, debemos expandir sus producciones.

Pero sí analizamos cuidadosamente el significado del item anterior, veremos que no tiene sentido reducir a este `A` si aparece `$` como *look-ahead*. ¿Por qué? Pues precisamente, el item nos dice que lo que viene en la cadena, si resultara que es correcta, debería reducirse a la forma oracional `A = A`. Por tanto, la primera parte de esa supuesta cadena, que se debería reducir al primer `A` de la forma oracional, solamente sería correcta si justo detrás de ese `A` viniera un token `=`. De lo contrario no podríamos seguir reduciendo el resto de la forma oracional.

Es decir, en el momento qué por culpa del item `E -> .A = A` nos toca adicionar el item `A -> .i`, lo que estamos esperando es que ese `i` se reduzca a `A` e inmediatamente después venga un `=`. El propio item nos está diciendo eso. Por tanto, si en el siguiente estado resultara que felizmente apareció el token `i` en la pila, antes de reducirlo ingenuamente a `A`, deberíamos verificar si justo detrás viene el `=` que estábamos esperando. De lo contrario, podríamos

decir que la reducción no tiene sentido, porque el item "padre" de este item (es decir, `E -> .A = A`) se va a quedar esperando un = que no viene en la cadena.

El problema con la heurística SLR radica justamente en que calculamos el Follow de cualquier no-terminal de forma *global*. Es decir, no tenemos en cuenta qué según las producciones que se vayan aplicando, solamente una parte de ese Follow es la que realmente puede aparecer a continuación. En este caso particular $Follow(A) = \{=, \$\}$. Pero cada uno de esos tokens está en el Follow por un motivo *distinto*. Justamente = aparece en el `Follow(A)` por culpa de la *primera* `A` en la producción `E -> A = A`. Pero $ aparece por culpa de la *segunda* `A` de esa producción. De cierta forma, es como si tuviéramos *dos instancias* distintas del mismo no-terminal `A`, aquella que aparece delante del token = y aquella que aparece detrás. Entonces cuando estamos parseando una cadena, y el autómata pasa al estado `E -> .A = A`, estamos esperando una `A`, pero no cualquier `A`, sino aquella *instancia* de `A` que viene delante del =. El error en la heurística SLR es justamente que no puede identificar estas situaciones distintas. Para el autómata SLR toda `A` es la misma `A`, por tanto tiene sentido reducir siempre con el mismo *look-ahead*.

¿Cómo podemos entonces identificar de forma distinta a cuál de las posibles `A` nos estamos refiriendo? Precisamente, durante la construcción del autómata, cuando el item `E -> .A = A` nos genera un item `A -> .i`, sabemos a "cuál" `A` nos referimos. Es justamente aquella que estaba detrás del punto. Por tanto, podemos en este momento decir qué es lo que puede venir detrás de esa `A` particular si se aplica esta producción particular. Lo que haremos entonces es adicionar a cada item un conjunto de tokens, que nos dirán explícitamente cuándo es que tiene sentido hacer **reduce**. Y este conjunto de tokens lo iremos calculando a medida que se crean los nuevos items, justamente mirando en el item "padre" que lo generó qué es lo que puede venir detrás de cada no-terminal.

Vamos a introducir entonces el concepto de **item LR(1)**, que no es más que un **item LR(0)** normal junto a token:

> Sea $G =< S, N, T, P >$ una gramática libre del contexto, un **item LR(1)** es una expresión de la forma $X \rightarrow \alpha \cdot \beta, c$ donde $X \rightarrow \alpha\beta$ es una producción ($\alpha, \beta \in (N \cup T)^*$), y $c \in T$.

El nuevo token que hemos adicionado a cada item nos servirá para ir rastreando qué terminales pueden aparecer el Follow de la forma oracional que estamos intentando reducir. De modo que un item de la forma $X \rightarrow \alpha \cdot \beta, c$ en un estado del autómata representa que ya hemos reconocido la parte α de la forma oracional, y esperamos reconocer la parte β, y que una vez reconocida toda esta porción β, esperamos que venga exactamente un terminal c. Por tanto, en algún momento tendremos un item $X \rightarrow \delta\cdot, c$, y entonces la operación de **reduce** la aplicaremos solamente si el siguiente terminal es c. A este token asociado a cada item le llamaremos *look-ahead*, y a la parte $X \rightarrow \alpha \cdot \beta$ le llamaremos centro. De modo que un item LR(1) está compuesto por un centro, que es un item LR(0), y un token *look-ahead*.

A partir de este nuevo tipo de item, vamos a construir un autómata similar al SLR(1), que llamaremos **LR(1) canónico** o simplemente LR(1). Para ello, necesitaremos definir cuál es el item inicial, y cómo se computan nuevos items a partir de los items ya existentes. El item LR(1) inicial es fácil de definir. El centro es idéntico al item inicial del autómata SLR: es decir, $S \rightarrow \cdot E$ (siendo `S` el nuevo símbolo inicial de la gramática aumentada, y `E` el símbolo inicial

de la gramática original). Ahora, para definir el token *look-ahead*, volvamos al significado de este item. Básicamente $S \rightarrow \cdot E$ significa que esperamos encontrar una forma oracional que se pueda reducir a `E`, y por tanto lo único que puede venir posteriormente es justamente `$`. De este modo el item LR(1) inicial significamente exactamente que queremos reducir **toda** la cadena a un simbolo `E`.

> Sea $G =< S, N, T, P >$ una gramática libre del contexto, S' el símbolo inicial de la gramática autmentada, entonces el item LR(1) inicial es $S' \rightarrow \cdot S, \$$.

Veamos entonces qué sucede con cualquier otro item LR(1). Recordemos que en SLR teníamos dos tipos de items, los *kernel* y los *no-kernel*. Aquí tendremos la misma separación. Si tenemos un item LR(1) $X \rightarrow \alpha \cdot x\beta, c$, donde $x \in T$, la operación **shift** nos generará el item $X \rightarrow es\alpha x \cdot \beta, c$. Dado que todavía estamos reconociendo la parte derecha de esta producción, el *look-ahead* se mantiene igual. Por otro lado, si tenemos $X \rightarrow \alpha \cdot Y\beta, c$ con $Y \in N$, entonces tenemos que computar (además de correr el punto) todos los items con centro $Y \rightarrow \cdot\delta$. La pregunta es entonces cuál es el *look-ahead* correspondiente. Básicamente la pregunta es, si logramos reducir a δ al Y que estamos esperando, ¿qué puede venir detrás? La respuesta es, en principio, todo terminal en el $First(\beta)$, y *nada más*.

Ahora, puede suceder que $\beta \rightarrow^* \epsilon$, es decir, que la parte detrás de Y no exista, o que desaparezca en el futuro. En este caso, tendremos que $\epsilon \in First(\beta)$. Pero no tiene sentido alguno decir que detrás de Y esperamos que venga ϵ (incluso definimos anteriormente que $\epsilon \notin Follow(Y)$ para todo Y). Sin embargo, en este caso, si β desaparece, entonces lo único que puede venir detrás de Y, es justamente el *look-ahead* de X. Por ejemplo, para el item $X \rightarrow \alpha \cdot Y, c$ generamos el item $Y \rightarrow \cdot\delta, c$, pues como Y está al final del X, le sigue lo mismo que habíamos decidido que seguía a X. Podemos generalizar de la siguiente manera, extendiendo los conceptos definidos para SLR:

> Sea I un conjunto de **items LR(1)** (kernel o no), el conjunto clausura de I se define como $CL(I) = I \cup \{X \rightarrow \beta, b\}$ tales que $Y \rightarrow \alpha X\delta, c \in CL(I)$ y $b \in First(\delta c)$.
>
> Sea I un conjunto de **items LR(1)**, se define la función $Goto(I, X) = CL(\{Y \rightarrow \alpha X\beta, c | Y \rightarrow \alpha X\beta, c \in I\})$

Para construir el autómata, seguimos el mismo algoritmo que para SLR. La función **Goto** para un conjunto de items (un estado) se define de igual forma como el conjunto de items (estado) que se obtienen de aplicar **Goto** a cada item en el conjunto origen. La función **Clausura** igualmente se define de forma recursiva como la clausura transitiva de cada uno de los items del propio conjunto. De la misma forma, tendremos conflictos **shift-reduce** o **reduce-reduce** en algún estado, si los *look-ahead* de algunas producciones **reduce** coinciden con otras, o con una de las transiciones salientes.

Pasemos entonces a construir el autómata LR(1) de la gramática anterior:

```
S -> E
E -> A = A | i
A -> i + A | i
```

Comenzamos por el estado inicial:

```
I0 = {
```

```
    S -> .E, $
}
```

Vamos a añadir entonces las producciones de `E`. Notemos que como en `S -> E` detrás de `E` no viene nada, el *look-ahead* será justamente `$`:

```
I0 = {
    S -> .E,     $
    E -> .A = A, $
    E -> .i,     $
}
```

Hasta el momento hemos obtenido un estado inicial equivalente al del autómata SLR correspondiente. Adicionamos entonces las producciones de `A`, y veremos el primer cambio importante. El centro de estos items serán (de forma equivalente al caso SLR), `A -> .i + A` y `A -> .i`. Sin embargo, al calcular el *look-ahead*, aplicando la definición, tenemos que computar $First(= \$)$ que es el token `=`. Luego:

```
I0 = {
    S -> .E,     $
    E -> .A = A, $
    E -> .i,     $
    A -> .i + A, =
    A -> .i,     =
}
```

Y ya podemos intuir cómo el autómata LR resolverá el conflicto **reduce-reduce** que teníamos anteriormente, ya que las producciones a reducir en el próximo estado tienen *look-ahead* distinto. Por completitud, continuemos con el resto del autómata.

```
I1 = Goto(I0, E) = {
    S -> E., $
}

I2 = Goto(I0, A) = {
    E -> A.= A, $
}

I3 = Goto(I0, i) = {
    E -> i.,    $
    A -> i.+ A, =
    A -> i.,    =
}
```

Como intuíamos, el estado I_3, que anteriormente tenía un conflicto **reduce-reduce**, ahora es válido. Veamos el resto de los estados:

```
I4 = Goto(I2, =) = {
    E ->  A =.A, $
    A -> .i + A, $
```

```
    A -> .i,     $
}

I5 = Goto(I3, +) = {
    A ->  i +.A, =
    A -> .i + A, =
    A -> .i,     =
}
```

Ahora veamos los estados que se derivan de estos. Comenzaremos a notar que aparecen estados muy similares a los anteriores, pero con conjuntos *look-ahead* distintos:

```
I6 = Goto(I4, A) = {
    E -> A = A., $
}

I7 = Goto(I4, i) = {
    A -> i.+ A, $
    A -> i.,    $
}

I8 = Goto(I5, A) = {
    A -> i + A., =
}

I9 = Goto(I5, i) = {
    A -> i.+ A, =
    A -> i.,    =
}
```

Como vemos, S_7 y S_9 son estados en principio idénticos, excepto porque los *look-ahead* asociados a cada item son distintos. Esta repetición de estados casi iguales será el próximo problema a resolver, pero por el momento continuemos con el autómata:

```
I10 = Goto(I7, +) = {
    A ->  i +.A, $
    A -> .i + A, $
    A -> .i,     $
}

Goto(I9, +) = S5

I11 = Goto(I10, A) = {
    A ->  i + A., $
}

Goto(I10, i) = S7
```

El autómata resultante tiene 12 estados. A modo de comparación, el autómata SLR corres-

pondiente tiene 9 estados. Como ya tenemos práctica, lo mostraremos sin más explicación:

```
I0 = {
    S -> .E
    E -> .A = A
    E -> .i
    A -> .i + A
    A -> .i
}

I1 = Goto(I0, E) = {
    S -> E.
}

I2 = Goto(I0, A) = {
    E -> A.= A
}

I3 = Goto(I0, i) = {
    E -> i.
    A -> i.+ A
    A -> i.
}

I4 = Goto(I2, =) = {
    E ->  A =.A
    A -> .i+ A
    A -> .i
}

I5 = Goto(I3, +) = {
    A ->  i +.A
    A -> .i + A
    A -> .i
}

I6 = Goto(I4, A) = {
    E -> A = A.
}

I7 = Goto(I4, i) = {
    A -> i.+ A
    A -> i.
}

I8 = Goto(I5, A) = {
    A -> i + A.
```

```
}

Goto(I5, i) = I7

Goto(I7, +) = I5
```

Los estados adicionales necesarios en LR son justamente aquellos donde existen conflictos.

Parsing LALR(1)

Si observamos el autómata LR(1) nuevamente, notaremos algunos estados que son muy semejantes, y solo se diferencian en los conjuntos de *look-aheads*. En particular, el estado I_7 y el estado I_9 tienen los mismos centros:

```
I7 = Goto(I4, i) = {
    A -> i.+ A, $
    A -> i.,    $
}

I9 = Goto(I5, i) = {
    A -> i.+ A, =
    A -> i.,    =
}
```

Al igual que los estados I_5 e I_10:

```
I5 = Goto(I3, +) = {
    A ->  i +.A, =
    A -> .i + A, =
    A -> .i,     =
}

I10 = Goto(I7, +) = {
    A ->  i +.A, $
    A -> .i + A, $
    A -> .i,     $
}
```

Y los estados I_8 e I_11:

```
I8 = Goto(I5, A) = {
    A -> i + A., =
}

I11 = Goto(I10, A) = {
    A -> i + A., $
}
```

Intuitivamente, estos son los estados que ayudan a desambiguar el autómata SLR, pues separan en diferentes subconjuntos de terminales lo que antes era el Follow de un no-terminal. De cierta forma este es el precio a pagar por el poder adicional del autómata LR sobre el SLR: es necesario separar en mútiples estados lo que antes era un solo estado, para poder discriminar con exactitud qué tokens activan una reducción.

Afortunadamente, en ocasiones podemos obtener lo mejor de ambos mundos. ¿Qué sucede si intentamos "compactar" estos estados "duplicados"? Idealmente, si esta compactación no introdujera nuevos conflictos, lograríamos un autómata con menos estados y el mismo poder de reconocimiento. Veamos como podríamos proceder. Tomemos por ejemplo los estados I_7 e I_9 y definamos un nuevo estado $I_{7,9}$. Para ello simplemente combinamos los *look-ahead* de cada par de items iguales en un conjunto:

```
I7-9 = {
    A -> i.+ A, =$
    A -> i.,    =$
}
```

De la misma forma, podemos hacer con los pares de estados restantes:

```
I5-10 = {
    A ->  i +.A, =$
    A -> .i + A, =$
    A -> .i,     =$
}

I8-11 = {
    A -> i + A., =$
}
```

La primera verificación que necesitamos hacer es si estos nuevos estados crean conflictos en sí mismos. Es decir, si al combinar, aparece un conflicto **reduce-reduce** que antes no existía, dado por dos reducciones distintas cuyos *look-ahead* ahora tengan intersección. En este caso vemos que no sucede, pues en los estados donde hay reducciones, afortunadamente son al mismo no-terminal, por tanto no hay ambigüedad.

Luego tenemos que ver cómo se comporta el autómata si reemplazamos los estados originales por estos nuevos "estados combinados". Para ello, tenemos que ver que sucede con las transiciones entrantes y salientes. En principio, todas las transiciones que iban a parar a alguno de los estados originales, irán a parar al nuevo estado combinado. Veamos entonces qué aristas entrantes tenía cada estado original:

```
I7-9  | I7  = Goto(I4, i) = Goto(I10, i)
      | I9  = Goto(I5, i)

I5-10 | I5  = Goto(I3, +) = Goto(I9, +)
      | I10 = Goto(I7, +)

I8-11 | I8  = Goto(I5, A)
      | I11 = Goto(I10, A)
```

Tenemos entonces que factorizar todas estas aristas entrantes hacia los nuevos estados, y comprobar que no ocurran conflictos. Por ejemplo, como `Goto(I4, i) = I7`, ahora esa arista irá hacia `I7-9`. Por otro lado, como `Goto(I5, i) = I9`, esa arista también irá para `I7-9`. Y en este segundo caso, como en estado de salida también va a ser parte de un estado combinado, realmente lo que sucederá es que tendremos una arista de `I5-10` hacia `I7-9`. Pero para que esto sea posible, necesitamos que la otra arista que salía de `I10` también caiga en el nuevo estado combinado, pues de lo contrario tendríamos una ambigüedad. Afortunadamente, en este caso `Goto(I10, i) = I7`, por lo que no existe conflicto. Luego, procedamos a compactar todas las transiciones entrantes:

```
I7-9  = Goto(I4, i) = Goto(I5-10, i)
I5-10 = Goto(I3, +) = Goto(I7-9, +)
I8-11 = Goto(I5-10, A)
```

En principio, nos queda ver las aristas salientes de estos estados combinados. En este caso particular todas las aristas salientes ya han sido analizadas, como aristas entrantes de otros estados. En el caso general, es posible que existan aristas salientes de un estado combinado de vayan a parar a estados distintos. Por ejemplo, si hubiera una arista de I_7 con un token `c` hacia un estado I_i, y otra arista desde I_9 con el mismo token `c` hacia otro estado I_j, que no fueran combinables (no tuvieran el mismo centro), entonces no sería posible realizar esta compactación.

Veamos el autómata final que hemos obtenido al combinar los estados con el mismo centro:

```
I0 = {
    S -> .E,     $
    E -> .A = A, $
    E -> .i,     $
    A -> .i + A, =
    A -> .i,     =
}

I1 = Goto(I0, E) = {
    S -> E., $
}

I2 = Goto(I0, A) = {
    E -> A.= A, $
}

I3 = Goto(I0, i) = {
    E -> i.,    $
    A -> i.+ A, =
    A -> i.,    =
}

I4 = Goto(I2, =) = {
    E ->  A =.A, $
```

```
    A -> .i + A, $
    A -> .i,     $
}

I5-10 = Goto(I3, +) = Goto(I7-9, +) {
    A ->  i +.A, =$
    A -> .i + A, =$
    A -> .i,     =$
}

I6 = Goto(I4, A) = {
    E -> A = A., $
}

I7-9 = Goto(I4, i) = Goto(I5-10, i) {
    A -> i.+ A, =$
    A -> i.,    =$
}

I8-11 = Goto(I5-10, A) = {
    A -> i + A., =$
}
```

El nuevo autómata que hemos construido tiene exactamente 9 estados, la misma cantidad que el autómata SLR, y sin embargo no presenta conflictos. Este autómata se denomina LALR, y constituye el resultado más importante en la práctica para la construcción de compiladores, ya que la mayoría de los generadores de parsers autómaticos usados en la industria construyen este tipo de autómatas. Esto se debe a que combina de forma ideal un poder reconocedor muy similar al LR, con una cantidad de estados mucho menor, proporcional al SLR.

Intuitivamente, el motivo por el que este autómata no presenta conflictos, y el SLR sí, se debe a que en este caso hemos hecho un análisis más riguroso de los Follow primero (construyendo el autómata LR), y solo entonces hemos intentado combinar los estados. Esto fue posible ya que los estados con los mismos centros no eran aquellos donde se producían los conflictos **reduce-reduce** en el autómata SLR. Es decir, al aplicar la técnica LR, y analizar cuidadosamente los Follow, logramos evitar el conflicto en el estado I_3, pero luego esa misma técnica nos llevó a crear estados independientes innecesarios. De cierta forma, la técnica LR es demasiado rigurosa, y nos lleva incluso a intentar evitar conflictos que en realidad no van a ocurrir. La técnica LALR entonces reconoce estas situaciones donde tenemos "demasiado rigor" y simplifica el autómata tanto como sea posible.

Siguiendo esta línea de pensamiento, debería ser posible construir el autómata LALR directamente, sin necesidad de primero construir el LR para luego combinar los estados innecesarios. De hecho, esto es posible, y es lo que hacen los generadores de parsers usados en la práctica. Aunque no presentaremos un algoritmo detallado para esto, podemos ver intuitivamente que los estados se pueden ir combinando "sobre la marcha", a medida que se descubren estados con el mismo centro que otros ya creados, y arreglando las transiciones

existentes en caso de ser necesario.

Un punto de importancia en el autómata LALR, es que no se puede hacer "una parte" de este. Es decir, o bien todos los estados con el mismo centro de combinan, y no aparece ningún conflicto nuevo, o no se combina ninguno y el autómata se queda LR. De modo que no existen "grados" de LALR.

Implementación del parser LR

Veamos entonces como construir un algoritmo de parsing lineal que obtenga el árbol de derivación correspondiente. Recordemos que lo que tenemos hasta el momento es un autómata que nos permite reconocer si el contenido de la pila es un prefijo viable. En principio, en cada iteración, tras realizar un **shift** o un **reduce**, es necesario volver a correr el autómata en todo el contenido de la pila, para determinar en qué estado termina, y poder decidir la próxima operación. Esto es innecesariamente costoso. Intuitivamente, dado que tras una operación **shift** o **reduce** sabemos exactamente como cambia la pila, deberíamos poder "hacer backtrack" en el autómata, y solamente ejecutar la parte necesaria para reconocer el nuevo sufijo de la pila.

Por ejemplo, supongamos que tenemos un estado del parser $\alpha|c\omega$, y al ejecutar el autómata, terminamos en un estado I_i que indica **shift** con el token `c`. Si $Goto(I_i, c) = I_j$, entonces sabemos que la siguiente iteración terminaremos en el estado I_j. Efectivamente, tras un **shift** tendremos un nuevo estado en la pila $\alpha c|\omega$, y como el autómata es determinista, tras reconocer α tendrá que terminar necesariamente en el estado I_i. Luego, el token `c` lo envía al estado I_j, precisamente porque esa es la definición de la función **Goto**. Por tanto, tras una operación de **shift**, no es necesario volver a correr el autómata en todo el contenido de la pila. Simplemente podemos transitar directamente al estado $Goto(I_i, c)$.

Veamos que sucede tras una operación **reduce**. Sea $\alpha\beta|\omega$ el estado de la pila antes del **reduce**, y $\alpha X|\omega$ el estado después de reducir $X \to \beta$. Supongamos que el autómata, tras reconocer α, cae en el estado I_i. Entonces tras reconocer αX debe caer en el estado $Goto(I_i, X)$ por definición. Por tanto, una vez sacados $|\beta|$ terminales de la pila, solamente necesitamos ser capaces de "recordad" en que estado estaba el autómata cuando reconoció los primeros $|\alpha|$ terminales, y de ahí movernos una sola transición. Para ello, sencillamente almacenaremos en la pila, además de la forma oracional que se está construyendo, también los estados que transita el autómata en cada símbolo (ya sea almacenando pares $< X, i >$ o con una pila paralela para los estados). Por tanto, cuando extraemos los $|\beta|$ terminales de la pila, en el tope está justamente el estado I_i.

Con estas dos estrategias, podemos demostrar que tenemos un algoritmo de parsing lineal. Definamos entonces de una vez y por todas este algoritmo formalmente. Para ello vamos a construir una tabla, que llamaremos **tabla LR**, y que nos indicará en cada situación qué hacer (al estilo de la **tabla LL**). Mostraremos a continuación la **tabla LR** para el autómata construido anteriormente, y luego veremos paso a paso los detalles sobre su construcción:

Estado	=	+	i	$	E	A
0			S3	S->E	1	2

Estado	=	+	i	$	E	A
1				OK		
2	S4					
3	A->i		S5		E->i	
4			S7			6
5			S9			8
6				E -> A = A		
7		S10		A -> i		
8	A -> i + A					
9	A -> i		S5			
10			S7			11
11				A -> i + A		

La tabla LR contiene una fila por cada estado del autómata, y una columna por cada símbolo (terminales y no-terminales). Usaremos la notación $T[i, x]$ para referirnos a la entrada asociada al símbolo x en el estado I_i:

- Si $S \to E\cdot, \$$ pertenece al estado I_i, entonces la entrada $T[i, \$] = OK$.
- Si $Goto(I_i, X) = I_j$:t
  - Si $X \in N$, entonces $T[i, X] = j$.
  - Si $X \in T$, entonces $T[i, X] = S_j$ (**shift**):
- Si el item $Y \to \beta\cdot, c$ está en el estado I_i, entonces $T[i, c] = Y \to \beta$ (**reduce**).

Para usar la tabla, veamos qué significa cada posible valor en una entrada. Sea $S = \alpha|c\omega$ el estado de la pila de símbolos, e I_i el estado del autómata en el tope de la pila de estados (asumiendo una implementación basada en dos pilas paralelas). Buscamos entonces la entrada $T[i, c]$, y según su valor realizamos la siguiente operación:

- Si $T[i, c] = OK$, entonces terminamos de parsear y la cadena se reconoce.
- Si $T[i, c] = S_j$, entonces hacemos **shift**, la pila de símbolos se convierte en $\alpha c|\omega$, y ponemos el estado I_j en el tope de la pila de estados.
- Si $T[i, c] = X \to \delta$, entonces se garantiza que $\alpha = \beta\delta$. Extraemos $|\delta|$ elementos de la pila de símbolos **y** de la pila de estados. Sea I_k el estado que queda en el tope de la pila de estados; colocamos a X en el tope de la símbolos y colocamos $T[k, X]$ en el tope de la pila de estados.
- En cualquier otro caso, se ha encontrado un error.

Para entender por qué el algoritmo descrito anteriormente funciona, tratemos de interpretar qué significa cada entrada en la tabla. Una entrada de la forma S_j significa que con el token correspondiente existe una transición hacia el estado I_j. Por tanto, la operación a realizar es **shift**, y como ya hemos visto anteriormente, en la pila de estados sabemos que el siguiente estado hacia el que transitar será justamente j. Una entrada de la forma $X \to \beta$ significa que existe un item **reduce** con el token correspondiente de *look-ahead*. Por lo tanto se realiza la operación de **reduce**. Como vimos anteriormente, en el tope de la pila de estados quedará el estado I_j que habíamos transitado justo antes de reconocer la producción reducida. Por tanto, de ese estado anterior, el nuevo estado al que transitaremos es justamente $T[j, X]$.

El único punto que nos queda por discutir, es cómo se construye el árbol de derivación. Esbozaremos un algoritmo para esto, que ejemplificaremos más adelante. Dado que la derivación se construye de abajo hacia arriba, intuitivamente puede verse que construiremos el árbol empezando por las hojas. La idea general consiste en mantener una tercera pila donde se irán acumulando sub-árboles de derivación. Cada vez que se haga una operación **reduce** $X \to \beta$, tendremos en el tope de esta pila $|\beta$ sub-árboles de derivación, que agruparemos bajo una nueva raíz X. Al finalizar el reconocimiento, en la pila quedará solamente un árbol, que será la derivación de toda la cadena.

Veamos ahora un ejemplo de cómo funciona el algoritmo de parsing bottom-up con la cadena `i = i + i`, usando la tabla definida anteriormente. Ilustraremos la ejecución del algoritmo, representando el estado de la pila de símbolos de la forma usual, y además representando las dos pilas correspondientes a los estados y los sub-árboles de derivación. Para representar un árbol, usaremos la notación $X(t_1, t_2, \ldots, t_n)$, donde X es el símbolo de la raíz, y t_i es una hoja, o un árbol a su vez. Comenzamos entonces con todas las pilas en su estado inicial:

```
Symbols ::  |i = i + i $
States  :: 0|
Trees   ::  |
```

Consultamos la tabla. En el estado 0, con *look-ahead* `i`, la operación es **shift 3**. Notemos cómo en este estado inicial, todos los demás token darían error. Colocamos el terminal, el estado, y el árbol recién creado en el tope de las respectivas pilas:

```
Symbols ::   i|= i + i $
States  :: 0 3|
Trees   ::   i|
```

Nos encontramos ahora en el estado 3, con *look-ahead* `=`, la operación es **reduce** `A -> i`. Entonces sacamos el token `i` de la pila y lo reemplazamos por `A`. El estado 3 también se saca y se reemplaza por $Goto(0, A) = 2$ (0 es el estado que queda justo debajo de 3 en la pila). El árbol i se ubica como único hijo del nuevo árbol A creado:

```
Symbols ::    A|= i + i $
States  ::  0 2|
Trees   :: A(i)|
```

Consultamos de nuevo la tabla, $T[2, =]$ es **shift 4**:

```
Symbols ::    A =|i + i $
States  ::  0 2 4|
Trees   :: A(i) =|
```

Consultamos de nuevo, $T[4, i]$ es **shift 7**. En la pila de árboles se nos han ido acumulando sub-árboles distintos, que serán mezclados en el futuro:

```
Symbols ::    A = i|+ i $
States  ::  0 2 4 7|
Trees   :: A(i) = i|
```

Ahora $T[7, +]$ nos dice **shift 10**:

```
Symbols ::    A = i +|i $
States  :: 0 2 4 7 10|
Trees   :: A(i) = i +|
```

Por último, $T[10, i]$ nos dice nuevamente **shift 7**:

```
Symbols ::    A = i + i|$
States  :: 0 2 4 7 10 7|
Trees   :: A(i) = i + i|
```

Volvemos entonces al estado I_7, pero ahora con *look-ahead* `$` la operación indicada es **reduce** `A -> i`. El nuevo estado será $Goto(10, A) = 11$.

```
Symbols ::       A = i + A|$
States  ::   0 2 4 7 10 11|
Trees   :: A(i) = i + A(i)|
```

Ahora $T[11,]$\$ también nos indica **reduce** pero en este caso en `A -> i + A`. Vamos a sacar entonces 3 elementos de cada pila. En la pila de símbolos, sustituímos `i + A` por `A`. En la pila de estados, sacamos los tres últimos elementos, y ponemos $Goto(4, A) = 6$. En la pila de árboles, sacamos los tres árboles del tope y creamos un nuevo árbol A con esos tres como hijos:

```
Symbols ::              A = A|$
States  ::            0 2 4 6|
Trees   :: A(i) = A(i,+,A(i))|
```

Hemos dado un gran salto de fé, pues al sacar los últimos tres estados, hemos confiado en que la pila de estados nos recordará dónde estaba el autómata justo antes de hacer el primer **shift** que dio paso a toda la forma oracional `i + A`. Continuemos entonces con $T[6, \$]$ que nos dice justo lo que esperábamos: **reduce** `E -> A = A`. Repetimos toda la operación de reducción que ya conocemos, siendo $Goto(0, E) = 1$ el último estado que tendremos que analizar:

```
Symbols ::                    E|$
States  ::                  0 1|
Trees   :: E(A(i),=,A(i,+,A(i)))|
```

Finalmente, $T[1, E]$ nos dice que la cadena ha sido parseada. En el tope de la pila de símbolos queda el símbolo inicial, y en la pila de árboles hay un solo árbol que contiene la derivación que hemos construido. Honestamente, en una implementación computacional concreta la pila de símbolos es innecesaria, pues todas las decisiones se toman mirando solamente la pila de estados, y el resultado necesario se computa en la pila de árboles. De cierta forma, esta pila de árboles es un *upgrade* de la pila de símbolos. Hemos hecho la distinción en este ejemplo por cuestiones puramente didácticas, pero en la práctica, la pila de símbolos no se usa.

Comparaciones entre LL, SLR, LR y LALR

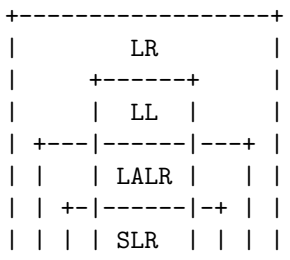
Hemos visto hasta el momento los dos paradigmas fundamentales para el parsing determinista de gramáticas libres del contexto. Hagamos entonces una reflexión final sobre los resultados que hemos obtenido, y las herramientas que hemos desarrollado.

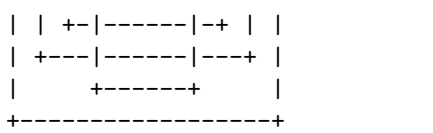
El parser LL es posiblemente el más sencillo de todos los parsers que es útil. En muchos contextos donde se tiene que diseñar un lenguaje bien sencillo (por ejemplo, un DSL integrado en un sistema más complejo), este parser puede brindar una solución fácil y eficiente. La mayor ventaja es que se puede escribir directamente, sin necesidad de usar un generador de parser. Los conjuntos First y Follow se computan a mano, y luego se escriben cada uno de los métodos recursivos asociados a las producciones. Para cualquier lenguaje de complejidad algo mayor, recurriremos entonces a construir un parser LR.

De manera general, el autómata SLR tiene una cantidad considerablemente menor de estados que el autómata LR correspondiente. En los casos donde la gramática es SLR(1) es preferible usar dicho autómata. Desgraciadamente, la mayoría de las construcciones sintácticas de interés para los lenguajes de programación usuales tienen gramáticas "naturales" que no son SLR, pero sí LR, y generalmente LALR. Por este motivo, el parser LR(1) es, en la práctica, el más usado. Su mayor desventaja radica en el elevado número de estados, que dificultan su almacenamiento. Este problema era especialmente complejo en el año 1965, cuando Donald Knuth propuso este parser. Por tal motivo, aunque teóricamente es una solución adecuada, en la práctica no se usó hasta que en 1969 James DeRemer propuso el parser LALR, que reduce considerablemente la cantidad de estados hasta un nivel comparable con el SLR, sin perder prácticamente en expresividad con respecto con al LR. Más adelante en 1971 el propio DeRemer propondría el SLR como una variante más sencilla de construir algo parecido al LALR de forma más sencilla.

De modo que, en una situación real, la primera decisión podría ser intentar directamente construir un parser LALR. Si esto funciona, no hay nada más que hacer. En caso contrario, deberíamos probar entonces a construir un parser LR, aunque tenga una cantidad mucho mayor de estados. En el caso peor en que esto no sea posible, tendremos que modificar la gramática. En la práctica esto no sucede comúnmente. Aunque es cierto que es fácil encontrar lenguajes didácticos que no sean LR y sean bien pequeños, los lenguajes de programación reales tienen construcciones que generalmente sí son LR. Incluso en los casos en que esto no sucede, veremos más adelante que es posible "pasar" algunos de los problemas del lenguaje para la fase semántica, y simplificar la gramática, haciéndola LR.

Desde el punto de vista teórico, tenemos una jerarquía de gramáticas que se comporta de la siguiente forma:





Es decir, las gramáticas LR son un conjunto estrictamente superior a las gramáticas LL, SLR y LALR. Entre las gramáticas SLR, LALR y LR hay una relación de inclusión que es un orden total. Sin embargo, aunque las gramáticas LL están estrictamente incluidas en las LR, existen gramáticas LL que no son ni SLR ni LALR.

Desde el punto de vista de la implementación, los parsers SLR, LALR y LR son idénticos. Solamente se diferencian en la forma en que se construye el autómata, que en última instancia determina la tabla obtenida. De la tabla en adelante el algoritmo de parsing es el mismo que hemos visto. De modo que, en una implementación concreta, es posible desacoplar el mecanismo que genera la tabla, del mecanismo que la ejecuta, y reutilizar toda la segunda parte para cualquiera de los tres tipos de parsers.

Otra cuestión interesante es qué sucede con los errores de parsing. Knuth demostró que el autómata LR reconoce los errores de parsing lo antes posible, para cualquier clase de parser determinista. Esto quiere decir que con la misma cadena (incorrecta) de entrada, ningún algoritmo de parsing podrá darse cuenta de que la cadena es errónea antes que el algoritmo LR. De hecho, los parsers LALR y SLR son más permisivos. Dada una cadena incorrecta, es posible que el parser SLR o LALR haga algunas reducciones de más, antes de darse cuenta de que la cadena es inválida. Al final ninguno de estos parsers deja pasar incorrectamente una cadena inválida, por supuesto. Pero reconocer un error lo antes posible, además de la ventaja en eficiencia, es también muy conveniente para brindar al programar un mensaje de error lo más acertado posible.

Cabe preguntarse entonces si al inventar el autómata LR hemos definitivamente terminado con el problema de parsing. Pues resulta que la respuesta teórica para esta pregunta es **sí**. En 1965 Knuth demostró que para todo lenguaje libre del contexto determinista tiene que existir una gramática LR(k). Los lenguajes libres del contexto deterministas son aquellos tales que existe un algoritmo de parsing lineal en la longitud de la cadena en caso peor. Por otro lado, toda gramática LR(k) puede ser convertida a LR(1), con la adición de nuevos no-terminales y producciones. De modo que tenemos un resultado teórico que dice: si un lenguaje puede ser parseado en tiempo lineal, entonces puede ser parseado con un parser LR(1). Aquellos lenguajes libres del contexto que no son LR(1) tienen que ser por necesidad ambiguos, o al menos es imposible diseñar un algoritmo de parsing con tiempo lineal. De cierta forma, podemos decir que hemos terminado, pues todo lenguaje "sensato" es LR(1).

Este resultado es de hecho impresionante, pero la historia no acaba ahí. Incluso aunque teóricamente LR(1) es suficiente, en la práctica hay lenguajes deterministas cuyas gramáticas LR(1) son tan complicadas, que es preferible usar una gramática más flexible, incluso incluyendo algo de ambiguedad que pueda ser resuelto en una fase superior. Afortunadamente, el problema de encontrar un árbol de derivación para cualquier gramática libre del contexto tiene solución con caso peor $O(n^3)$ con respecto a la longitud de la cadena. Es decir, existen parsers generales que pueden reconocer cualquier lenguaje libre del contexto, *incluso lenguajes ambiguos*, y en estos casos se pueden obtener **todos** los árboles de derivación que existen. En la práctica sin embargo, para diseñar lenguajes de programación, queremos

parsers lineales por motivos de eficiencia. Estos parsers más generales se emplean sobre todo en tareas de procesamiento de lenguaje natural (p.e. traducción automática).

Capítulo 4

Análisis Semántico

En la mayoría de los lenguajes de programación de tercera generación existe el concepto de variable, que en última instancia se mapea a una región de la memoria donde se almacena un valor. Una de las reglas básicas del uso de variables en casi todos los lenguajes tiene que ver con la declaración y/o inicialización de una variable antes de su uso. Por ejemplo, en los lenguajes tipo C (C++, C#, Java), tenemos la siguiente construcción:

```
int x = 5;
int y = 10;
// ....
int z = x + y;
```

Desde el punto de vista sintáctico, podemos pensar que un fragmento de la gramática que genera este lenguaje será algo como:

```
<assignment> := <type> <id> "=" <expr> ";" | ...
```

Una gramática como esta será incapaz de diferenciar situaciones como la anterior, de situaciones como la siguiente:

```
int x = 5;
int y = 10;
// ....
int z = p + q;
```

Donde las variables `p` y `q` no aparecen anteriormente en ninguna parte del método correspondiente. En la práctica es virtualmente imposible diseñar una gramática que tenga en cuenta que el identificador `p` tiene que haber sido usado en la parte derecha de una asignación antes de que aparezca en la parte izquierda.

Peor aún es el problema de determinar qué operaciones son válidas para un tipo determinado. Por ejemplo, impedir el uso del operador `+` entre una variable de tipo `int` y una de tipo `bool`. Incluso más complejo es verificar la consistencia de una invocación `x.F()`. En este

caso es necesario saber de qué tipo `T` es `x` para determinar si existe un método `F` declarado en la clase `T`, o peor aún, en algún padre de la clase `T`.

Un ejemplo aún más complicado desde el punto de vista sintáctico es validar si en una invocación `F(a,b,c)` la cantidad de parámetros es correcta, y si los tipos asociados a las expresiones `a`, `b` y `c` son compatibles a los tipos declarados en los parámetros formales de la función (iguales, herederos o existe una conversión implícita). Por ejemplo, distinguir en el fragmento de programa siguiente, que el método `G` es correcto, pero ni `H` ni `I` son correctos:

```
void F(int a, int b) {
    // ...
}

void G() {
    F(1, 2);
}

void H() {
    F(1, "2");
}

void I() {
    F(1, 2, 3);
}
```

Podemos pensar que la gramática "natural" para la declaración e invocación de funciones tiene la forma siguiente:

```
 <func-decl> := <type> <id> "(" <arg-list> ")" "{" <statement-block> "}"
  <arg-list> := <arg> | <arg> "," <arg-list> | epsilon
       <arg> := <type> <id>

<func-invok> := <id> "(" <expr-list> ")"
 <expr-list> := <expr> | <expr> "," <expr-list> | epsilon
      <expr> := ...
```

En esta gramática (o variantes similares) no existe ninguna diferencia que permita distinguir la invocación hecha en `G` de la invocación hecha en `H`. Intuitivamente no podemos expresar en una gramática libre del contexto las dependencias y relaciones entre los tipos, números de argumentos, y ámbitos de variables, dado que estas relaciones son intrínsicamente *dependientes del contexto*. Esto se debe a que dichas relaciones se ven expresadas, en principio, a todo lo largo del programa. La distancia entre una declaración de variable o función y su uso puede ser arbitrariamente larga. El orden en que las declaraciones y los usos se entrelazan es también arbritario. Por lo tanto, en principio, no deberíamos ser capaces de encontrar gramáticas libres del contexto que nos permitan expresar estas restricciones.

De forma general, el problema de reconocer si una cadena pertenece a cierto lenguaje, podemos verlo como el problema de determinar si dicha cadena cumple una serie de predicados lógicos. Por ejemplo, el lenguaje $a^n b^n$ está formado por las cadenas $\omega = s_1 s_2 \dots s_n$

que cumplen los predicados siguientes:

- $s_i = a$ o $s_i = b$ (el alfabeto es $\{a, b\}$)
- Si $j > i$ y $s_i = b$ entonces $s_j = b$ (todas las b aparecen luego de todas las a)
- $|\{s_i | s_i = a\}| = |\{s_i | s_i = b\}|$ (la cantidad de a y b es la misma)

Para muchas clases de predicados, podemos construir gramáticas que generan los lenguajes correspondientes. A veces, la intersección de estos lenguajes tiene una estructura tal que aún podemos seguir construyendo gramáticas para el lenguaje final. En otras ocasiones, la intersección de varios predicados nos da un lenguaje tal que, aunque somos capaces de reconocer las partes constituyentes, no podemos reconocer el lenguaje como un todo con gramáticas del mismo poder expresivo.

Consideremos entonces un lenguaje de programación determinado para el que queremos construir un compilador. Una de las tareas de este compilador es determinar qué cadenas (programas) son válidas en el lenguaje. Podemos pensar en la gran variedad de predicados que se aplican en estos casos. Por un lado, están todas las reglas que podemos llamar "sintácticas": los métodos empiezan por un identificador y una lista de argumentos entre paréntesis, las instrucciones terminan en un símbolo "punto y coma (;)", etc. Por otro lado, tenemos todas estas reglas que no son sintácticas: la consistencia en el uso de los tipos, que cierta función debe devolver un valor por todos los posibles caminos de ejecución, que las variables deben inicializarse antes de usarse en expresiones. Podemos llamar a estas reglas, "semánticas", porque de cierta forma nos indican cuál es el significado "real" del lenguaje.

Por ejemplo, en la instrucción `int x = 5;` tenemos por un lado el conjunto de reglas que determinan la forma de la instrucción (primero el tipo, luego un identificador, luego un igual, luego una expresión), y las que determinan el significado de la instrucción (almacenar un valor `5` en la zona de memoria asociada a la variable `x`). Podemos entonces imaginar que dividimos todos los predicados que definen los programas válidos en dos conjuntos: aquellos para los que podemos construir una gramática que los reconozca, y los que no. Los primeros serán justamente los *predicados sintácticos*, y los segundos, por analogía, los llamaremos *predicados semánticos*. Para el primer conjunto, tenemos un mecanismo formal que nos permite describirlos: las **gramáticas libres del contexto**. Para el segundo conjunto, desarrollaremos en este capítulo otro mecanismo formal similar, que nos permitirá describir de forma unívoca cuál es el "significado" de un programa.

Justamente, la fase de análisis sintáctico se encarga de validar que los predicados sintácticos se cumplen (y además construir un árbol de derivación). La fase de **análisis semántico**, que comenzaremos a estudiar en este capítulo, se encarga entonces de validar que los predicados semánticos se cumplan (y a su vez construir otras estructuras de datos que veremos más adelante).

Hemos visto que en la fase semántica el poder de las gramáticas libres del contexto es insuficiente. Además, los ejemplos de problemas semánticos que hemos presentado nos muestran la extrema dificultad de expresar estas reglas, incluso con gramáticas dependientes del contexto (para las que además no tenemos mecanismos reconocedores eficientes). Tenemos entonces que cambiar de paradigma.

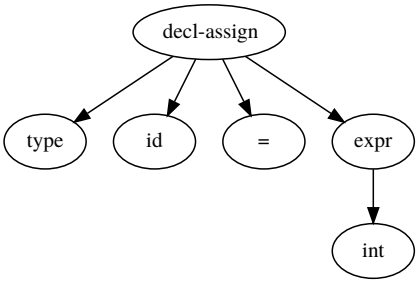
Recordemos el problema a resolver: verificar los predicados semánticos (que aun no hemos definido completamente), una vez tenemos la seguridad de que la sintaxis es correcta.

Recordemos entonces los árboles de derivación, que representan en una estructura computacionalmente cómoda de manipular el conjunto de producciones y el orden en que son aplicadas para producir la oración correspondiente. Este árbol de derivación a menudo se denomina árbol de sintaxis concreta, pues representa exactamente todos los elementos de la sintaxis descritos en la gramática. Intuitivamente, este árbol de derivación contiene todo el contexto del programa, en una estructura conveniente para ser explorada. ¿Y si intentáramos resolver los predicados semánticos, justamente viéndolos como predicados sobre el árbol de derivación, en vez de sobre la cadena de entrada? ¿Dado que tenemos toda cadena representa en una forma que nos expone toda la estructura sintáctica, no debería ser más fácil detectar aquí todos estos predicados que pueden, en principio, depender de *toda* la estructura global de la cadena?

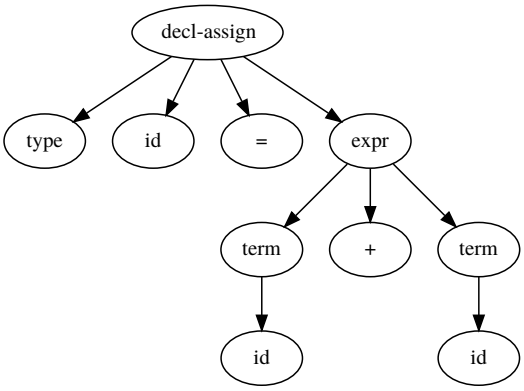
Veamos como podemos reescribir algunos de estos predicados en forma de predicados sobre el árbol. Por ejemplo, la regla de que toda variable debe haber sido declarada antes de usarse. Pensemos en un programa arbitrario de C, por ejemplo:

```
int x = 5;
//...
int y = x + 1;
```

Pudiéramos pensar en un posible árbol de derivación para este programa (para una gramática correcta del lenguaje C), donde habría necesariamente un subárbol con la forma:



Y por otro lado, tendríamos un subárbol con la forma:



Y ambos subárboles serían hijos de algún nodo que representa a la función correspondiente. Entonces podemos pensar en una estrategia para determinar si todas las variables son declaradas antes de su uso. En un recorrido en pre-orden por dicho árbol, podemos ir recolectando todas las declaraciones hechas en cierta estructura de datos, y luego cada vez que nos encontremos una variable referenciada en parte izquierda de una expresión, ¡simplemente consultamos dicha estructura y verificamos la existencia de una variable con dicho nombre, y de paso la consistencia de los tipos! De forma similar podemos pensar en la verificación de la cantidad de argumentos usados en una función y sus tipos.

Árboles de Sintaxis Abstracta

En general, una vez tenemos el árbol de derivación, tenemos suficiente información sobre la cadena para reconocer cuáles son todas las variables, métodos, invocaciones, los tipos de cada expresión, etc. El árbol de derivación justamente nos asocia cada token de la cadena a su **función sintáctica**. Nos dice, por ejemplo, que el identificador `x` es el nombre de una variable, mientras que el identificador `printf` pudiera ser el nombre de una función. Además, nos dice cuando este identificador `x` aparece por primera vez, y mejor aún, cuándo aparece en parte derecha o en parte izquierda de una expresión. Tenemos entonces todo el poder expresivo de nuestros algoritmos y estructuras de datos (recorridos de árboles, diccionarios y tablas hash, etc.) para diseñar mecanismos de validación semántica. ¡Justamente es por este motivo que construimos el árbol de derivación en primer lugar!

Sin embargo, antes de que lanzarnos a diseñar algoritmos de validación semántica, revisemos nuestro árbol de derivación, y notaremos que su estructura no es exactamente idónea para esta tarea. Tomemos por ejemplo la siguiente gramática, que genera un lenguaje de expresiones aritméticas simples:

```
E = T + E | T
T = int * T | int | (E)
```

Y la cadena siguiente:

```
2 * (3 + 5)
```

Como sabemos, esta cadena realmente como secuencia de tokens es:

```
int * ( int + int )
```

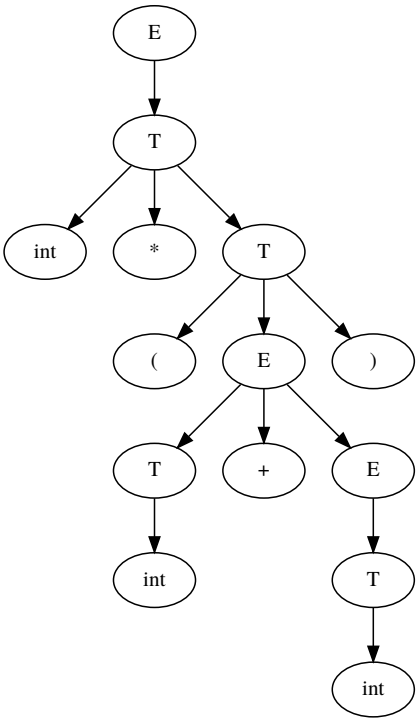
De momento no nos preocuparemos por el valor concreto del número almacenado en cada token. Como sabemos del análisis lexicográfico, el lexer asocia a cada token además de la clase correspondiente, el fragmento de cadena original que lo forma. Cuando sea conveniente, podemos ver la cadena anterior como:

```
int{2} * ( int{3} + int{5} )
```

Donde indicamos explícitamente el **lexema**. Durante el análisis sintáctico hemos obviado este detalle pues solo nos interesaban los símbolos que aparecen en la gramática. Más

adelante volveremos a incluir en nuestro análisis el valor concreto de cada token, que en última instancia constituyen los datos del programa a compilar.

Por el momento, concentrémonos nuevamente en la cadena a parsear. Esta cadena es generada **de forma única** por el siguiente árbol de derivación:



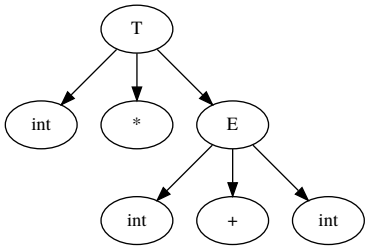
Este árbol de derivación efectivamente codifica todas las operaciones necesarias a realizar para **evaluar** la expresión (que es en última instancia el problema a resolver). Sin embargo, este árbol representa con demasiado detalle la expresión. Supongamos que queremos diseñar una jerarquía de clases para representar este árbol. Dicha jerarquía tendría varias clases innecesarias, como aquellas que representan a los nodos `(` y `)`. Por otro lado, la estructura del árbol es poco eficiente para representar la expresión, pues hay varios nodos que son redundantes. Por ejemplo, el nodo raíz `E` no nos da ninguna información sobre el tipo concreto de la expresión. De forma general podemos reconocer dos tipos de elementos innecesarios:

- Nodos que representan elementos sintácticos innecesarios (e.j. los paréntesis)
- Nodos que derivan en un solo hijo (e.j. `E -> T`)

Los elementos sintácticos innecesarios, como los paréntesis, se emplean en la gramática para representar la prioridad entre sub-expresiones. Sin embargo, una vez construído el árbol de derivación, la prioridad entre las sub-expresiones queda explícitamente descrita en la propia estructura del árbol. Por otro lado, los nodos que derivan en un solo nodo hijo, tales como `E -> T`, son necesarios desde el punto de vista de la gramática para resolver las

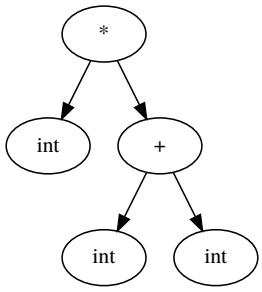
ambigüedades, pero una vez que se construye el árbol de derivación, no aportan ninguna información adicional.

Intentemos eliminar estos elementos innecesarios en el árbol de derivación anterior:



Este árbol representa exactamente la misma expresión aritmética, y quedan explícitamente descritos el orden y el tipo de las operaciones. Una vez llegado a este punto, podemos notar que hay otro elemento innecesario en el árbol. Si nos fijamos con atención, veremos que el nodo asociado a un operador (`*` o `+`) siempre estará como hijo de exactamente el mismo tipo de nodo (`T` o `E`) respectivamente. Este hecho se desprende directamente de la gramática, pues el terminal `*` solo se genera por `T` y el terminal `+` solo se genera por `E`. Por tanto, ambos nodos respectivos (`T` y `*` o `E` y `+`) siempre aparecerán juntos, y por tanto es redundante tener ambos.

Pensemos ahora en la jerarquía de clases que representa este árbol, y un posible algoritmo recursivo de evaluación. Una vez en el nodo `T`, evaluados recursivamente las expresiones izquierda y derecha, ¿qué operación es necesario aplicar? Evidentemente, dependerá del nodo que representa la operación. Pero este nodo (`*` o `+`) siempre es un terminal, por lo tanto, nunca será necesario "bajar" recursivamente para descubrir que tipo de operación hay que hacer en un nodo `T` o `E`. Podemos entonces conformarnos con un árbol donde la operación a realizar esté explícita en el nodo padre (`T` o `E`) y no como un hijo adicional:



Intuitivamente, el árbol anterior es capaz de representar con todo el detalle necesario la semántica de la expresión a evaluar, y no tiene ningún elemento innecesario.

A este tipo de estructura se le denomina **árbol de sintaxis abstracta (AST)**, precisamente porque representa solamente la porción de sintaxis necesaria para evaluar la expresión o programa reconocido. En el AST solamente existen nodos por cada tipo de elemento

semántico diferente, es decir, por cada significado distinto.

Por ejemplo, en nuestro lenguaje de expresiones aritméticas, existen tres tipos de "entidades" o conceptos diferentes:

- un número,
- una operación de suma, y
- una operación de multiplicación.

De modo que si el árbol de sintaxis concreta (o árbol de derivación) tiene un tipo de nodo por cada tipo de **función sintáctica** diferente, entonces un árbol de sintaxis abstracta tiene un tipo de nodo por cada tipo de **función semántica** distina. Las diferentes funciones sintácticas se obtienen directamente de la gramática, y por tanto están muy influenciadas por el tipo de *parser* que se use. Las funciones semánticas, por otro lado, se diseñan a partir de la funcionalidad que se quiere obtener en el lenguaje, por lo que se tiene generalmente mayor libertad creativa.

Diseño de un AST

A modo de ejemplo, vamos a definir un lenguaje muy sencillo, para el que diseñaremos un árbol de sintaxis abstracta. Este lenguaje será sobre el dominio de las expresiones aritméticas. Nos permitirá operar con variables y funciones predefinidas, así como definir nuevas funciones. Comenzaremos por listar de manera informal las características que queremos obtener de este lenguaje, y veremos luego como formalizar cada una.

Veamos primero las reglas **sintácticas**:

- El lenguaje tiene tres tipos de instrucciones: `let`, `def` y `print`:
  - `let <var> = <expr>` define una variable denominada `<var>` y le asigna el valor de `<expr>`
  - `def <func>(<arg1>, <arg2>, ...) -> <expr>` define una nueva función `<func>` con los argumentos `<arg*>`
  - `print <expr>` imprime el valor de una expresión
- Las expresiones pueden ser de varios tipos:
  - Expresiones ariméticas
  - Invocación de funciones predefinidas (`sin`, `cos`, `pow`, ...)
  - Invocación de funciones definidas en el programa

Formalizar estas características sintácticas es relativamente fácil con las herramientas que ya tenemos. Simplemente definiremos una gramática para ello:

```
   <program> := <stat-list>
 <stat-list> := <stat> ";"
              | <stat> ";" <stat-list>
      <stat> := <let-var>
              | <def-func>
              | <print-stat>
  <let-var> := "let" ID "=" <expr>
 <def-func> := "def" ID "(" <arg-list> ")" "->" <expr>
```

```
<print-stat> := "print" <expr>
  <arg-list> := ID
              | ID "," <arg-list>
      <expr> := <term> + <expr>
              | <term> - <expr>
              | <term>
      <term> := <factor> * <term>
              | <factor> / <term>
              | <factor>
    <factor> := <atom>
              | "(" <expr> ")"
      <atom> := NUMBER
              | ID
              | <func-call>
 <func-call> := ID "(" <expr-list> ")"
 <expr-list> := <expr>
              | <expr> "," <expr-list>
```

La gramática anterior es bastante "natural", en el sentido de que no contiene reglas "extrañas" para resolver, por ejemplo, los problemas de ambigüedad típicos de las gramáticas LL(1). Este es el tipo de gramáticas que usualmente queremos diseñar para transmitir a otros lectores las reglas sintácticas del lenguaje. Luego, para una implementación concreta de un parser, es posible (y altamente probable), que tengamos que redefinir la gramática añadiendo producciones para convertirla en LL o LR, según el caso. En cualquier modo, como ya sabemos resolver ese problema, nos conformaremos con pensar que alguien nos dará un parser para esta gramática.

Definamos ahora informalmente las reglas **semánticas** de nuestro lenguaje:

- Una variable solo puede ser definida una vez en todo el programa.
- Los nombres de variables y funciones no comparten el mismo ámbito (pueden existir una variable y una función llamadas igual).
- No se pueden redefinir las funciones predefinidas.
- Una función puede tener distintas definiciones siempre que tengan distinta cantidad de argumentos (es decir, funciones del mismo nombre pero con cantidad de argumentos distintos se consideran funciones distintas).
- Toda variable y función tiene que haber sido definida antes de ser usada en una expresión (salvo las funciones pre-definidas).
- Todos los argumentos definidos en una misma función tienen que ser diferentes entre sí, aunque pueden ser iguales a variables definidas globalmente o a argumentos definidos en otras funciones.
- En el cuerpo de una función, los nombres de los argumentos ocultan los nombres de variables iguales.

  Nota: Una consecuencia interesante de estas reglas es que no permiten funciones recursivas...

Como vemos, todas estas reglas son de naturaleza dependiente del contexto, pues su alcance puede ser en principio todo el programa. Por lo tanto, no tenemos forma de expresar estas

reglas en una gramática libre del contexto. Sin embargo, una vez tengamos el árbol de sintaxis abstracta de un programa concreto, veremos que es relativamente fácil verificar cada una de estas reglas.

Para diseñar el AST de este lenguaje, pensemos en las diferentes funciones semánticas que tiene nuestro programa. Veremos que en muchos casos tenemos un paralelo directo entre un nodo del AST y un símbolo de la gramática, pero en otros casos existirán símbolos que no tienen una función semántica (como los terminales `let`, `def` y `print`) y existirán funciones semánticas (clases de nodos del AST) que no corresponden a ningún símbolo de la gramática.

Comencemos entonces por el inicio. De forma general es convieniente diseñar una jerarquía de clases con una raíz `Node` (o de nombre similar) que nos representa cualquier nodo del AST. Agrupar todos los nodos del AST en una jerarquía común nos permite definir funciones abstractas sobre todos los nodos (que usaremos para el chequeo semántico más adelante). Por tanto, comenzamos por ahí:

```
public abstract class Node {
    //...
}
```

A diferencia del árbol de derivación, en esta clase `Node` no hemos puesto explícitamente una propiedad `Children` ni nada similar. Esto se debe a que en el AST en principio cada hijo de un nodo tiene una función semántica propia. Por tanto, en vez de tener una lista de hijos general, es preferible en cada subclase defnir exactamente que nodos (y de qué tipo) son los hijos esperados. De modo que no tendremos un "árbol" en el sentido puro como estructura de datos, sino una colección de clases relacionadas entre sí, cuya estructura en memoria será un árbol.

La primera función semántica que necesitamos es aquella que nos represente a nuestro programa. Como hemos visto en la gramática, un programa simplemente es una colección de instrucciones (*statements*), por lo tanto, necesitamos también definir esta función semántica.

```
public class Program {
    public List<Statement> Statements;
}

public abstract class Statement {
    //...
}
```

Notar que hemos definido a `Program` como una clase concreta, pero a `Statement` como a una clase abstracta. Esto se debe a que tenemos diversos tipos de instrucciones concretas, pero la instrucción "base" en sí no es realmente una función semántica propia. Una regla más o menos común es que cuando un no-terminal deriva en más de una producción, cada una de esas producciones puede corresponder a una función semántica distinta. Sin embargo, en otros casos, un no-terminal deriva en más de una producción por motivos puramente sintácticos, como en el caso de `<stat-list>`, cuya única función es describir una lista de instrucciones. En el AST, no tiene mucho sentido tener un nodo particular para una lista de instrucciones, y por tanto en `Program` tenemos esta lista de forma explícita.

Veamos entonces los tres tipos concretos de instrucciones:

```
public class LetVar : Statement {
    public string Identifier;
    public Expression Expr;
}

public class DefFunc : Statement {
    public string Identifier;
    public List<string> Arguments;
    public Expression Expr;
}

public Print : Statement {
    public Expression Expr;
}
```

El motivo por el que estos nodos son herederos de `Statement` es justamente para poder definir en `Program` una lista de diferentes tipos de instrucciones. De modo que estamos la herencia solamente para definir estructura, y no para aprovecharnos del polimorfismo (ya que no tenemos comportamiento en los nodos). Esto cambiará más adelante cuando nos interesemos por la verificación semántica.

Vamos a definir la parte de la jerarquía que representa las expresiones en sí. En la gramática tenemos los no-terminales `<expr>`, `<term>`, `<factor>` y `<atom>` que nos sirven para definir la prioridad de los operadores. Sin embargo, desde el punto de vista semántico, todos estos no-terminales juegan el mismo papel. Por ejemplo, las expresiones binarias, desde el punto de vista semántico, son prácticamente idénticas. La única diferencia es el operador concreto a aplicar. De modo que podemos tener **una única clase** de expresión binaria:

```
public enum Operator { Add, Sub, Mult, Div }

public abstract class Expression : Node {
    //...
}

public class BinaryExpr : Expression {
    public Operator Op;
    public Expression Left;
    public Expression Right;
}
```

Este diseño puede parecer a algunos "anti-orientado-a-objetos", pues no estamos definiendo una clase por cada tipo de expresión binaria. La discusión de cuál diseño es mejor es mucho más compleja que lo que podemos abordar en este párrafo. Solamente diremos algo a favor de este enfoque, y es que, al menos de momento, si tuviéramos una clase `MultNode` y una clase `AddNode`, estas serían exactamente iguales. Por lo tanto, como no tenemos comportamiento **ni** estructura distintos en ambas clases, no tiene sentido, al menos de momento, tener una clase por cada tipo de operador. Más adelante revisaremos esta

hipótesis y tendremos una discusión más profunda al respecto.

Nos quedan entonces tres tipos de expresiones: constante numérica, variable y llamada a una función. Para cada una de estas tendremos una clase distinta, pues tienen estructura diferente:

```
public class FuncCall : Expression {
    public string Identifier;
    public List<Expression> Args;
}

public class Variable : Expression {
    public string Identifier;
}

public class Number : Expression {
    public string Value;
}
```

Hemos definido la clase `Number` con un valor de tipo `string`, ya que, en principio, no es hasta el chequeo semántico que nos interesa saber el valor concreto del número que este token representa. Por lo tanto, podemos pensar que ni el *lexer* ni el *parser* se interesarán por obtener este valor.

Y en este punto hemos terminado con nuestro árbol de sintaxis abstracta. Tenemos un total de 11 clases, de las cuáles 3 son abstractas y 8 son concretas. En la gramática tenemos un total de 25 producciones, que en principio son 25 funciones sintácticas diferentes. Algunas de ellas son listas, y por tanto no tienen función semántica asociada. Otras, como en el caso de las expresiones, están para desambigüar y establecer el orden operacional, por lo que tampoco tienen una función semántica asociada. En lenguajes más complejos, podemos tener también varias producciones que simplemente sean formas sintácticas diferentes de expresar la misma función semántica (e.j. **if** con **else** e **if** sin **else**).

Nos queda pendiente la tarea de obtener el AST a partir del árbol de derivación (que es la salida que realmente nos da el *parser*). De momento vamos a asumir que este algoritmo existe, y nos concentraremos en los tipos de análisis que podemos hacer sobre el AST una vez construido. Más adelante veremos un mecanismo formal que nos permite expresar como se construye el AST a partir del árbol de derivación, y un algoritmo para ello.

Validando las reglas semánticas

Volvamos a la tarea que dio origen a toda esta discusión del AST: validar el cumplimiento de las reglas semánticas. Recordemos nuevamente estas reglas:

- Una variable solo puede ser definida una vez en todo el programa.
- Los nombres de variables y funciones no comparten el mismo ámbito (pueden existir una variable y una función llamadas igual).
- No se pueden redefinir las funciones predefinidas.

- Una función puede tener distintas definiciones siempre que tengan distinta cantidad de argumentos (es decir, funciones del mismo nombre pero con cantidad de argumentos distintos se consideran funciones distintas).
- Toda variable y función tiene que haber sido definida antes de ser usada en una expresión (salvo las funciones pre-definidas).
- Todos los argumentos definidos en una misma función tienen que ser diferentes entre sí, aunque pueden ser iguales a variables definidas globalmente o a argumentos definidos en otras funciones.
- En el cuerpo de una función, los nombres de los argumentos ocultan los nombres de variables iguales.

En esta sección vamos a presentar una solución *ad-hoc* para estos problemas, en el caso particular del lenguaje que hemos definido. Más adelante formalizaremos esta estrategia de solución y mostraremos como extenderla a lenguajes más generales.

De manera general, el enfoque que usaremos será el siguiente. Cada regla semántica realmente se aplica solo a algunos tipos de nodos, pero para cada tipo de nodo podemos determinar qué reglas se aplican. Vamos a definir entonces un método `Validate()` en cada nodo, que nos dirá si dicho nodo es correcto. Por supuesto, ya podemos intuir que la implementación concreta de este método en un nodo dependerá recursivamente de los nodos hijos.

Por otro lado, notemos que la mayoría de las reglas semánticas nos hablan sobre las definiciones y uso de las variables y funciones. De hecho, por este motivo es justamente que dichas reglas son dependientes del contexto. De modo que, intuitivamente, en cada nodo necesitaremos tener acceso a este "contexto", donde pueden estar definidas las funciones y variables que su usan en dicho nodo. Dado que tenemos todo el AST construido, cabe pensar que dicho contexto se puede calcular en un recorrido sobre el árbol. De hecho, como veremos, en el mismo recorrido que vamos validando las reglas semánticas, iremos construyendo el contexto correspondiente. Vamos entonces a diseñar una estructura de datos que nos represente dicho contexto. En esta estructura tenemos que almacenar las funciones y variables definidas (en el caso de las funciones con la cantidad de argumentos), y poder consultar en todo momento qué es lo definido. Supongamos entonces que tenemos una implementación de la siguiente **interface**:

```
public interface IContext {
    bool IsDefined(string variable);
    bool IsDefined(string function, int args);
    bool Define(string variable);
    bool Define(string function, string[] args);
}
```

Agregamos entonces el método siguiente:

```
public abstract class Node {
    public abstract bool Validate(IContext context);
}
```

Veamos entonces como se implementa esta validación en los nodos concretos. En el caso de `Program` es muy sencillo, el programa está correcto solo si cada una de las instrucciones que lo componen se valida correctamente:

```
public class Program : Node {
    public List<Statement> Statements;

    public override bool Validate(IContext context) {
        foreach(var st in Statements) {
            if (!st.Validate(context)) {
                return false;
            }

            return true;
        }
    }
}
```

Pasemos entonces a la validación de las expresiones, cuyas regla semántica fundamental es que todos los identificadores usados tienen que haber sido definidos antes. En el caso de las expresiones binarias es muy sencillo:

```
public class BinaryExpr : Expression {
    public Operator Op;
    public Expression Left;
    public Expression Right;

    public override bool Validate(IContext context) {
        return Left.Validate(context) && Right.Validate(context);
    }
}
```

Ahora, en el caso de las expresiones atómicas es donde realmente sucede lo importante. Comezamos por el caso más sencillo:

```
public class Number : Expression {
    public string Value;

    public override bool Validate(IContext context) {
        return true;
    }
}
```

Veamos entonces el caso del nodo `Variable`. Este nodo nos representa una variable siendo usada como parte de una expresión, por lo tanto, como dice la regla semántica, tiene que haber sido definida antes:

```
public class Variable : Expression {
    public string Identifier;

    public override bool Validate(IContext context) {
        return context.IsDefined(Identifier);
    }
```

```
}
```

Por último, para el caso de una función, la validación es muy similar, pero teniendo en cuenta la cantidad de argumentos. Además, dado que una invocación a función define recursivamente un conjunto de expresiones para los valores de cada argumento, es necesario validar cada una:

```
public class FuncCall : Expression {
    public string Identifier;
    public List<Expression> Args;

    public override bool Validate(IContext context) {
        foreach(var expr in Args) {
            if (!expr.Validate(context)) {
                return false;
            }
        }

        return context.IsDefined(Identifier, Args.Count);
    }
}
```

Veamos finalmente la implementación de los nodos `LetVar` y `DefFunc`. Es justamente en estos nodos donde se definen nuevas variables y funciones, por lo que en el momento de validar estos nodos, es donde realmente se calcula este "contexto" del que tanto hemos hablado. El caso de la definición de variable es el más sencillo, así que comenzaremos por ahí:

```
public class LetVar : Statement {
    public string Identifier;
    public Expression Expr;

    public override bool Validate(IContext context) {
        if (!Expr.Validate(context)) {
            return false;
        }

        if (!context.Define(Identifier)) {
            return false;
        }

        return true;
    }
}
```

El caso de la definición de función es mucho más interesante. Por un lado es necesario validar que no exista una función con el mismo nombre y la misma cantidad de argumentos, lo que es sencillo. Por otro lado, es necesario validar recursivamente el cuerpo de la función. El problema aquí es que en el cuerpo de la función permitimos no solo usar las variables

definidas globalmente, sino además *los argumentos* de la propia función. Es decir, durante la validación del cuerpo de la función, existen unas variables "especiales", que son justamente los argumentos, cuyos nombres sí pueden coincidir con los nombres de variables definidas anteriormente, pero no entre sí.

Una solución rápida a este problema consiste en simplemente definir los argumentos cuyos identificadores sean nuevos (es decir, que no coincidan con una variable global), y al final de la validación, "des-definir" estos argumentos. Por un lado, esto implica adicionar un mecanismo para "des-definir", que es una operación que realmente no tiene un significado real en nuestra semántica. Además, es necesario llevar la cuenta de cuáles fueron los argumentos que se adicionaron al contexto para quitar solamente esos. Pero más importante que todos esos motivos, es el hecho de aunque un argumento tiene el mismo nombre de una variable en realidad es una variable distinta. Como ahora no estamos almacenando en el contexto nada adicional asociado a las variables, realmente esto no importa. Pero en el momento en que para cada variable querramos almacenar algo adicional, tenemos que hacer esta distinción. Por ejemplo, en algún momento vamos a querer ejecutar nuestro programa, y tendremos que asociar a cada variable y argumento un valor numérico concreto.

Una solución mucho más elegante y extensible es introduciendo un nuevo concepto que llamaremos **ámbito** (*scope* en inglés). De manera general, un ámbito es una región de un programa donde están definidos ciertos símbolos (variables, funciones, etc.). Definimos entonces que entre ámbitos distintos puede haber coincidencias de nombres de símbolos, pero dentro del mismo ámbito no. Por ejemplo, podemos tener un ámbito distinto para cada función (básicamente una instancia de `IContext` distinta), y de esta forma nunca tendremos que preocuparnos por la colisión de nombres entre argumentos en diferentes funciones.

Ahora, el otro problema a resolver, es que en el cuerpo de una función sí pueden aparecer referencias a variables definidas fuera de la función. De modo que nos sirve simplemente crear un nuevo `IContext` vacío y definir todos los argumentos ahí. Necesitamos que este `IContext` pueda resolver no solo lo que tiene definido explícitamente, sino todo aquello definido "afuera". Decimos entonces que este nuevo ámbito es "hijo" del ámbito global, de modo que puede sobre-escribir algunos símbolos, pero sigue teniendo acceso a todo lo definido anteriormente.

En el caso actual solamente vamos a tener un ámbito global, y luego un ámbito por cada función; pero en lenguajes más complejos, es posible definir tantos ámbitos anidados como se desee (por ejemplo, en C# cada instrucción **for** crea un nuevo ámbito más interno). De modo que en general podemos pensar en los ámbitos también como una estructura árborea, donde existe un ámbito global, y luego se van creando ámbitos "hijo" según sea necesario. Este patrón de diseño es muy común en la mayoría de los lenguajes de programación existentes, por lo que vale la pena presentarlo aquí. Desde el punto de vista de diseño, simplemente necesitamos adicionar un método a nuestra **interface**:

```
public interface IContext {
    bool IsDefined(string variable);
    bool IsDefined(string function, int args);
    bool Define(string variable);
    bool Define(string function, string[] args);
```

```
    IContext CreateChildContext(); // <- esto es lo nuevo
}
```

Sin embargo, desde el punto de implementación, ahora los métodos `IsDefined` deben modificarse para buscar no solo en el diccionario de símbolos del ámbito actual, sino recursivamente en el ámbito padre. Aunque esta implementación es sencilla, no es del todo trivial, por lo que vamos a mostrar una posible solución:

```
public class Context : IContext {
    IContext parent;
    HashSet<string> variables = new ...
    Dictionary<string, string[]> functions = new ...

    bool IsDefined(string variable) {
        return variables.Contains(variables) ||
               (parent != null && parent.IsDefined(variable));
    }

    bool IsDefined(string function, int args) {
        if (functions.ContainsKey(function) &&
            functions[function].Length == args) {
            return true;
        }

        return parent != null && parent.IsDefined(function, args);
    }

    bool Define(string variable) {
        return variables.Add(variable);
    }

    bool Define(string function, string[] args) {
        if (functions.ContainsKey(function) &&
            functions[function].Length == args) {
            return false;
        }

        functions[function] = args;
        return true;
    }

    IContext CreateChildContext() {
        return new Context() { parent = this };
    }
}
```

Nuestra implementación de contexto nos garantiza que no es posible definir la misma variable o la misma función (con igual cantidad de argumentos) en un mismo contexto,

pero sí nos permite sobreescribir los símbolos existentes en el contexto padre. Varios autores llaman a esta estructura de datos **tabla de símbolos**, ya que almacena todos los símbolos definidos en el programa. Nosotros le llamamos **contexto**, pues consideramos que es un nombre más general que da la idea de que en esta estructura se almacena la información dependiente del contexto que es útil para cada elemento semántico del programa.

Haciendo uso de esta estructura, podemos entonces finalmente implementar la validación del nodo `DefFunc`:

```
public class DefFunc : Expression {
    public string Identifier;
    public List<string> Args;
    public Expression Body;

    public override bool Validate(IContext context) {
        var innerContext = context.CreateChildContext();

        foreach(var arg in Args) {
            innerContext.Define(arg);
        }

        if (!Body.Validate(innerContext)) {
            return false;
        }

        if (!context.Define(Identifier, Args.ToArray())) {
            return false;
        }

        return true;
    }
}
```

En este punto, ya tenemos toda la semántica de nuestro lenguaje validada, y solamente quedaría el problema de realmente ejecutar las instrucciones existentes. Siguiendo el mismo enfoque visto hasta ahora, podemos pensar en una solución que vaya recursivamente evaluando cada expresión, y almacenando en una especie de "contexto de ejecución" los valores de las expresiones definidas hasta el momento. Dado que no es posible redefinir variables, es totalmente válido calcular el valor de una variable tan pronto como se define, y almacenarlo. Para el caso de las funciones, lo más conveniente es almacenar directamente la expresión que se refiere al cuerpo de la función, para poder ejecutarla cuando sea necesario. En ese caso, un problema interesante de resolver es qué los argumentos de la función se definen en el nodo `DefFunc`, pero los valores que tendrán realmente solo se conocen en algún nodo `FuncCall`. Por lo tanto, en cada invocación, es necesario poder acceder al nodo `DefFunc` correspondiente para poder asignar los valores concretos que en esa invocación serán usados. Dejamos como sugerencia intentar implementar este mecanismo de evaluación.

Capítulo 5

Gramáticas Atributadas

Hasta el momento hemos visto como obtener un árbol de derivación de un lenguaje, y luego hemos presentado el árbol de sintaxis abstracta como una descripción más cómoda para realizar análisis semántico. Sin embargo, todavía no tenemos un mecanismo para construir un árbol de sintaxis abstracta a partir de un árbol de derivación. En principio, podemos pensar en estrategias *ad-hoc*, y para cada gramática particular escribir un algoritmo que construye el AST. De forma general estos algoritmos serán estrategias recursivas que irán recorriendo el árbol de derivación y computando fragmentos del AST a partir de los fragmentos obtenidos en los nodos hijos. En este capítulo veremos un esquema formal para describir esta clase de algoritmos, que además nos permitirá resolver varios problemas dependientes del contexto de forma elegante y sencilla.

Construyendo un AST

Empecemos por mostrar, de forma intuitiva, como construir un AST para un árbol de derivación de una gramática conocida. Usaremos la gramática de expresiones aritméticas tan gastada:

```
E -> E + T
   | E - T
   | T

T -> T * F
   | T / F
   | F

F -> ( E )
   | i
```

Una posible jerarquía para el AST de esta gramática es la siguiente:

```
public enum Op { Add, Sub, Mult, Div }

public abstract class Expression : {

}

public class BinaryExpr : Expression {
    public Op Operator;
    public Expression Left;
    public Expression Right
}

public class Number : Expression {
    public float Value;
}
```

Supongamos entonces que tenemos un árbol de derivación para una cadena particular. Recordemos que un árbol de derivación es una estructura donde en cada nodo hay un símbolo, y los hijos de dicho nodo coinciden exactamente con los símbolos de la parte derecha de la producción aplicada. Una posible definición de un árbol de derivación para esta gramática particular sería:

```
public enum Symbol { E, T, add, sub, mult, div, i, left, right }

public class Node {
    public Symbol Symbol;
    public List<Node> Children = new ...
}
```

Supongamos entonces que tenemos un algoritmo de parsing que nos devuelve este árbol de derivación (LR en este caso, dado la gramática que hemos definido). Queremos entonces adicionar un método `GetAST()` en la clase `Node` que nos devuelve un objeto de tipo `Expression`.

```
public class Node {
    // ...

    public Expression GetAST() {
        // ...
    }
}
```

De forma general hay 3 patrones distintos en la gramática que nos generan nodos semánticamente distintos en el árbol de derivación:

- las producciones que derivan en una expresión binaria;
- las producciones que derivan en un solo no-terminal, incluyendo `F -> ( E )`; y
- el símbolo `i` que es el único terminal con una función semántica asociada.

Cada uno de estos tipos de producciones genera un nodo particular del AST. Por lo tanto,

nuestro código consiste en identificar en qué caso nos encontrarmos, y construir el nodo correspondiente.

Si la producción es de la forma $X \to YoZ$, donde X, Y y Z son símbolos y o es un operador, creamos un nuevo nodo `BinaryExpr` con el operador adecuado, y los hijos se convierten recursivamente en las expresiones izquierda y derecha:

```
public Expression GetAST() {
    // X -> Y o Z
    if (Children.Count == 3 && Children[1].Symbol >= 2) {
        return new BinaryExpr() {
            Left  = Children[0].GetAST(),
            Op    = GetOperator(Children[1]),
            Right = Children[2].GetAST()
        };
    }
}
```

Si la producción es de la forma $X \to Y$, simplemente devolvemos la expresión hija. Notemos que en este caso el resultado que se produce es una "compactación" del árbol de derivación, eliminando las producciones de un solo símbolo:

```
public Expression GetAST() {
    // ...
    // X -> Y
    else if (Children.Count == 1) {
        return Children[0].GetAST();
    }
}
```

Si la producción es particularmente `F -> ( E )` también devolvemos la expresión hija:

```
public Expression GetAST() {
    // ...
    // F -> ( E )
    else if (Children.Count == 3 && Children[1].Symbol == Symbol.E) {
        return Children[1].GetAST();
    }
}
```

Finalmente, cuando el nodo actual del árbol de derivación es una hoja, tiene que ser un nodo `i` y se crea la expresión unaria `Number`:

```
public Expression GetAST() {
    // ...
    // i
    else if (Children.Count == 0 && Symbol == Symbol.i) {
        return new Number() { Value = /* valor del token */ };
    }
```

```
    throw new InvalidParseTreeException("xP");
}
```

Reglas y atributos semánticos

Veamos nuevamente la implementación que hemos hecho para nuestra gramática de expresiones. De forma general, lo que hemos hecho ha sido intentar identificar qué producción concreta está representada en cada nodo, y según el caso, aplicar una regla que nos permite construir el fragmento de AST. Vamos a intentar generalizar este concepto de asociar reglas a producciones. Para ello, adicionaremos a cada símbolo de la gramática un conjunto de atributos, y a cada producción un conjunto de reglas que describen cómo se computan dichos atributos. Llamaremos a estas gramáticas: **gramáticas atributadas**:

Una **gramática atributada** es una tupla $< G, A, R >$, donde:

- $G =< S, P, N, T >$ es una gramática libre del contexto,
- A es un conjunto de atributos de la forma $X \cdot a$ donde $X \in N \cup T$ y a es un identificador único entre todos los atributos del mismo símbolo, y
- R es un conjunto de reglas de la forma $< p_i, r_i >$ donde $p_i \in P$ es una producción $X \to Y_1, \ldots, Y_n$, y r_i es una regla de la forma:
  1. $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$, o
  2. $Y_i \cdot a = f(X \cdot a_0, Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$.

  En el primer caso decimos que a es un **atributo sintetizado**, y en el segundo caso, un **atributo heredado**.

Hablemos ahora sobre la notación. Aunque en la definición formal hemos especificado los atributos y reglas como elementos adicionales a la gramática, desde el punto de vista notacional es conveniente especificar las reglas y los atributos directamente asociados a la producción que corresponden:

```
X -> YZ { X.a = f(Y.a, Z.a) }
```

De este modo, tenemos una notación compacta, donde se puede reconocer a simple vista la gramática, los atributos, y las reglas. Por convenio llamamos a estas reglas, **reglas semánticas**, pues nos permiten definir de cierta forma la *función semántica* de cada producción. Consideraremos que los terminales tienen un conjunto de atributos cuyos valores son suministrados por el *lexer*. Cuando tenemos más de un símbolo con el mismo nombre, pondremos índices, por ejemplo:

$X \to cX\{X_0 \cdot a = f(X_1 \cdot a_1, c \cdot a_2)\}$

En realidad es necesario definir algunas restricciones adicionales sobre las reglas para que una gramática atributada sea consistente. En particular, necesitamos que no existan definiciones contradictorias de los atributos (es decir, no haya más de una regla que defina el mismo atributo en la misma producción, y que todo atributo sea o bien sintetizado, o bien heredado), y que todos los atributos estén bien definidos en cualquier árbol de derivación. A este último problema nos dedicaremos más adelante cuando nos preocupemos por la evaluación de los atributos. De momento simplemente digamos que necesitamos definir las reglas de forma que sean consistentes y completas.

En esta definición no hemos especificado de qué naturaleza son los atributos o las reglas. De forma general, asumimos que los atributos son de tipos que sean convenientes computacionalmente: numéricos, conjuntos, diccionarios, listas, o incluso tipos complejos en un entorno orientado a objetos. Las reglas las consideraremos en general como funciones computables en algún sistema de cómputo. Cuando solo nos interese definir formalmente un lenguaje, emplearemos una notación matemática y funciones puras (sin efectos colaterales).

Veamos entonces cómo redefinir la gramática de expresiones vista anteriormente para computar el AST. Usaremos la misma notación para definir árboles que hemos visto en la sección de *parsing* LR: un nodo de tipo T cuyos hijos son los árboles $t_1, \ldots, t_n$ lo representaremos como $T(t_1, \ldots, t_n)$. Es decir, representaremos un árbol según el **recorrido en pre-orden** de sus nodos. Particularmente en esta gramática, usaremos las siguiente notación:

- Un nodo terminal de tipo `Number` lo representaremos mediante el símbolo `n`.
- Un nodo no-terminal de tipo `BinaryExpr` lo representaremos según el tipo del operador, como `exp(op, e1, e2)`

Definiremos un atributo de nombre `ast` en cada símbolo, que almacenará el árbol de sintaxis abstracta. La gramática nos quedaría entonces:

```
E -> E + T { E0.ast = exp(+, E1.ast, T.ast) }
   | E - T { E0.ast = exp(-, E1.ast, T.ast) }
   | T     { E0.ast = T.ast }

T -> T * F { T0.ast = exp(*, T1.ast, F.ast) }
   | T / F { T0.ast = exp(/, T1.ast, F.ast) }
   | F     { T0.ast = F.ast }

F -> ( E ) {  F.ast = E.ast }
   | i     {  F.ast = n }
```

Puede parecer que no hemos logrado mucho, más allá de formalizar en una notación una idea que ya sabíamos manejar. Y de alguna manera es cierto, lo que hemos hecho ha sido simplemente formalizar en una notación la idea intuitiva de cómo construir a partir de un árbol de derivación una representación más conveniente. La ventaja de tener esta notación formalizada, además de permitirnos razonar y comunicarnos al respecto, es que hemos simplificado considerablemente la cantidad de "código" a escribir para construir el AST. Hemos quitado del medio toda la sintaxis superflua de definición de métodos, parámetros, variables temporales, etc, y solamente hemos puesto en cada producción exactamente la "línea de código" que iría dentro del `if` correspondiente. De hecho, esta notación es tan conveniente, que la mayoría de los generadores de *parsers* usan una sintaxis similar para describir justamente cómo se construye el AST, y generan todo el engranaje de métodos recursivos, variables, paso de parámetros y demás que son necesarios para hacer funcionar este mecanismo.

Resolviendo dependencias del contexto

En nuestra definición de gramática atributada no hemos dicho nada del AST. Aunque en la práctica usamos los atributos la mayoría de las veces para construir el AST, y luego resolver los problemas dependientes del contexto sobre el AST, en ocasiones es conveniente resolver algunos de estos problemas directamente empleando el mecanismo de gramática atributada. Esto tiene sentido sobre todo si estamos construyendo lenguajes pequeños, con pocas reglas semánticas y no queremos diseñar una jerarquía de AST independiente.

Por ejemplo, tomemos el lenguaje dependiente del contexto canónico $L = a^n b^n c^n$, y veamos como podemos describir mediante una gramática atributada los predicados semánticos que definen este lenguaje. Comenzamos por la gramática:

```
S -> ABC
A -> aA | epsilon
B -> bB | epsilon
C -> cC | epsilon
```

Como convenio, vamos a definir un atributos `S.ok` cuyo valor será `true` si y solo si la cadena reconocida pertenece al lenguaje. En cada no terminal vamos a definir un atributo `cnt` que almacenará la cantidad de veces que este no terminal ha derivado en el terminal correspondiente. De modo que en cada producción de `A`, `B` o `C`, iremos "contando" la cantidad de `a`, `b` o `c` que se van produciendo, y luego en `S` determinaremos si la cadena es correcta. Vamos a usar para la definición de las reglas una notación más parecida a un lenguaje de programación convencional:

```
S -> ABC       {   S.ok = (A.cnt == B.cnt && B.cnt == C.cnt) }
A -> aA        { A0.cnt = 1 + A1.cnt }
   | epsilon   {  A.cnt = 0 }
B -> bB        { B0.cnt = 1 + B1.cnt }
   | epsilon   {  B.cnt = 0 }
C -> cC        { C0.cnt = 1 + C1.cnt }
   | epsilon   {  C.cnt = 0 }
```

En este punto hemos definido una gramática con atributos que reconoce todas las cadenas de la forma $a^* b^* c^*$, pero para las cuales el valor del atributo `S.ok` es `true` solamente en las cadenas exactamente de la forma $a^n b^n c^n$. La forma de resolverlo ha sido definir una gramática libre del contexto, y hemos adicionado la dependencia del contexto en forma de reglas semánticas. Si observamos la gramática definida, notaremos que nos hemos conformado con reconocer solamente la parte regular del lenguaje, y hemos dejado todo el resto del problema a las reglas semánticas. Un enfoque alternativo consiste en intentar resolver en el análisis sintáctico tanto como sea posible.

Por ejemplo, podemos reconocer $a^n b^n c^*$, en la gramática, y luego solamente verificar la correspondencia entre la cantidad de `b` y la cantidad de `c`. Comencemos por definir una gramática libre del contexto para esto:

```
S -> XC
X -> aXb | epsilon
C -> cC  | epsilon
```

Almacenaremos entonces en `X.cnt` la cantidad de `b` que aparecen, y luego lo compararemos con `C.cnt`:

```
S -> XC       {   S.ok = (X.cnt == B.cnt) }
X -> aXb      { X0.cnt = 1 + X1.cnt }
   | epsilon  {  X.cnt = 0 }
C -> cC       { C0.cnt = 1 + C1.cnt }
   | epsilon  {  C.cnt = 0 }
```

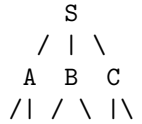
La pregunta es por supuesto, entre estas dos alternativas, cuál es mejor. Y como casi siempre la respuesta será que depende de qué queremos lograr. Por un lado, una gramática más simple es más fácil de leer y entender, y (en ocasiones) más fácil de parsear. En ese caso la mayor carga queda en la fase semántica, que generalmente es más compleja y tiene un costo computacional mayor. Por otro lado, si la gramática es más compleja, pero captura una mayor cantidad de propiedades del lenguaje, es posible reconocer una mayor cantidad de errores en la fase sintáctica, y probablemente se obtenga un compilador más eficiente. Sin embargo, estas gramáticas son más complejas, y por tanto más difíciles de entender.

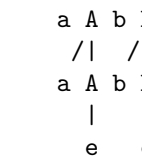
Otro factor a tener en cuenta, desde el punto de vista de la ingeniería de software, es la *mantenibilidad* del compilador. En general, es más difícil hacer cambios en las primeras fases, pues es más probable que un cambio en la gramática o en el *lexer* provoquen una disrupción en las fases siguientes. En cambio, por la propia naturaleza de la programación orientada a objetos, con un buen diseño es posible lograr que los cambios en la fase semántica sean lo menos disruptores posibles. Por este motivo, es conveniente definir bien temprano una gramática lo más sencilla posible que englobe todas las propiedades sintácticas del lenguaje en cuestión, y dejar para la fase semántica toda propiedad cuyo significado o implementación tenga una alta probabilidad de variar.

Computando el valor de los atributos

Una vez definida una gramática atributada, nos queda pendiente el problema de cómo evaluar las reglas semánticas para asignar el valor correcto a los atributos. Como es de esperar, esta evaluación la vamos a realizar una vez construido el árbol de derivación de una cadena particular. De forma general, podemos tener en cualquier nodo tanto atributos sintetizados como atributos heredados. Necesitamos entonces encontrar un orden para evaluar los atributos que garantice que siempre sea posible evaluar las reglas correspondientes. Para esto, necesitamos saber las *dependencias* de cada atributo en una regla particular, es decir, cuales son los otros atributos que es necesario haber evaluado antes. Intuitivamente, estas dependencias son justamente los atributos usados dentro de la función f en la regla semántica.

Tomemos entonces a modo de ejemplo la cadena `aabbcc` y veamos un árbol de derivación para la gramática atributada definida anteriormente:

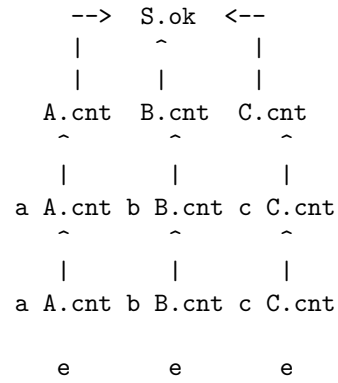




Vamos entonces a construir sobre este árbol de derivación un *grafo de dependencia* de los atributos asociados. Este grafo se construye de la siguiente forma:

- Los nodos son los atributos de cada símbolo en cada nodo del árbol de derivación
- Existe una arista dirigida entre un par de nodos $v \to w$ del grafo de dependencia, si los atributos correspondientes participan en una regla semántica definida en la producción asociada al nodo del árbol de derivación donde aparece w, de la forma $w = f(\ldots, v, \ldots)$. Es decir, si el atributo *w depende* del atributo v.

Veamos entonces como quedaría el árbol anterior, una vez señaladas las dependencias entre los atributos. Para simplificar la notación, vamos a representar solamente las aristas del grafo de dependencias, y no las del árbol de derivación en sí.:



La propiedad más interesante de este grafo de dependencias es que nos dice, en primer lugar, si existe una forma de evaluar los atributos, y en caso de ser posible, nos da un orden para la evaluación. Si existe algún ciclo en este grafo de dependencias, evidentemente no podremos evaluar los atributos que participan en el ciclo. Por lo tanto, decimos que la gramática es evaluable si y solo si el grafo de dependencias es acíclico, es decir, un DAG. En este caso, la evaluación se realiza siguiendo algún *orden topológico* del grafo de dependencia.

En el caso anterior, nuestro grafo de dependencias es justamente un árbol, donde todas las aristas están orientadas de un hijo a un padre en el AST. Esto sucede cada vez que en una gramática todos los atributos son *sintetizados*. Evidentemente, en este caso es posible evaluar siempre los atributos de un nodo padre, una vez evaluados todos los atributos de los hijos. A este tipo de gramática les llamaremos **gramáticas s-atributadas** (la *s* viene de *sintetizado*). Este tipo de gramáticas son las más sencillas de evaluar, y también las más comunes. En general cuando estamos definiendo el AST nos saldrán casi siempre atributos sintetizados, pues como lo que queremos construir es un árbol, es natural que definamos los atributos en este orden. La buena noticia con las gramáticas s-atributadas, es que siempre es posible evaluar los atributos (en todo árbol de derivación), y esto siempre será con un recorrido en

post-orden. Formalmente:

> Una gramática atributada es **s-atributada** si y solo si, para toda regla r_i asociada a una producción $X \to Y_1, \ldots, Y_n$, se cumple que r_i es de la forma $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$.

Para evaluar los atributos en una gramática s-atributada, podemos seguir una estrategia como la siguiente. Asumiremos que durante la construcción del árbol de derivación, el *parser* introduce en cada nodo una lista de todas las reglas semánticas asociadas a dicha producción, y que los atributos están definidos en cada símbolo.

```
public class Node {
    public Symbol Symbol;
    public List<Node> Children;
    public List<Rule> Rules;

    public void EvaluateSynthesized() {
        foreach(var node in Children)
            node.EvaluateSynthesized();

        foreach(var rule in Rules)
            rule.evaluate(Symbol, Children);
    }
}
```

Veamos entonces otro caso un poco más general, donde también podemos decir de antemano que la gramática es evaluable. Este es el caso cuando todo atributo es, o bien sintetizado, o si es heredado, solo depende de atributos de símbolos que le anteceden en la producción. Es decir, si tenemos una regla $Y_i \cdot a_i = f(X \cdot a, Y_1 \cdot a_1, \ldots, Y_{i-1} \cdot a_{i-1})$. A estas gramáticas las llamaremos **gramáticas l-atributadas** (donde *l* viene de *left*), ya que todo atributo depende solo de atributos que se evalúan antes en el pre-orden. En este caso, también podemos siempre evaluar la gramática haciendo un recorrido en profundidad, aunque es un poco más complicado que para el caso anterior. En cada llamado recursivo, primero evaluaremos los atributos heredados, pues solo dependen de atributos en los hermanos anteriores y el padre. Luego descendemos recursivamente en cada hijo, y al retornar, evaluamos los atributos sintetizados en el nodo actual. Formalizando:

> Una gramática atributada es **l-atributada** si y solo si toda regla r_i asociada a una producción $X \to Y_1, \ldots, Y_n$ es de una de las siguientes formas: 1. $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$, ó 2. $Y_i \cdot a_i = f(X \cdot a, Y_1 \cdot a_1, \ldots, Y_{i-1} \cdot a_{i-1})$.

Y un ejemplo de cómo evaluar los atributos, suponiendo que el *parser* es capaz de suministrar en par de listas separadas aquellas reglas que sintetizan atributos de las que los heredan. Supongamos que cada nodo recibe además una referencia al nodo padre en el árbol de derivación (para poder acceder a los hermanos). En el nodo raíz simplemente pasaremos `null` como padre:

```
public class Node {
    public Symbol Symbol;
    public List<Node> Children;
```

```
    public List<Rule> SyntethicRules;
    public List<Rule> InheritedRules;

    private void EvaluateInherited(Node parent) {
        foreach(var rule in InheritedRules)
            rule.evaluate(Symbol, parent, parent.Children);

        foreach(var node in Children)
            node.EvaluateInherited(this);

        foreach(var rule in SyntheticRules)
            rule.evaluate(Symbol, Children);
    }
}
```

Evaluando atributos durante el proceso de *parsing* descendente

Cabe preguntarnos entonces sino es posible evaluar las reglas semánticas a medida que se realiza el proceso de parsing. Intuitivamente, al menos para las gramáticas s-atributadas y l-atributadas esto debe ser posible, pues existe un orden claro de dependencia entre los atributos. Por supuesto, esto dependerá también del algoritmo de *parsing* empleado.

Por ejemplo, para gramáticas LL, si estamos empleando un algoritmo recursivo descendente, el orden en que se construye el árbol de derivación es exactamente el mismo que el orden de evaluación de una gramática l-atributada. Tomemos como ejemplo la gramática LL(1) de expresiones aritméticas que hemos visto anteriormente:

```
E -> T X
T -> i Y | ( E )
X -> + E | - E | epsilon
Y -> * T | / T | epsilon
```

Como hemos visto anteriormente, los árboles de derivación que salen de esta gramática son bastante complicados. Vamos a intentar escribir un conjunto de reglas semánticas que nos calculen directamente el valor de la expresión durante el proceso de parsing. El problema radica en que cuando tenemos la producción `E -> T X`, no sabemos en este punto si la operación a realizar es una suma o una resta. Es solo cuando vemos la producción en que derivó `X` que podemos decidir que operación hacer. Sin embargo, en la producción `X -> + E`, por ejemplo, ya no tenemos el valor de `T` disponible para hacer el cálculo. De modo que tenemos que "pasar" el valor de `T` a `X` en la producción `E -> T X`, para que luego `X` pueda decidir que hacer con este valor.

Vamos a definir entonces 3 atributos diferentes, de tipo numérico. En primer lugar, el atributo `i.val`, suministrado por el *lexer*, que contiene el valor numérico del token. Tendremos además los atributos `E.sval`, `T.sval`, `X.sval` y `Y.sval`, que son respectivamente los valores de las expresiones aritméticas representadas en cada caso. Los hemos llamado con el prefijo `s`

justamente porque estos atributos serán *sintetizados*. Finalmente, vamos a tener los atributos `X.hval` y `Y.hval` que contendrán los valores de las expresiones a la izquierda de `X` o `Y` y son atributos *heredados*.

Una vez definidos estos atributos, veamos algunas ideas sobre cómo computarlos. Algunos casos bases son fáciles, por ejemplo para `T -> ( E )` simplemente tenemos que "subir" el valor de `E.sval` a `T.sval`. Por otro lado, tomemos por ejemplo la producción `E -> T X`. Sabemos que en `T` tenemos un atributo sintetizado con el valor numérico de lo que sea que haya sido reconocido como `T` por ese lado. Sin embargo, aún no sabemos si sumarlo o restarlo con lo que resta. La estrategia en este caso será "pasar" este valor computado hasta el momento a `X` (por medio del atributo `X.hval`), y luego pedirle a `X` que compute el valor `X.sval` como la suma (o resta) del valor heredado de `T` y lo que sea que `X` en sí haya parseado. Es decir, primero "bajaremos" el valor de `T` hacia `X`, sumando o restando según el caso, hasta llegar a las hojas (donde `X -> epsilon`). En este punto, lo que sea que tengamos acumulado lo "subiremos" hacia la raíz (por medio de `X.sval`).

Veamos entonces toda la lista de reglas semánticas:

```
E -> T X      { X.hval = T.sval, E.sval = X.sval }
T -> i Y      { Y.hval = i.val,  T.sval = Y.sval }
   | ( E )    { T.sval = E.sval }
X -> + E      { X.sval = X.hval + E.sval }
   | - E      { X.sval = X.hval - E.sval }
   | epsilon  { X.sval = X.hval }
Y -> * T      { Y.sval = Y.hval * T.sval }
   | / T      { Y.sval = Y.hval / T.sval }
   | epsilon  { Y.sval = Y.hval }
```

Intentemos ahora parsear una cadena, a la vez que vamos evaluando los atributos. Vamos a representar por un lado la cadena que vamos parseando, y por otro lado la pila de "llamados" recursivos que se va formando con cada llamado del *parser* recursivo descedente asociado a esta gramática. En cada "llamado" recursivo reprsentaremos también el valor de cada atributo en la forma `S[attr=val]` (un `.` significa que el valor aún no ha sido calculado). Comenzamos por la cadena completa, y el llamado al símbolo `E`. Una advertencia para los débiles de espíritu, cualquiera que haya almorzado recientemente debería saltar al siguiente capítulo...

```
|i{2} * ( i{5} - i{3} )

 E[sval=.]
```

En el primer paso, como `E` solo deriva en `T X`, no queda otra que llamar a `T`. Vamos a representar también el llamado a `X` para no olvidar que al retornar `T` debemos descender recursivamente por `X`.

```
|i{2} * ( i{5} - i{3} )

 T[sval=.] , X[sval=.,hval=.]
 E[sval=.]
```

Ahora, mirando el token `i` y los *First* de cada producción, aplicamos `T -> i Y`:

```
|i{2} * ( i{5} - i{3} )

 i[val=.] , Y[sval=.,hval=.]
 T[sval=.] , X[sval=.,hval=.]
 E[sval=.]
```

Podemos entonces reconocer `i` y asociar el valor adecuado al atributo:

```
 i{2}|* ( i{5} - i{3} )

 i[val=2] , Y[sval=.,hval=.]
 T[sval=.] , X[sval=.,hval=.]
 E[sval=.]
```

Al retornar de `i` a `T`, como tenemos ya en `i` el valor asociado al token, aplicamos la regla semántica correspondiente a `Y`:

```
 i{2}|* ( i{5} - i{3} )

 Y[sval=.,hval=2]
 T[sval=.] , X[sval=.,hval=.]
 E[sval=.]
```

Procedemos entonces expandir el símbolo `Y`:

```
 i{2}|* ( i{5} - i{3} )

 *[] , T[sval=.]
 Y[sval=.,hval=2]
 T[sval=.] , X[sval=.,hval=.]
 E[sval=.]
```

En este punto solo es necesario reconocer el token `*`, y luego descendemos recursivamente por `T`, pero esta vez por la producción `( E )`:

```
 i{2} *|( i{5} - i{3} )

 ([], E[sval=.], )[]
 T[sval=.]
 Y[sval=.,hval=2]
 T[sval=.] , X[sval=.,hval=.]
 E[sval=.]
```

Avanzamos el siguiente token:

```
 i{2} * (|i{5} - i{3} )

 E[sval=.], )[]
 T[sval=.]
 Y[sval=.,hval=2]
```

```
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y ahora volvemos a descender recursivamente:

```
i{2} * (|i{5} - i{3} )

T[sval=.] , X[sval=.,hval=.]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y luego:

```
i{2} * (|i{5} - i{3} )

i[val=.] , Y[sval=.,hval=.]
T[sval=.] , X[sval=.,hval=.]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Reconocemos `i`:

```
i{2} * ( i{5}|- i{3} )

i[val=5] , Y[sval=.,hval=.]
T[sval=.] , X[sval=.,hval=.]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y retornamos recursivamente poniendo el valor de `i` en el atributo `hval` de Y:

```
i{2} * ( i{5}|- i{3} )

Y[sval=.,hval=5]
T[sval=.] , X[sval=.,hval=.]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Dado que - está en el $Follow(Y)$, aplicamos `Y -> epsilon`, y aquí veremos la magia de las gramáticas atributadas actuando, pues el valor de `hval` pasa a `sval` justo antes de retornar:

```
i{2} * ( i{5}|- i{3} )

Y[sval=5,hval=5]
T[sval=.] , X[sval=.,hval=.]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y ahora al retornar aplicamos la regla semántica en `X` que "sube" el valor de `Y.sval`:

```
i{2} * ( i{5}|- i{3} )

T[sval=5] , X[sval=.,hval=.]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Una vez parseado `T`, antes de retornar, pasamos su valor a `hval` de `X`:

```
i{2} * ( i{5}|- i{3} )

X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Expandimos entonces la producción `X -> - E`:

```
i{2} * ( i{5}|- i{3} )

-[], E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Avanzamos el siguiente token y expandimos:

```
i{2} * ( i{5} -|i{3} )
```

```
T[sval=.], X[sval=.,hval=.]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Nuevamente, expandimos `T -> i Y` (ya esto se está volviendo un poquito repetitivo...):

```
i{2} * ( i{5} -|i{3} )

i[val=.], Y[sval=.,hval=.]
T[sval=.], X[sval=.,hval=.]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y de nuevo hacemos la secuencia de `match` y paso de valor de `i` para `Y`:

```
i{2} * ( i{5} - i{3}|)

Y[sval=.,hval=3]
T[sval=.], X[sval=.,hval=.]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Ahora vendrá una cascada de evaluaciones y reducciones a ϵ que disfrutaremos en cámara lenta. Primero, pasamos el valor `Y.hval` para `Y.sval`:

```
i{2} * ( i{5} - i{3}|)

Y[sval=3,hval=3]
T[sval=.], X[sval=.,hval=.]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
```

```
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Al retornar `Y`, sintetizamos el valor de `T.sval`:

```
i{2} * ( i{5} - i{3}|)

T[sval=3], X[sval=.,hval=.]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y lo pasamos para `X.hval`:

```
i{2} * ( i{5} - i{3}|)

X[sval=.,hval=3]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Como `)` está en el $Follow(X)$, solo queda pasar el valor `X.hval` para `X.sval`:

```
i{2} * ( i{5} - i{3}|)

X[sval=3,hval=3]
E[sval=.]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y al retornar felizmente este valor va para `E.sval`:

```
i{2} * ( i{5} - i{3}|)

E[sval=3]
X[sval=.,hval=5]
E[sval=.], )[]
T[sval=.]
```

```
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Ahora se está poniendo interesante la pila. En el tope tenemos a `E[sval=3]` que debe retornar para `X`. Recordemos que esta producción era justamente `X -> - E`, por lo que al retornar `E` se aplica la regla semántica que computa, por primera vez, la operación resta! LoL! Siguiendo esta regla, tenemos que hacer `X.sval` igual a `X.hval - E.sval`.

```
i{2} * ( i{5} - i{3}|)

X[sval=2,hval=5]
E[sval=.], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Retornamos entonces "subiendo" el valor de `X.sval`:

```
i{2} * ( i{5} - i{3}|)

E[sval=2], )[]
T[sval=.]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y ahora al retornar `E`, justo antes de llamar a `match`, "subimos" a `T` el valor correspondiente:

```
i{2} * ( i{5} - i{3}|)

)[]
T[sval=2]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Luego avanzamos al último token que nos queda de la cadena de entrada:

```
i{2} * ( i{5} - i{3} )|

T[sval=2]
Y[sval=.,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

Y ahora vuelve a producirse otro paso mágico, pues `Y` al retornar `T` realiza la tan esperada operación de multiplicación!

```
i{2} * ( i{5} - i{3} )|
```

```
Y[sval=4,hval=2]
T[sval=.] , X[sval=.,hval=.]
E[sval=.]
```

El retorno de `Y` es recibido de buena gana por `T`:

```
i{2} * ( i{5} - i{3} )|

T[sval=4] , X[sval=.,hval=.]
E[sval=.]
```

Que estaba esperando solo para pasarle este valor a `X`:

```
i{2} * ( i{5} - i{3} )|

X[sval=.,hval=4]
E[sval=.]
```

Ahora `X`, sin nada más que hacer, debe expandirse en ϵ, pero antes, computará el valor de su atributo `X.sval`:

```
i{2} * ( i{5} - i{3} )|

X[sval=4,hval=4]
E[sval=.]
```

Y lo devuelve con gracia a `E`:

```
i{2} * ( i{5} - i{3} )|

E[sval=4]
```

De modo que hemos terminado de parsear la cadena, y no solo hemos reconocido su bien merecida pertenencia al lenguaje, sino que además nos la hemos ingeniado para obtener, a la misma vez, el valor ya computado de la expresión aritmética! xD.

Si parece que hemos tenido que hacer malabares para calcular el valor de la expresión, intentemos pensar en el código equivalente que realiza esta misma evaluación sobre el árbol de derivación ya creado. Hemos definido un mecanismo formal para expresar el cómputo de atributos sobre un árbol de derivación, que nos ha permitido expresar fácilmente la evaluación de la expresión aritmética, sin tener que lidiar con todos los detalles de implementación, paso de parámetros, etc. De hecho, este mecanismo es tan formal, que lo hemos ejecutado de forma mecánica, sin pararnos a razonar, solamente siguiendo a ciegas las reglas semánticas, ¡y ha funcionado!

Evaluando atributos durante el proceso de *parsing* ascendente

Antes de terminar, para aquellos que no quedaron satisfechos con la sección anterior, pues tenemos más… Vamos a realizar un proceso similar, pero esta vez sobre una gramática LR. En este tipo de gramáticas, tenemos que pagar un precio a la hora de definir atributos, pues solamente podemos evaluarlos durante el proceso de parsing si todos los atributos son sintetizados. Intuitivamente, esto es cierto ya que en la pila de símbolos nunca tendremos a un "padre" antes que a un "hijo", debido a la naturaleza *bottom-up* del proceso de *parsing*. Por tanto, los atributos del símbolo "padre" solamente pueden depender de los atributos de sus "hijos", y nunca de su propio "padre". Por otro lado, como se construye el árbol de derivación extrema derecha, tampoco pueden depender de los atributos de los hermanos.

El proceso de evaluación general funcionará de la siguiente manera. Cada vez que se introduce un símbolo en la pila, computaremos el valor de todos sus atributos, a partir de los símbolos hijo que acabamos de sacar de la pila. Los tokens entran a la pila con sus atributos ya evaluados por el *lexer*.

Con esto en mente, vamos a definir entonces la gramática LR de expresiones aritméticas con un único atributo sintetizado `val` en cada nodo. En el proceso de *parsing* ascendente, iremos computando los valores correspondientes a partir de los valores de los hijos. Las reglas semánticas que nos quedan son bastante intuitivas, y no requieren de mayor explicación:

```
E -> E + T  { E0.val = E1.val + T.val }
   | E - T  { E0.val = E1.val - T.val }
   | T      { E0.val = T.val }
T -> T * F  { T0.val = T1.val * F.val }
   | T / F  { T0.val = T1.val / F.val }
   | F      { T0.val = F.val }
F -> i      {  F.val = i.val }
   | ( E )  {  F.val = E.val }
```

Vamos entonces a representar para esta gramática en la pila de símbolos además los valores de los atributos, y la cadena de entrada como siempre:

```
|i{2} * ( i{5} - i{3} )
|
```

No vamos a mostrar aquí la tabla LR, ya que para esta gramática es bastante intuitivo que operación realizar en cada caso. Por tanto nos ahorraremos especificar exactamente en qué estado nos encontramos, y simplemente asumiremos que automáta LR funciona. Comenzamos entonces por introducir `i` en la pila, y calculamos el valor correspondiente (que viene en el token):

```
    i{2}|* ( i{5} - i{3} )
i[val=2]|
```

En este punto podemos reducir `F -> i`, y calcular el valor correspondiente:

```
    i{2}|* ( i{5} - i{3} )
```

```
F[val=2]|
```

Seguimos teniendo un *handle*, así que volvemos a reducir:

```
    i{2}|* ( i{5} - i{3} )
T[val=2]|
```

El siguiente paso es un **shift** (pues no hay * en el $Follow(E)$):

```
    i{2} *|( i{5} - i{3} )
T[val=2] *|
```

Y el siguiente:

```
    i{2} * (|i{5} - i{3} )
T[val=2] * (|
```

Y el siguiente:

```
        i{2} * ( i{5}|- i{3} )
T[val=2] * ( i[val=5]|
```

Podemos entonces reducir nuevamente (subiendo el valor del atributo):

```
        i{2} * ( i{5}|- i{3} )
T[val=2] * ( F[val=5]|
```

Y de nuevo:

```
        i{2} * ( i{5}|- i{3} )
T[val=2] * ( T[val=5]|
```

Y de nuevo:

```
        i{2} * ( i{5}|- i{3} )
T[val=2] * ( E[val=5]|
```

Seguimos entrando en la pila:

```
        i{2} * ( i{5} -|i{3} )
T[val=2] * ( E[val=5] -|
```

Y el siguiente:

```
            i{2} * ( i{5} - i{3}|)
T[val=2] * ( E[val=5] - i[val=3]|
```

Y ahora vamos a reducir a `F -> i` de nuevo:

```
            i{2} * ( i{5} - i{3}|)
T[val=2] * ( E[val=5] - F[val=3]|
```

Y a `T -> F`:

```
            i{2} * ( i{5} - i{3}|)
T[val=2] * ( E[val=5] - T[val=3]|
```

Y ahora podemos reducir `E -> E - T` y aplicar la regla semántica que computa la resta:

```
 i{2} * ( i{5} - i{3}|)
T[val=2] * ( E[val=2]|
```

Luego no nos queda otra opción que hacer **shift** por última vez:

```
 i{2} * ( i{5} - i{3} )|
T[val=2] * ( E[val=2] )|
```

Y ahora, como es de esperar, reducimos `F -> ( E )` quedándonos con el valor almacenado:

```
i{2} * ( i{5} - i{3} )|
   T[val=2] * F[val=2]|
```

Luego reducimos `T -> T * F` computando el valor correspondiente:

```
i{2} * ( i{5} - i{3} )|
              T[val=4]|
```

Y por último `E -> T`, dejando en la pila el símbolo inicial, y teniendo computado el valor asociado al atributo `val`:

```
i{2} * ( i{5} - i{3} )|
              E[val=4]|
```

Como hemos podido ver, el proceso de *parsing* LR es bastante más sencillo (una vez computada la tabla, claro), y además más poderoso en términos de las gramáticas que puede reconocer que el *parsing* LL. Sin embargo, desde el punto de vista de los atributos, el *parsing* LR nos obliga a definir todos nuestros atributos de forma sintetizada, lo que reduce el poder expresivo de las gramáticas atributadas. En la práctica, sin embargo, este es el tipo de *parser* más utilizado, y las reglas semánticas que queremos expresar son lo suficientemente amables como para contentarnos con gramáticas s-atributadas.

El proceso de *parsing* completo

Hemos visto entonces los conceptos fundamentales que nos permiten llegar desde una cadena, hasta un árbol de sintaxis abstracta. De forma general, el proceso completo es el siguiente:

- Definimos una gramática libre del contexto que capture las propiedades sintácticas del lenguaje, de la forma más "natural" posible (con la menor cantidad de producciones "superfluas").
- Diseñamos un árbol de sintaxis abstracta con los tipos de nodos que representan exactamente las funciones semánticas de nuestro lenguaje.
- Definimos las reglas semánticas que construyen el árbol de sintaxis abstracta, preferiblemente quedando una gramática s-atributada.
- Construimos un *lexer* a partir de las expresiones regulares que definen a los tokens.
- Construimos un *parser*, idealmente LALR (o LR si no es posible) que además de reconocer la cadena, nos evalúe y construya el AST durante el proceso de *parsing*.
- Implementamos los predicados semánticos restantes sobre el AST construido.

Este proceso está tan bien estudiado, que la mayoría de los generadores de *parser* existentes automatizan toda esta parte. A partir de una gramática libre del contexto con reglas semánticas, y una jerarquía de nodos del AST, estos generadores de *parsers* son capaces de construir el autómata LR y devolvernos directamente el AST instanciado. Contar con herramientas de este tipo nos permite iterar muy rápidamente sobre el lenguaje, modificando la gramática o el AST de forma independiente, ya que el único punto de acoplamiento son las reglas semánticas. Por este motivo es preferible tener gramáticas más sencillas, y dejar para la fase semántica la mayor cantidad de problemas.

De modo que prácticamente toda la complejidad de diseñar un lenguaje radica en definir correctamente las funciones semánticas, e implementar la fase de chequeo semántico. De hecho, en los últimos años, casi toda la investigación sobre *parsing* se ha movido a lenguajes ambigüos o lenguaje natural, y la investigación en temas de compilación puros se ha concentrado en implementar funciones semánticas más complejas. En los capítulos siguientes nos concentraremos en problemas típicos del análisis semántico, y veremos estructuras de datos y estrategias para su implementación.

Capítulo 6

Semántica de Tipos

En la mayoría de los lenguajes de programación modernos existe el concepto de "tipo". De manera informal, diremos que un tipo es una definición de especifica cuáles operaciones son válidas a realizar sobre un objeto particular. Un objeto en este caso puede ser un valor simple (`int` o `bool` en los lenguajes tipo C), o un objeto compuesto en algún lenguaje orientado a objetos. El paradigma **orientado a objetos** ha venido a convertirse en los últimos años en una de las columnas fundamentales del diseño y la investigación de nuevos lenguajes de programación. En este paradigma, cada "valor" que se puede manipulado en un programa es un *objeto*, y los objetos pueden agruparse para definir objetos más complejos. A cada objeto se le asocia un *tipo*, que define las operaciones válidas a realizar sobre dicho objeto.

La implementación más usual del concepto de tipo es una **clase**. Una clase (en C, C++, C#, Java, Python, Ruby, y tantos otros lenguajes orientados a objetos) es fundamentalmente una definición de las operaciones disponibles para un tipo. En general las clases permiten definir *atributos* que almacenan un valor, y *métodos* (o *funciones*) que permiten realizar una serie de operaciones (con o sin efectos colaterales) sobre el tipo en cuestión y los *argumentos* del método. A todos estos lenguajes los llamados *lenguajes tipados*, porque manejan el concepto de tipo. De forma general, si la operación `f(x)` (o `x.f()`) es válida en algunos contextos, e inválida en otros, aunque en ambos casos `f` y `x` son símbolos definidos, entonces diremos que dicho lenguaje es tipado, pues la validez de una operación no solo depende de que estén definidos los símbolos que participan, sino de **cómo** están definidos dichos símbolos. Un ejemplo de lenguaje no tipado es el lenguaje para expresiones que definimos en la sección *Diseño de un AST*.

Una forma usual de clasificar a los lenguajes tipados es la distinción entre tipado *dinámico* y *estático*. En ambos casos cada expresión, variable y método tiene asociado un tipo que define las operaciones válidas. La diferencia fundamental radica en que en los lenguajes con tipado estático, además existe una *declaración explícita* del tipo que deseamos para una expresión, variable, método, etc. Esto lo hacemos con la esperanza de poder capturar en la fase de chequeo semántico la mayor cantidad de errores asociados a inconsistencias de tipos posibles. En los lenguajes con tipado dinámico las inconsistencias de tipos no pasan desapercibidas, simplemente se espera hasta la ejecución para detectarlas.

En general, la discusión entre si es preferible el tipado estático o dinámico es futil. En muchas ocasiones, es conveniente tener lo antes posible una validación de que la expresión que queremos compilar no tendrá inconsistencias de tipos, y por este motivo surgieron los lenguajes con tipado estático. Por otro lado, es inevitable que existan circunstancias en las que el compilador será incapaz de inferir exactamente el tipo real que tendrá una expresión y nos impedirá realizar alguna operación cuando en realidad dicha operación sería posible. Por ejemplo, si tenemos la siguiente declaración de clases en C#:

```
class A {
    public void F() { /* ... */ }
}

class B : A {
    public void G() { /* ... */ }
}
```

El siguiente fragmento código da error de compilación pues el tipo declarado para la variable `a` es `A`, donde no está definida la operación `G`, aunque sabemos que en caso de ejecutar, no existiría realmente ningún error de inconsistencia de tipos, pues el tipo real del objeto almancenado en `a` es `B`:

```
A a = new B();
a.G();
```

Por analogía, le llamaremos *tipo estático* al tipo declarado de una variable, atributo, método, o cualquier construcción sintáctica que almacene o produzca un valor, y *tipo dinámico* al tipo asociado a dicho valor durante la ejecución del programa. Es decir, en el caso anterior, `a` es una variable con tipo estático `A`, pero que en tiempo de ejecución almacena un objeto cuyo tipo dinámico es `B`. Más adelante podremos formalizar esta noción.

En esta sección nos dedicaremos entonces a construir un **verificador de tipos**, que nos es más que un algoritmo que nos dirá si todos los usos de tipos en nuestro AST son consistentes. Idealmente, queremos que nuestro verificador de tipos nos permita decidir exactamente cuáles programas hacen un uso consistente de los tipos, pero como hemos visto en el ejemplo anterior, en ocasiones es imposible determinar exactamente cuál será el tipo dinámico de una expresión en tiempo de ejecución. En estos caso, generalmente preferimos errar por exceso, es decir, evitar la ejecución de aquellos programas donde *podría* existir una inconsistencia de tipos, aunque en la realidad no suceda. Cuando un verificador de tipos cumple esta propiedad, decimos que es *consistente*. Es decir, un verificador consistente detecta todos los programas con errores de tipo, aunque puede decidir erróneamente que un programa correcto es incorrecto. Por supuesto, queremos reducir al mínimo posible este segundo caso.

En los lenguajes tipados es muy común que se permita construir *jerarquías de tipos*. Estas jeraquías se construyen mediante una operación, denominada generalmente **herencia**, que define que un tipo `B` es un *subtipo* del tipo `A`. La semántica exacta de la herencia varía de lenguaje en lenguaje, pero en general significa que todas las operaciones definidas para `A` también lo están para `B`, aunque `B` puede introducir nuevas operaciones (métodos, atributos, etc.), o *sobrescribir* la implementación de algunas de las operaciones definidas en `A`.

Esta sobrescritura generalmente se asocia al nombre de **polimorfismo**, que para nuestro

interés simplemente será el mecanismo que permite que una expresión `a.f()` se traduzca como la ejecución de una implementacion particular de `f` que depende del tipo dinámico de `a`, y no simplemente la implementación definida en el tipo estático. A este proceso le llamamos *resolución de métodos virtuales*.

Las reglas de la herencia varían en diversos lenguajes de programación, pero en general se distinguen dos grandes paradigmas, los lenguajes con *herencia simple*, donde cada tipo puede heredar de un solo tipo "padre", y los lenguajes con herencia múltiple. En los primeros, la jerarquía de tipos es un árbol (o conjunto de árboles, si no existe un tipo base de todos los tipos), y en el segundo caso la jerarquía de tipos se comporta como un grafo dirgido y acíclico (por supuesto, la herencia cíclica es, en principio, imposible). Del mismo modo que con los lenguajes estáticos y dinámicos, existen argumentos a favor y en contra de cada paradigma, aunque de forma general, los lenguajes con herencia simple son más sencillos de verificar que los lenguajes con herencia múltiple.

En cualquier caso, de forma general existe una relación entre los tipos de una jerarquía, que llamaremos **relación de conformidad**, y denotaremos $B \leq A$, es decir, *B se conforma a A*, si se cumple que B hereda de A o, recursivamente, si hereda de algún tipo C que se conforme a A. Verificar esta relación de conformidad en una implementación concreta de un verificador de tipos implica recorrer el árbol o grafo de la jerarquía de tipos. De momento asumiremos la relación como dada, y más adelante veremos ideas para su implementación en un ejemplo concreto.

Verificando tipos

De modo que el problema que tenemos que resolver es, para un nodo particular del AST, si el uso de los tipos es consistente. Este proceso en general lo haremos *bottom-up*, ya que la consistencia del uso de tipos en una expresión particular dependerá de los tipos en sus partes componentes. Por lo tanto, en un recorrido en post-orden del AST, iremos computando los tipos asociados a los nodos "hijos", y en el retorno chequearemos en cada "padre" la consistencia y computaremos el tipo del padre. Por ejemplo, si estamos en un nodo `SumExpr` que representa una expresión binaria de suma, podemos decir que este nodo es consistente si y solo sí cada una de las expresiones son a su vez consistentes, y son de tipo `int`, y la expresión en general es de tipo `int` también:

```
class SumExpr : Expression {
    public Expression Left;
    public Expression Right;
    public Type NodeType;

    public bool CheckTypes() {
        if (!Left.CheckTypes() || !Right.CheckTypes()) {
            return false;
        }

        if (Left.NodeType != Type.Integer ||
            Right.NodeType != Type.Integer) {
```

```
            return false;
        }

        NodeType = Type.Integer;
        return true;
    }
}
```

De este modo, recursivamente, podemos computar el tipo de todas las expresiones de un lenguaje (todos los posibles nodos de un AST). Vamos a definir a continuación una notación formal para expresar esta noción del chequeo de tipos de forma recursiva. La notación que definiremos tiene la forma de una demostración en el lenguaje de la lógica de predicados. Comienza con una lista de precondiciones lógicas (sobre los tipos de las sub-expresiones), y termina con una conclusión que dice cuál es el tipo de la expresión actual. Por ejemplo, para el caso anterior, podemos escribir:

$$\frac{e_1 : Integer \quad e_2 : Integer}{\vdash e_1 + e_2 : Integer}$$

Podemos leer esta expresión de la siguiente forma: si e_1 es una expresión de tipo *Integer* y e_2 es una expresión de tipo *Integer*, entonces se deduce que la expresión $e_1 + e_2$ es de tipo *Integer*. El símbolo $\vdash$ significa "se deduce que".

Contextos

Una pregunta interesante es ¿qué sucede con las variables? Dado que estamos en un recorrido *bottom-up*, nos encontraremos la declaración de una variable luego de su uso. Luego, si nos encontramos un nodo `VarExpr` que representa el uso de una variable, ¿qué tipo le asociamos? En la sección *Validando las reglas semánticas* introdujimos el concepto de *contexto*, para almacenar las declaraciones anteriores, de forma que siempre supiéramos qué símbolo estaba declarado en cualquier expresión. Ahora vamos a extender este contexto, para especificar no solo los símbolos declarados, sino qué tipo está asociado a cada símbolo. Vamos a introducir entonces una función O que llamaremos *contexto de objetos*, y que usaremos de la forma $O(x) = T$ para decir que este nodo existe un símbolo x definido con tipo estático declarado T. Por tanto, extenderemos nuestra notación para incluir el contexto de objetos como parte de las precondiciones y la conclusión. Luego, la expresión suma quedaría de la forma:

$$\frac{O \vdash e_1 : Integer \quad O \vdash e_2 : Integer}{O \vdash e_1 + e_2 : Integer}$$

Que podemos leer de la siguiente forma: si dado el contexto de objetos O, podemos deducir (recursivamente) que el tipo de e_1 es *Integer* (e igual para e_2), entonces en este mismo

contexto de objetos podemos deducir que el tipo de $e_1 + e_2$ es $Integer$. Es importante notar que hemos dicho $O \vdash e : T$, y no $O(e) = T$, pues e es una expresión en el sentido general, y O solamente está definido para símbolos (variables, atributos, etc.). Por tanto, $O \vdash e : T$ nos indica que es necesario chequear el tipo de e recursivamente en el contexto O para computar el tipo que tiene la expresión.

Los contextos de objetos se modifican cuando aparecen expresiones (instrucciones) que introducen nuevos símbolos (e.g.declaraciones de variables). Supongamos entonces que tenemos una expresión de la forma $Tx \leftarrow e$, que indica que la variable x se define con el tipo T y se inicializa con la expresión e. Este es el tipo de instrucción que comunmente vemos en lenguajes tipo C para inicializar una variable recién declarada:

```
int x = 4 + int.Parse(Console.ReadLine());
```

En esta instrucción se introduce un nuevo símbolo en el contexto de objetos, con el nombre x, y el tipo T. Luego, es necesario verificar que el tipo de la expresión e se conforme al tipo T, y luego definir el tipo de retorno de toda la expresión. En el caso particular de C, una instrucción de tipo asignación como esta no puede ser usada como expresión, y por tanto no devuelve valor, así que usaremos el símbolo $\emptyset$ para especificar que no tiene asociado (equivalente a `void`). Para especificar la modificación del contexto de objetos, introduciremos la sintaxis $O[T/x]$ que significa, informalmente, un nuevo contexto de objetos con las mismas definiciones que tenía O, pero además adicionando la definición del símbolo x con tipo T. Formalmente:

$$O[T/x](c) = \begin{cases} T & c = x \\ O(c) & c \neq x \end{cases}$$

Armados con esta nueva notación, podemos definir la semántica de un nodo de declaración e inicialización:

$$\frac{\begin{array}{l} O \vdash e : T' \\ T' \leq T \end{array}}{O[T/x] \vdash Tx \leftarrow e : \emptyset}$$

Podemos leer esta definición de la siguiente forma: si en el contexto actual el tipo de la expresión e es T', y T' conforma a T, entonces en un nuevo contexto definimos x con tipo T, y el tipo de retorno de la expresión es `void`.

El otro problema de interés es cuando nos encontramos con la declaración o invocación de un método. De forma general, vamos a considerar que es permitido sobrescribir la implementación de un método Af en una clase $B \leq A$, siempre y cuando el tipo declarado de los parámetros de f y el tipo de retorno de f no cambien. Esto es lo que sucede en la mayoría de los lenguajes de tipado estático. En este caso, desde el punto de vista del chequeo de tipos, no nos interesa realmente cuál es la implementación concreta de f, ya que todas las implementaciones coinciden en cuanto a las definiciones de tipos.

Vamos a introducir entonces un nuevo tipo de contexto M, que llamaremos *contexto de métodos*, y que usaremos de la forma $M(T, f) = \{T_1, \ldots, T_{n+1}\}$, para expresar que el método

f definido en el tipo T (o definido en algún tipo T' tal que $T \leq T'$), tiene n argumentos de tipo $T_1, \ldots, T_n$, y tipo de retorno T_{n+1}. El contexto M no se modifica, sino que se construye en un primer recorrido por el AST, y luego simplememente sirve para consultar. De modo que asumiremos que en todo momento durante el chequeo de tipos, ya son conocidos de antemano todos los métodos declarados en todos los tipos accesibles por el programa que está siendo compilado.

Tenemos entonces dos tipos de expresiones interesantes, la *declaración* de un método, y la *invocación*. Comenzaremos por la invocación de un método de instancia. Supongamos una sintaxis de la forma $x \cdot f(e_1, \ldots, e_n)$ para la invocación de métodos. Tenemos entonces que verificar el tipo de la expresión x, obtener el método f asociado a ese tipo, verificar la conformación de los tipos de los argumentos, y entonces podemos computar el tipo de f:

$$\frac{\begin{array}{l} O \vdash x : T \\ M(T, f) = \{T_1, \ldots, T_{n+1}\} \\ O \vdash e_i : T_i' \ \ \forall i = 1 \ldots n \\ T_i' \leq T \ \ \forall i = 1 \ldots n \end{array}}{O, M \vdash x \cdot f(e_1, \ldots, e_n) : T_{n+1}}$$

La invocación de un método estático es muy similar. Supongamos una sintaxis de la forma $T \cdot f(e_1, \ldots, e_n)$, la diferencia fundamental es que no es necesario computar el tipo de la expresión a quién se le invoca el método, pues la clase está definida explícitamente. Todo lo demás es prácticamente idéntico.

$$\frac{\begin{array}{l} M(T, f) = \{T_1, \ldots, T_{n+1}\} \\ O \vdash e_i : T_i' \ \ \forall i = 1 \ldots n \\ T_i' \leq T \ \ \forall i = 1 \ldots n \end{array}}{O, M \vdash T \cdot f(e_1, \ldots, e_n) : T_{n+1}}$$

Para poder formalizar la declaración de métodos, tenemos que introducir un nuevo elemento en nuestra notación, que llamaremos C, y que representará la *clase* actual donde se está realizando la verificación de la expresión correspondiente. En los lenguajes sin orientación a objetos (C) o donde no todo el código reside dentro de una clase (C++, Python), podemos definir un tipo especial Ω que representa el contexto "global". En última instancia, lo que queremos es poder diferenciar un método f de otro del mismo nombre pero definido en un contexto diferente. Las clases son una de las posibles formas de definir un contexto, pero nada nos impide extender esta noción e incluir un contexto global. En algunos lenguajes se le llama *espacio de nombres* o **namespace** a un contexto donde todos los símbolos son de nombre distinto. En Python, por ejemplo, cada módulo define un nuevo espacio de nombres, donde los símbolos definidos ocultan, pero no sobrescriben a los símbolos definidos en otro módulo.

Por otro lado, vamos a extender el concepto de contexto de objetos, para especificar O_C como el contexto específico dentro de la clase C. Es decir, en este contexto daremos por supuesto que ya están definidos todos los atributos y métodos de C, que son visibles dentro del cuerpo de cada función. Por ejemplo, en C# serían todos los campos declarados en C o declarados en algún padre de C con la visibilidad adecuada (no `private`).

Podemos entonces presentar una formalización de la semántica de tipos la definición de métodos. Supongamos una sintaxis de la forma:

$$f(x_1 : T_1, \ldots, x_n : T_n) : T\{e\}$$

Donde $x_i : T_i$ representa un argumento de f de nombre x_i y tipo T_i, T es el tipo de retorno, y e representa el cuerpo del método (visto como una expresión para simplificar).

A grandes razgos el proceso es el siguiente: se definen los argumentos x_i en el contexto de objetos hijo de O_C donde se va a verificar la expresión e y se verifica que el tipo del cuerpo conforme al tipo declarado de la función. Un tratamiento especial es necesario para el valor asociado a la instancia actual, que generalmente se llama `this` (C#) o `self` (Python). Por definición este símbolo en la instancia donde se está definiendo el método tiene exactamente el tipo C:

$$\frac{\begin{array}{l} M(C, f) = \{T_1, \ldots, T_n, T\} \\ O_C[T_1/x_1, \ldots, T_n/x_n, C/self] \vdash e : T' \\ T' \leq T \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T\{e\}}$$

Existen disímiles expresiones con reglas de tipos diversas pero, a grandes razgos, hemos mostrado cómo luce la formalización de la semántica de tipos de una expresión arbitraria. Armados con los conceptos de contexto de objetos y métodos, y una definición formal de la semántica de tipos para un lenguaje concreto de ejemplo, ya estamos en condiciones de implementar un verificador de tipos.

Implementando un verificador de tipos

Para implementar un verificador de tipos necesitamos concretar dos elementos: una clase que nos permita manejar el contexto de objetos y el contexto de métodos, y un algoritmo para recorrer el AST. Vamos a ejemplificar algunos detalles de implementación, asumiendo un lenguaje orientado a objetos muy simple, con las siguientes características:

- Un programa consiste en una lista de definiciones de clases.
- Todas las clases se definen en el mismo espacio de nombres global.
- Cada clase tiene atributos y métodos.
- Los atributos tienen un tipo asociado.
- Los métodos tienen un tipo de retorno (que puede ser `void`), y una lista de argumentos.
- Todos los atributos son privados y todos los métodos son publicos.
- Existe herencia simple.
- Un método se puede sobrescribir sí y solo sí se mantiene exactamente la misma definición para los tipos de retorno y de los argumentos.
- No existen sobrecargas de métodos ni de operadores.
- El cuerpo de todo método es una expresión.

No vamos a especificar formalmente qué tipos de expresiones son válidas en este lenguaje, pero de forma general podemos pensar en ciclos, condicionales, expresiones aritméticas, inicializaciones (`new T()`), invocaciones a métodos, etc. En fin, todos los tipos de expresiones comunes en un lenguaje orientado a objetos moderno. Solamente formalizaremos algunas de estas expresiones cuando nos interese mostrar la implementación.

Comencemos por representar los conceptos de tipo, atributo y método:

```
interface IType {
    string Name { get; }
    IAtribute[] Attributes { get; }
    IMethod[] Methods { get; }
    IAttribute GetAttribute(string name);
    IMethod GetMethod(string name);
    bool DefineAttribute(string name, IType type);
    bool DefineMethod(string name, IType returnType,
                      string[] arguments, IType[] argumentTypes);
}

interface IAttribute {
    string Name { get; }
    IType Type { get; }
}

interface IMethod {
    string Name { get; }
    IType ReturnType { get; }
    IAttribute[] Arguments { get; }
}
```

Para representar un contexto podemos usar la siguiente *interface*, similar al contexto que hemos usado anteriormente en la sección *Validando las reglas semánticas*, pero modificada para adaptarse las definiciones anteriores:

```
interface IContext {
    IType GetType(string typeName);
    IType GetTypeFor(string symbol);
    IContext CreateChildContext();
    bool DefineSymbol(string symbol, IType type);
    IType CreateType(string name)
}
```

Esta *interface* básicamente nos representa el contexto de objetos. El contexto de métodos realmente lo tenemos modelado como parte de la definición de cada tipo, por lo que no es necesario tener un contexto de métodos global. El propio tipo *C* nos permitirá acceder al contexto de métodos del tipo actual. Por otro lado, para acceder al contexto de métodos de un tipo específico, basta resolver la instancia de `IType` correspondiente (mediante el método `GetType`).

Vamos a definir entonces un subconjunto de los nodos del AST que nos interesa modelar. Comenzaremos como siempre por una clase base:

```
public abstract class Node {

}
```

Nuestro siguiente nodo es un programa, que se compone de una lista de definiciones de clase:

```
public class Program : Node {
    public List<ClassDef> Classes;
}
```

Cada definición de clase a su vez define un tipo con un nombre y una lista de atributos y métodos:

```
public class ClassDef : Node {
    public string Name;
    public List<AttrDef> Attributes;
    public List<MethodDef> Methods;
}
```

Los atributos se definen como un nombre, tipo asociado, y una expresión de inicialización:

```
public class AttrDef : Node {
    public string Name;
    public string Type;
    public Expression Initialization;
}
```

Mientras que los métodos se definen como un nombre, una lista de argumentos (nombres y tipos), un tipo de retorno, y un cuerpo que es una expresión:

```
public class MethodDef : Node {
    public string Name;
    public string ReturnType;
    public List<string> ArgNames;
    public List<string> ArgTypes;
    public string ReturnType;
    public Expression Body;
}
```

Luego nos quedarían todos los nodos de la jerarquía de expresiones, que no mostraremos por simplicidad.

Centraremos a continuación nuestra atención en el problema de la verificación en sí. De forma general, primero tenemos que hacer un recorrido por todo el AST para encontrar todas las definiciones de tipos. Este primer paso es necesario antes de verificar los métodos o atributos de un tipo particular, ya que podemos tener una declaración de un tipo `A` con un atributo de tipo `B`, donde la declaración del tipo `B` aparece luego de la declaración de `A`. Por este motivo, es necesario recolectar primero todos los nombres de todas las clases

declaradas y adicionarlos al contexto. Luego es necesario volver a recorrer todo el AST pero esta vez recogiendo las declaraciones de métodos y atributos, a quienes ya podemos asociar los tipos correspondientes. Esta segunda vez también es necesaria antes de pasar a analizar el cuerpo de los métodos, pues podemos tener un método `F` llamando a otro método `G` que está definido posteriormente. De esta forma completamos el contexto de métodos, y podemos finalmente recorrer el AST una vez más, esta vez adentrándonos en el cuerpo de los métodos y verificando la semántica de las expresiones.

De modo que necesitamos hacer al menos 3 pasadas por el AST, ya que podemos tener definiciones que usen a su vez símbolos definidos posteriormente. Para evitar esto, en lenguajes como C y C++ es necesario declarar de antemano en archivo *header* (con extensión `.h`) los símbolos que luego serán definidos (en un archivo `.c`).

Una primera aproximación a este problema nos lleva a definir al menos 3 métodos recursivos, uno para buscar las definiciones de tipos, otro para los métodos y atributos, y otro finalmente para la semántica. A medida que nuestro lenguaje crece, la semántica se complica, y aparecen nuevas fases (como la generación de código que veremos más adelante), notaremos que se repite este patrón de recorrer todo el AST buscando algunos nodos particulares y haciendo algunas operaciones en ellos. Por lo tanto, esta solución de definir un método recursivo para cada posible recorrido por el AST se vuelve cada vez menos atractivo. Cada vez que adicionamos algún tipo de chequeo tendremos que modificar la jerarquía del AST para acomodar los métodos recursivos necesarios. Por otro lado, si decidimos introducir un tipo de nodo nuevo (porque nuestra gramática ha cambiado o porque es conveniente especializar alguna función semántica), tendremos que redefinir todos estos métodos de chequeo en consecuencia. Por último, la mayoría de estos métodos recursivos se van a parecer mucho, pues todos tienen que descender recursivamente por los nodos del AST, y realizar alguna operación en pre-orden o post-orden.

El problema que tenemos aquí es un caso típico de acoplamiento, que nos lleva a una replicación de comportamiento similar. Tenemos que reconocer que existen dos responsabilidades diferentes en cada uno de estos casos: una es la que juega cada nodo del AST, que consiste en representar la función semántica correspondiente; y la otra es justamente el procesamiento necesario a realizar en un nodo para verificar algún predicado semántico. Vamos entonces a separar la responsabilidad de procesar a un nodo del nodo en sí, y ponerla en una clase distinta. A este diseño que presentaremos se le denomina el **patrón visitor**, y es uno de los patrones de diseño más populares y útiles, especialmente en la implementación de compiladores.

El Patrón Visitor

Comenzaremos por definir la siguiente *interface*:

```
interface IVisitor<TNode> where TNode : Node {
    void Visit(TNode node);
}
```

La *interface* `IVisitor<TNode>` nos permitirá abstraer el concepto de procesamiento sobre un nodo. En cada implementación particular, escogeremos qué procesamiento realizar sobre

cada tipo de nodo particular, y cómo "caminar" sobre la porción de AST correspondiente. Veamos entonces como implementar la primera pasada, que nos permite recolectar todos los tipos definidos. Este *visitor* solamente se interesa en nodos de tipo `Program` y nodos de tipo `ClassDef`. Su tarea consiste en crear un contexto, y definir en este contexto todos los tipos que se encuentre:

```
public class TypeCollectorVisitor: IVisitor<Program>,
                                   IVisitor<ClassDef> {
    public IContext Context;

    public void Visit(Program node) {
        Context = new // ...

        foreach(var classDef in node.Classes) {
            this.Visit(classDef);
        }
    }

    public void Visit(ClassDef node) {
        Context.CreateType(node.Name);
    }
}
```

El llamado `this.Visit(classDef)` resolverá estáticamente la sobrecarga adecuada. Veamos ahora como implementar un *visitor* que construya todo el contexto de métodos y atributos. En este caso, nos interesan además los nodos de tipo `AttrDef` y `MethodDef`:

```
public class TypeBuilderVisitor : IVisitor<Program>,
                                  IVisitor<ClassDef>,
                                  IVisitor<AttrDef>,
                                  IVisitor<MethodDef> {

    public IContext Context;
    private IType currentType;

    public void Visit(Program node) {
        foreach(var classDef in node.Classes) {
            this.Visit(classDef);
        }
    }

    public void Visit(ClassDef node) {
        currentType = Context.GetType(node.Name);

        foreach(var attrDef in node.Attributes) {
            this.Visit(attrDef);
        }
```

```
        foreach(var methodDef in node.Methods) {
            this.Visit(methodDef);
        }
    }

    // ...
}
```

Hasta este punto simplemente hemos descendido por las definiciones de tipos hasta llegar a cada definición de atributo o método. En cada paso, nos hemos asegurado además de mantener una referencia al tipo concreto dentro del cual se están realizando las definiciones. Veamos entonces los métodos restantes:

```
public class TypeBuilderVisitor : IVisitor<Program>,
                                  IVisitor<ClassDef>,
                                  IVisitor<AttrDef>,
                                  IVisitor<MethodDef> {

    public IContext Context;
    private IType currentType;

    // ...

    public void Visit(AttrDef node) {
        IType attrType = Context.GetType(node.Type);
        currentType.DefineAttribute(node.Name, attrType);
    }

    public void Visit(MethodDef node) {
        IType returnType = Context.GetType(node.ReturnType);
        var argTypes = node.ArgTypes.Select(t => Context.GetType(t));

        currentType.DefineMethod(node.Name, returnType,
                                 node.ArgNames.ToArray(),
                                 argTypes.ToArray());
    }
}
```

Por último, tendremos un `TypeCheckerVisitor` que verificará finalmente la consistencia de tipos en todos los nodos del AST. Este implementará la *interface* `IVisitor<T>` en cada tipo de nodo que sea capaz de chequear, incluidos todos los tipos de expresiones e instrucciones que no hemos definido. A este `TypeCheckerVisitor` le pasaremos el contexto ya construido anteriormente, y lo dejaremos procesar todo el AST. Este *visitor* además de verificar el tipo de todas las expresiones, será el encargado de computar el tipo asociado a cada expresión y almacenarlo en el nodo `Expression`:

```
public abstract class Expression {
    public IType ComputedType;
```

```
}
```

Hemos obviado hasta el momento cualquier consideración de error. Si alguno de los tipos definidos para alguno de los atributos o métodos no es válido, o alguno de los tipos aparece declarado más de una vez, o cualquier otro error semántico es detectado, nuestro verificador lanzará una excepción, o peor, fallará silenciosamente. Para manejar los errores de forma consistente, adicionaremos a los métodos `Visit` un argumento `IErrorLogger` con la *interface* siguiente:

```
public interface IErrorLogger {
    void LogError(string msg);
}
```

De modo que ante un error cualquier de chequeo de tipos, simplemente nos remitiremos a este objeto para indicar que ocurrió un error. Una vez detectado el error, la pregunta que queda es ¿qué hacer a continuación? Si detenemos el chequeo entonces nuestro compilador solamente será capaz de detectar un error en cada iteración, pero idealmente quisiéramos indicar la mayor cantidad de errores posibles en cada corrida. Por lo tanto, lo que se sugiere es tomar alguna acción de reparo sensata y continuar la verificacion. Por ejemplo:

```
public class TypeCheckerVisitor : IVisitor<...> {
    public Context Context;

    public void Visit(BinaryExpr node, IErrorLogger logger) {
        this.Visit(node.Left, logger);
        this.Visit(node.Right, logger);

        if (node.Left.ComputedType != node.Right.ComputedType) {
            logger.LogError("Type mismatch...");
            node.ComputedType = null;
        }
        else {
            node.ComputedType = node.Left.ComputedType;
        }
    }
}
```

Con estas herramientas, podemos adicionar un método en la clase `Program` que realice todas las pasadas correspondientes:

```
public class Program : Node {
    // ...

    public void CheckSemantics(IErrorLogger logger) {
        var typeCollector = new TypeCollectorVisitor();
        typeCollector.Visit(this, logger);

        var typeBuilder = new TypeBuilderVisitor() {
            Context = typeCollector.Context
```

```
        };

        typeBuilder.Visit(this, logger);

        var typeChecker = new TypeCheckerVisitor() {
            Context = typeBuilder.Context;
        };

        typeChecker.Visit(this, logger);
    }
}
```

Capítulo 7

Generación de Código

Código de 3 Direcciones Orientado o Objetos (CIL)

Para la generación de código intermedio de COOL a MIPS vamos a diseñar un lenguaje de máquina con capacidades orientadas a objeto. Este lenguaje nos va a permitir generar código de COOL de forma más sencilla, ya que el salto directamente desde COOL a MIPS es demasiado complejo. Este lenguaje se denomina CIL, **3-address object-oriented**.

Un programa en CIL tiene 3 secciones:

Jerarquía de Tipos

La primera es una sección (opcional) de declaración de tipos:

```
.TYPES

type A {
    attribute x ;
    method f : f1;
}

type B {
    attribute x ;
    attribute y ;
    method f : f1 ;
    method g : f2 ;
}

type C {
    attribute x ;
    attribute z ;
```

```
    method f : f2;
}
```

El "tipo" de los atributos en CIL no importa, pues todos los atributos son de tipo numérico. El único tipo por valor es `Integer` que almacena un entero de 32 bits. Todos los demás tipos son por referencia, y por lo tanto se representan por el valor de 32 bits del lugar de memoria virtual donde se ubican.

El orden de los atributos es muy importante, ya que luego veremos instrucciones para acceder a los atributos que usan realmente la dirección de memoria del atributo. Esta dirección está definida por el orden en que se declaran. Por este motivo, por ejemplo, si la clase `A` hereda de la clase `B`, y la clase `B` tiene un atributo `x`, es importante que en ambas declaraciones de tipos haya un atributo `x` y además que esté en el mismo orden. Lo mismo sucede con los métodos. Más adelante cuando completemos la generación de código veremos estos detalles más en profundidad.

Datos

En la sección de datos se declaran todas las cadenas de texto constantes que serán usadas durante todo el programa.

```
.DATA

msg = "Hello World";
```

Funciones

> Se toma como convenio que la primera función declarada es la función que se ejecuta al iniciar el programa.

```
.CODE

function f1 {
    ...
}

function f2 {
    ...
}
```

El cuerpo de cada función se divide a su vez en dos secciones. Primero se definen todos los parámetros y variables locales, y luego las instrucciones en sí:

```
function f1 {
    PARAM x;
    PARAM y;

    LOCAL a;
    LOCAL b;
```

```
    <body>
}
```

El cuerpo de una función en CIL se compone de una secuencia de instrucciones, que siempre reciben a lo sumo 3 argumentos, uno para almacenar el valor de retorno, y 2 operandos.

Entre las instrucciones básicas tenemos:

Asignación simple

```
x = y ;
```

Operaciones aritméticas

```
x = y + z ;
```

Acceso a atributos

El acceso a atributos se utiliza para obtener el valor de un atributo almacenado en un lugar de la memoria en una variable temporal.

```
x = GETATTR y b ;
```

Es equivalente a `x = y.b`.

```
SETATTR y b x ;
```

El "atributo" `b` realmente es solamente la dirección de memoria donde está el atributo.

Es equivalente a `y.b = x`.

Acceso a arrays

Cuando una variable es de un tipo array (o `string`), se puede acceder al iésimo elemento:

```
x = GETINDEX a i ;
```

O asignar un valor:

```
SETINDEX a i x ;
```

Manipulación de memoria

Para crear un nuevo objeto existe una instrucción de alocación de memoria:

```
x = ALLOCATE T ;
```

Esta instrucción crea en memoria espacio suficiente para alojar un tipo `T` y devuelve en `x` la dirección de memoria de inicio del tipo.

Para obtener el tipo dinámico de una variable se utiliza la sintaxis:

```
t = TYPEOF x ;
```

Para crear arrays se utiliza una sintaxis similar:

```
x = ARRAY y ;
```

Donde y es una variable de tipo numérico (como todas) que define el tamaño del array.

Invocación de métodos

Para invocar un método se debe indicar el tipo donde se encuentra el método, además del método en concreto que se desea ejecutar.

```
x = CALL f ;
```

Este llamado es una invocación estática, es decir, se llama exactamente al método `f`.

Además existe un llamado dinámico, donde el método `f` se buscan en el tipo `T` y se resuelve la dirección del método real al que invocar:

```
x = VCALL T f ;
```

Todos los parámetros deben ser pasados de antemano, con la siguiente instrucción:

```
ARG a ;
```

Cada método espera que los parámetros estén ubicados en la memoria en el mismo orden en que están declarados en el método. Es responsabilidad del que invoca pasar los parámetros de forma adecuada.

En particular, para todos los métodos "de instancia", que tiene un argumento `self` o similar, es responsabilidad del que invoca pasar este `self` como primer parámetro.

Saltos

Los saltos condicionales en CIL siempre tienen una condición y una instrucción de salto:

```
IF x GOTO label ;
```

Donde `label` tiene que ser una etiqueta declarada en algún lugar de la propia función. La etiqueta puede estar declarada después de su uso. Si la etiqueta no está declarada correctamente, el resultado de la operación no está definido.

```
LABEL label ;
```

Los saltos incondicionales simplemente se ejecutan con:

```
GOTO label ;
```

Como en CIL no existen variables *booleanas*, el valor de verdad de una expresión es realmente que la expresión sea distinta de cero, por lo tanto, los siguientes saltos son equivalentes:

```
x = y != z;
IF x GOTO label ;
```

```
x = y - z
IF x GOTO label ;
```

Retorno de Función

Finalmente todas las funciones deben tener una instrucción de retorno:

```
RETURN x ;
```

Esta instrucción pone el valor de `x` en la dirección de retorno de `f` y además termina la ejecución de la función. Si esta instrucción no se ejecuta en el cuerpo de una función el resultado de la invocación de la función no está bien definido. Si no importa el valor de retorno de la función, simplemente se puede usar cualquiera de las siguientes dos variantes, (que son equivalentes):

```
RETURN ;

RETURN 0 ;
```

Funciones de cadena

Las cadenas de texto se pueden manipular con funciones especiales. Primero es necesario obtener una dirección a la cadena:

```
x = LOAD msg ;
```

Luego se puede operar sobre una cadena con instrucciones tales como `LENGTH`, `CONCAT` y `SUBSTRING`, con la semántica esperada:

```
y = LENGTH x ;
```

Además hay una función STR que computa la representación textual de un valor numérico y devuelve la dirección en memoria:

```
z = STR y ;
```

Operaciones IO

Finalmente, hay 2 instrucciones de entrada-salida, `READ` que lee de la entrada estándar hasta el siguiente cambio de línea (incluído):

```
x = READ ;
```

Y `PRINT` que imprime en la salida estándar, sin incluir el cambio de línea.

```
PRINT z ;
```

Programas de ejemplo

Veamos algunos programas de ejemplo directamente escritos en CIL antes de pasar a definir cómo haremos la generación de código.

Hola Mundo

El "Hola Mundo" en CIL es muy sencillo:

```
.DATA

msg = "Hello World!\n" ;

.CODE

function main {
    LOCAL x ;

    x = LOAD msg ;
    PRINT x ;
    RETURN 0 ;
}
```

Ahora, este programa si lo fuéramos a generar desde COOL realmente sería un poco más complejo. En COOL sería necesario tener una clase con un método `main`, y algunos detalles adicionales:

```
class Main: IO {
    msg : string = "Hello World!\n";

    function main() : IO {
        self.print(self.msg);
    }
}
```

El programa completo en CIL que representa al programa anterior de COOL sería el siguiente. Hemos tenido cuidado de usar convenios de nombres que luego nos será útil respetar durante la generación de código.

```
.TYPES

type Main {
    attribute Main_msg ;
    method Main_main: f1 ;
}

.DATA

s1 = "Hello World!\n";

.CODE

function entry {
    LOCAL lmsg ;
```

```
    LOCAL instance ;
    LOCAL result ;

    lmsg = LOAD s1 ;
    instance = ALLOCATE Main ;
    SETATTR instance Main_msg lmsg ;

    ARG instance ;
    result = VCALL Main Main_main ;

    RETURN 0 ;
}

function f1 {
    PARAM self ;

    LOCAL lmsg ;

    lmsg = GETATTR self Main_msg ;
    PRINT lmsg ;

    RETURN self ;
}
```

Podemos ver que el código generado es más ineficiente que el código que se podría concebir manualmente. Este es un resultado necesario del hecho de subir de nivel de abstracción. Un programador humano escribiendo directamente CIL probablemetne pueda generar código más eficiente que nuestro compilador de COOL (al menos hasta que comencemos a aplicar técnicas de optimización de código), pero la ganancia en productividad es tal, que no tiene sentido tal comparación.

Generando código

Veamos entonces algunas ideas sobre cómo generar código concreto de COOL en CIL. Primero nos vamos a concentrar en la generación de expresiones concretas y al final veremos cómo combinarlo todo para generar un programa completo, con sus clases y métodos virtuales.

Un lenguaje plantilla para la generación de código

Para ejemplificar y documentar nuestras reglas de generación de código vamos a expandir el lenguaje CIL y el lenguaje COOL con una notación informal de "plantilla". No vamos a ser muy formales con esta notación, ya que no la usaremos nada más que como documentación. La idea básica es que permitiremos algunas expresiones "vagas", como puntos suspensivos, nombres genéricos, etc., donde nos sea conveniente. Además, no seremos estrictos con escribir todo el código CIL necesario, solamente la parte que corresponda al fragmento de

COOL que nos interesa generar, y asumiremos que el lector entiende el contexto. A medida que veamos ejemplos se irá haciendo más claro este "lenguaje".

Supongamos que tenemos una expresión en COOL de la forma:

```
let x : Integer = 5 in
    x + 1
end
```

Esta expresión puede ser parte del cuerpo de un método cualquiera, rodeada por cualquier contexto. Asumamos que el método se denomina `f`, a falta de un nombre mejor. Como no hemos especificado exactamente aquí todo el contexto de la expresión, tendremos que ser vagos con la generación de código. Permitiremos entonces obviar las partes "poco importantes" y concentrarnos solo en la parte que nos interesa. Sin más, presentamos el código CIL:

```
function f {
    ...
    LOCAL x ;
    LOCAL <value> ;

    ...
    x = 5 ;
    <value> = x + 1 ;
    ...
}
```

Aquí hemos tomado por convenio que el valor de la expresión lo guardamos en una variable de nombre `<value>`, que hemos puesto entre angulares `<...>` para indicar que este no es el nombre final que quedará en el código CIL real, sino un nombre plantilla que estamos usando ahora a falta de no tener el contexto completo. Es decir, en la generación real de código ese nombre se sustituirá por algo como `value_017` o cualquier indicador que sea conveniente en ese momento, generado de forma automática.

Por otro lado, supongamos que tenemos una expresión de la forma:

```
let x : Integer = 5 in
    x + <expr>
end
```

En esta expresión también estamos abusando de la notación, con el uso de `<expr>` para indicar que en COOL aquí va una expresión válida cuya forma exacta no nos interesa. Podemos entonces tomar un convenio como el siguiente:

```
function f {
    ...
    <expr.locals>
    LOCAL x ;
    LOCAL <value> ;

    ...
```

```
    x = 5 ;
    <expr.code>
    <value> = x + <expr.value> ;
}
```

El convenio que estamos tomando aquí, es usar `<expr.locals>` para indicar que se debe rellenar esta parte con todas las inicializaciones de variables locales que sean necesarias para poder computar la expresión `<expr>`. Además, luego en el cuerpo del método usamos `<expr.code>` para indicar que se debe rellenar con todo el código generado para la expresión `<expr>`. Finalmente, asumimos que `<expr.value>` será reemplazado por el nombre de la variable local que haya sido escogida para almacenar temporalmente el valor de `<expr>`.

Esperamos que esta notación se vaya haciendo más clara a medida que veamos más ejemplos. Vamos a comenzar entonces con ejemplos concretos:

Expresiones Let-In

Sea la expresión genérica de COOL `let-in` con la forma:

```
let <var> : <type> = <init> in
    <body>
end
```

Supongamos que esta expresión ocurre dentro de un método `f` arbitrario. Podemos entonces definir la generación de código de esta instrucción de la siguiente forma:

```
function f {
    ...
    <init.locals>
    LOCAL <var> ;
    <body.locals>
    LOCAL <value> ;

    ...
    <init.code>
    <var> = <init.value> ;
    <body.code>
    <value> = <body.value> ;
}
```

Puede parecer que no hemos ganado mucho con esta notación, pero en realidad hemos logrado definir para una expresión de tipo `let-in` arbitraria, exactamente lo que hace falta para su generación de código en CIL. Si interpretamos las expresiones `<init.locals>` y `<body.locals>` como, recursivamente, generar las inicializaciones de variables correspondientes, e igualmente `<init.code>` y `<body.code>` como, recursivamente, generar los fragmentos de código asociados, podemos ver que esta sintaxis nos permite decir de forma precisa en qué orden hay que recorrer el AST generando qué parte de cada tipo de nodo. No es difícil ver cómo este tipo de plantillas se convierten en respectivos llamados a *visitors* adecuados para cada parte del código.

Capítulo 8

El lenguaje HULK

En este capítulo definimos el lenguaje **HULK** (*Havana University Language for Kompilers*), un lenguaje de programación didáctico diseñado para este curso. A grandes razgos, **HULK** es un lenguaje orientado a objetos, con herencia simple, polimorfismo, y encapsulamiento a nivel de clases. Además en **HULK** es posible definir funciones globales fuera del contexto de cualquier clase. También es posible definir *una única expresión global* que constituye el punto de entrada al programa.

La mayoría de las construcciones sintácticas en **HULK** son expresiones, incluyendo las instrucciones condicionales y los ciclos. **HULK** es un lenguaje estáticamente tipado con inferencia de tipos opcional, lo que significa que algunas (o todas) las partes de un programa pueden ser anotadas con tipos, y el compilador verificará la consistencia de todas las operaciones.

Un lenguaje didáctico e incremental

El lenguaje **HULK** ha sido diseñado para ser utilizado como mecanismo de aprendizaje y evaluación de un curso de Compilación. Por tal motivo, ciertas decisiones de diseño de lenguaje responden más a cuestiones didácticas que a cuestiones teóricas o pragmáticas. Un ejemplo ilustrativo es la inclusión de un solo tipo numérico básico. En la práctica los lenguajes de programación cuentan con varios tipos numéricos (`int`, `float`, `double`, `decimal`) para cubrir el amplio rango de *trade-off* entre eficiencia y expresividad. Sin embargo, desde el punto de vista didáctico, ya es suficiente complejidad el tener que lidiar con un tipo numérico, y la inclusión de otros no aporta nada nuevo desde nuestro punto de vista.

Otra decisión importante es el tipado estático con inferencia de tipos, que será explicado más adelante en detalle. La motivación detrás de esta característica es permitir a los estudiantes implementar primero un evaluador para el lenguaje, y luego preocuparse por la verificación de tipos. Así mismo, la decisión de tener expresiones globales, funciones globales, y clases, responde a la necesidad de introducir los diversos elementos del lenguaje poco a poco. Al tener expresiones globales, es posible implementar un intérprete de expresiones sin

necesidad de resolver los problemas de contexto. Luego se pueden introducir las funciones y finalmente las características orientadas a objetos. De esta forma los estudiantes pueden ir aprendiendo sobre la marcha a medida que adicionan características al lenguaje, siempre teniendo un subconjunto válido del lenguaje implementado.

El lenguaje **HULK** realmente es un conjunto de lenguajes de programación muy relacionados. Lo que llamaremos **HULK** *básico* consiste en un subconjunto mínimo al que se le definen un conjunto extenso de características adicionales. El lenguaje *básico* contiene expresiones, funciones globales y un sistema unificado de tipos con herencia simple. Las extensión incluyen desde soporte para *arrays*, hasta delegados, inferencia de tipos, iteradores, entre otras características. Cada una de estas extensiones se ha diseñado para que sea compatible con el resto de **HULK**, incluyendo el resto de la extensiones. Debe ser posible, una vez implementado un compilador básico, adicionar cualquier subconjunto de estas extensiones.

Este diseño ha sido concebido para permitir el uso de **HULK** en un amplio rango de niveles de aprendizaje. Como lenguaje de expresiones y funciones, es útil para cursos introductorios sobre *parsing* y técnicas básicas de compilación. La orientación a objetos introduce todo un universo de complejidades semánticas; sin embargo, el sistema de tipos de **HULK** es suficientemente sencillo como para ilustrar los problemas más comunes en la verificación semántica de tipos. Por su parte, cada una de las extensiones introduce problemáticas avanzadas e interesantes. Los *arrays* introducen problemas relacionados con el manejo de memoria, mientras que las funciones anónimas y los iteradores son fundamentalmente problemas de transpilación y generación de código. La inferencia de tipos y la verificación de *null-safety* es todo un ejercicio en inferencia lógica, que puede servir en cursos avanzados. La idea es que cada curso defina sus objetivos de interés, y pueda utilizar un subconjunto apropiado de **HULK** para ilustrar y evaluarlos.

HULK básico

En esta sección definimos el subconjunto mínimo de **HULK**, que llamaremos *básico*. Este subconjunto consiste en un lenguaje de expresiones y funciones globales con tipado estático, y un sistema unificado de tipos con herencia simple.

El programa más sencillo en **HULK** es, por supuesto, *Hola Mundo*:

```
print("Hola Mundo");
```

En **HULK** un programa puede ser simplemente una expresión (terminada en `;`), en este caso, la invocación de una función global llamada `print` (que se presentará más adelante). La cadena de texto `"Hola Mundo"` es un literal del tipo `String`, definido en la biblioteca estándar, que se comporta de la forma convencional en la mayoría de los lenguajes de programación más populares: es inmutable, y codificado en UTF8.

Tipos básicos

Además de los literales de cadena, en **HULK** hay 2 tipos básicos adicionales: `Number`, que representa valores numéricos y `Boolean` que representa valores de verdad (con los literales usuales `True` y `False`).

El tipo `Number` representa tanto números enteros, como números con coma flotante. La semántica concreta dependerá de la arquitectura. Siempre que sea posible se representará con un valor entero de 64 (o 32) bits, o en su defecto, un valor flotante de 64 (o 32) bits, según permita la arquitectura. Las constantes numéricas se pueden escribir como `42` (valor entero) o `3.14` (valor flotante).

HULK tiene una jerarquía de tipos unificada, cuya raíz es el tipo `Object`. Todos los demás tipos definidos en el lenguaje son concretos.

Expresiones elementales

En **HULK** se definen todas las expresiones usuales, en orden decreciente de precedencia:

- Operaciones lógicas entre expresiones de tipo `Boolean`: `a & b`, `a | b`, `!a`, siempre evaluadas con cortocircuito.
- Operaciones aritméticas entre expresiones de tipo `Number`: `-a`, `a % b`, `a * b`, `a / b`, `a + b`, `a - b`, con la precedencia y asociatividad usuales, y agrupamiento mediante paréntesis. El resultado es de tipo `Number` siempre.
- Comparaciones entre expresiones de tipo `Number`: `a < b`, `a > b`, `a <= b`, `a >= b`, con menor precedencia que las aritméticas, y sin asociatividad.
- Comparaciones de igualdad entre expresiones de cualquier tipo: `a == b`, `a != b`, con la semántica de igualdad por valor entre expresiones de tipo `Number`, `String` o `Boolean`, e igualdad por referencia en todos los demás tipos. Se permite comparar expresiones de cualquier tipo, y si sus tipos no son compatibles (e.j, `"Hola Mundo" == 42`) el resultado será `False`.
- El operador infijo `@` de concatenación entre `String`: `"Hello "@"World" == "Hello World"`. Para los casos donde es conveniente, el operador `@@` adiciona un espacio intermedio: `"Hello" @@ "World"` es igual a `"Hello World"`.

En el espacio de nombres global siempre se encontrarán además las funciones `print`, `read` y `parse`, además de una serie de funciones elementales matemáticas, tales como `exp`, `pow`, `log`, `sqrt`, `min`, `max` y `random`.

El valor `Null` es un valor especial que puede tener cualquier tipo, excepto `Number` y `Boolean`. `Null` representa la no existencia de una instancia asociada a la variable (*l-value* en caso general) correspondiente, y cualquier operación que se intente sobre un valor `Null` lanzará un error en tiempo de ejecución, excepto `==`, que siempre devuelve `False`, y `!=` (que siempre devuelve `True`, incluso entre expresiones ambas iguales a `Null`). El literal `Null` es una expresión de tipo `Object` con valor igual a `Null`.

Variables

Las variables en **HULK** se introducen con una expresión `let`:

```
let <var>[:<type>]=<init> in <body>
```

La semántica de la expresión `let` consiste en que se crea un nuevo ámbito donde se define la variable `<var>` cuyo valor es el resultado de evaluar `<init>`, y se evalua en este ámbito la expresión `<body>`.

```
let msg:String="Hola Mundo" in print(msg)
```

Como se verá, indicar el tipo de una variable al declararla es opcional. Los detalles de la inferencia y verificación de tipos se darán más adelante.

Existe una variante extendida de la expresión `let` en la que se permite introducir más de una variable:

```
let x=1, y=2, z=3 in x+(y*z)
```

Esta variante es semánticamente idéntica a:

```
let x=1 in let y=2 in let z=3 in x+(y*z)
```

El cuerpo de una expresión `let` puede ser también una *lista de expresiones*, encerradas entre `{` y `}` y separadas por `;`, siendo el valor final de la expresión `let` el valor de la última expresión de la lista.

```
let x=0 in {
    print(x==0); # Imprime True
    print(x==1); # Imprime False
}
```

Una *lista de expresiones* **no es** una expresión en sí, es decir, no puede ser usada donde quiera que se requiera una expresión. Solamente se puede usar en el cuerpo de algunas construcciones sintácticas que se irán introduciendo poco a poco. Es decir, el siguiente ejemplo **no es válido**:

```
let x={0;1} in print(x) # NO es válido
```

Como tampoco lo es:

```
{1;2} + {3;4;let x=5 in 5} # NO es válido
```

Ni ningún otro ejemplo similar donde se use un bloque de expresiones como una expresión, excepto en los contextos donde se indique explícitamente más adelante.

Asignación

La asignación en **HULK** se realiza con el operador `:=`, y solamente es posible asignar a una variable que exista en el contexto actual:

```
let color="green" in {
    print(color);    # Imprime green
    color:="blue";
    print(color);    # Imprime blue
}
```

La asignación devuelve el valor asignado, y asocia a la derecha. Tiene menor prioridad que todas las expresiones aritméticas:

```
let x=0, y=0 in {
    y := x := 5 + 5;
    print(x); # 10
```

```
    print(y); # 10
    y := (x := 5) + 1;
    print(x); # 5
    print(y); # 6
}
```

Espacios en blanco e indentación

En **HULK** los espacios en blanco no son significativos, ni tampoco la indentación. La sintaxis del lenguaje permite indicar explícitamente, cuando es necesario, el ámbito de un bloque de código. El ejemplo anterior es equivalente a:

```
let x=1 in
let y=2 in
let z=3 in
    x + (y * z)
```

O cualquier otra forma de indentar que sea conveniente.

Identificadores

Los identificadores empiezan con un caracter del alfabeto ASCII (o _) y pueden opcionalmente contener números. Ejemplos de identificadores válidos son:

```
x
name
CamelCase
smallCamelCase
snake_case
_ugly_case
hunter42
```

Funciones

HULK soporta funciones globales con 2 formas sintácticas muy similares. Una función se define por un nombre, argumentos con tipo opcional, un tipo de retorno también opcional, y el cuerpo. Todas las funciones globales deben ser definidas **antes** de la expresión global que define el *cuerpo* del programa.

En la primera forma sintáctica, que llamamos "compacta", el cuerpo de la función debe ser exactamente una expresión (terminada en `;`):

```
function isEven(n:Number):Boolean -> n % 2 == 0;
```

En la segunda forma, que llamaremos "extendida", el cuerpo de una función puede ser una *lista de expresiones* separadas por `;`. El valor de retorno de la función es el valor de la última expresión de la lista. En esta notación **no se incluye** un `;` al final de la declaración de la función.

```
function f(a:Number, b:Number, c:Number):Number {
    a := b + c;
    b := c + a;
    c := a + b;
}

let a:Number=1, b:Number=2, c:Number=3 in print(f(a,b,c)); # Imprime 13
```

En **HULK** no existe una instrucción ni palabra reservada con semántica similar a `return`. Todas las funciones tienen un tipo de retorno y devuelven siempre un valor, aunque este valor puede ser `Null`.

Condicionales

Las expresiones condicionales se introducen con la sintaxis siguiente:

```
if (<cond>) <body> [elif (<cond>) <body>]* [else <body>]?
```

Es decir, una parte `if`, seguida de cero o más partes `elif` y finalmente una parte `else` opcional. Una expresión `if` devuelve el valor de la parte que se ejecuta. Si no se ejecuta ninguna (no hay else), devolverá `Null`. Si esto invalida la consistencia de tipos (como se verá más adelante), será necesario definir una parte `else` para garantizar al compilador/intérprete el tipo esperado.

```
function fib(n:Number):Number -> if (n <= 1) 1 else fib(n-1) + fib(n-2);
```

Al igual que con las expresiones `let` y las funciones, cada cuerpo puede ser o bien una expresión o una lista de expresiones. Si no hay parte `else` y ninguna rama condicional se ejecuta, se devuelve `Null`.

Ciclos

La expresión de ciclo más general en **HULK** es un ciclo `while` con la semántica común:

```
while (<cond>) <body>
```

Como ya es usual, `<body>` puede ser una expresión o una lista de expresiones.

```
function gcd(a:Number, b:Number):Number {
    let q:Number = a%b in while (q != 0) {
        a := b;
        b := q;
        q := a%b;
    };

    b;
}
```

El valor de retorno de la expresión `while` es el valor de retorno del cuerpo la última vez que se ejecutó el ciclo, o `Null` en caso de que nunca se ejecute. Si es necesario, se puede adicionar una cláusula `else` para definir el valor cuando no haya ejecución.

Con esta expresión, la manera más sencilla de implementar un contador (el común ciclo `for`) es:

```
let i:Number=0 in while (i < n) {
    # ...
    i := i+1;
}
```

Lidiando con valores `Null`

Si una variable tiene valor `Null`, se lanzará un error en tiempo de ejecución si se intenta cualquier operación sobre ella (expecto `==` y `!=`). **No** es posible evitar esto comprobando explícitamente:

```
if (x != Null) x.value else 0
```

Ya que el operador `!=` devuelve `True` siempre que uno de los dos valores sea `Null`. Es decir `Null != Null == True`. Para estos casos, **HULK** introduce una sintaxis específica:

```
with (<expr> as <id>) <expr> [else <expr>]
```

Por ejemplo, en este caso:

```
with (x as o) o.value else 0
```

La ventaja de esta sintaxis es que dentro del cuerpo de `with` se garantiza que `<id>` nunca será `Null`. Además, la variable `<id>` es **una referencia de de solo lectura** a la expresión `<expr>`, y el compilador impide que se le hagan asiganciones. De modo que es posible garantizar que este código nunca lanzará excepción en tiempo de ejecución por accesos o usos de `<id>`.

Prioridad de las expresiones

Las expresiones `let`, `if`, `while`, `case` y `with` tienen **menor prioridad** que todas las expresiones elementales (aritméticas, etc.), y siempre asocian a la derecha. Por lo tanto, para poder usar una de estas expresiones dentro de una expresión aritmética, por ejemplo, se deben encerrar entre paréntesis.

Por ejemplo, el siguiente es un caso común:

```
let x:Number=5 in let y:Number=8 in x+y
```

Que es equivalente a:

```
let x:Number=5 in (let y:Number=8 in (x+y))
```

Sin embargo, el siguiente caso **no es válido**, pues no se puede sumar con `let` sin parentizar (`let` tiene menor prioridad):

```
let x:Number=5 in x + let y:Number=8 in y # NO es valido
```

La forma correcta es:

```
let x:Number=5 in x + (let y:Number=8 in y)
```

Por último, el bloque `else` siempre asocia al `if` (o `while`) más cercano. Es decir, la siguiente expresión:

```
if (a) if (b) y else z
```

Es no ambigua, y equivalente a:

```
if (a) (if (b) y else z)
```

Por otro lado, la invocación de funciones, instanciación, el acceso a miembros (e.g., `self.x`) y el indizado en *arrays* tienen mayor prioridad que todas las expresiones elementales aritméticas.

Orientación a objetos en HULK

Además de las características estructuradas y funcionales presentadas, el lenguaje **HULK** soporta el concepto de *tipo*, implementado mediante *clases*. Todos los valores creados en un programa de **HULK** tienen un tipo asociado, y este tipo no puede ser cambiado en tiempo de ejecución. Por esto decimos que **HULK** es un lenguaje con tipado estático.

Aparte de los tipos nativos presentados (`Number`, `Boolean` y `String`), es posible definir nuevos tipos mediante la sintaxis:

```
class <name>[<args>] [is <base>[<init>]] {
    [<attribute>;]*
    [<method>]*
}
```

Todas las clases deben ser definidas **antes** que todas las funciones globales, pero esto *no impide* que dentro del cuerpo de un método en una clase (explicado más adelante), se llame a una función global, o se use una clase definida posteriormente.

Todas las clases en **HULK** heredan de una clase base. En caso de no especificarse, esta clase será `Object`, que es la raíz de la jerarquía de tipos en **HULK**. Los tipos básicos `Number`, `String` y `Boolean` también heredan de `Object`, pero a diferencia del resto de las clases, **no es permitido heredar de los tipos básicos**. Esto se restringe ya que los tipos básicos generalmente se implementan de forma especial para garantizar una mayor eficiencia, y por lo tanto deben ser tratados con cuidado en la jerarquía de tipos.

Atributos y métodos

Dentro del cuerpo de una clase se pueden definir dos tipos de elementos: atributos y métodos. Los atributos se definen con un nombre, un tipo opcional, y una expresión de inicializacion *obligatoria* (terminado en `;`):

```
class Point {
    x:Number = 0;
    y:Number = 0;
}
```

Todos los atributos en **HULK** son **privados**, es decir, no está permitido acceder a ellos desde otras clases, ni desde clases herederas.

Los métodos se definen con una sintaxis muy similar a las funciones globales. La única diferencia es que en el contexto de un método siempre existe una variable implícita `self` que referencia a la instancia en cuestión. Es obligatorio acceder a los atributos y métodos de una clase a través de `self`, **nunca** usando su nombre directamente.

```
class Point {
    x:Number = 0;
    y:Number = 0;

    translate(x,y) -> Point(self.x + x, self.y + y);
    length() -> sqrt(self.x * self.x + self.y * self.y);
}
```

Todos los atributos deben ser definidos **antes** que todos los métodos, y sus expresiones de inicialización no pueden utilizar métodos de la propia clase, ni valores de otros atributos (aunque sí pueden utilizar funciones globales). Todos los métodos en **HULK** son **públicos** y **virtuales**, redefinibles por los herederos. Además, todos los métodos son de instancia, no existen métodos estáticos, y no existe sintaxis para invocar a un método que no sea a través de una referencia a una instancia de una clase.

Instanciando clases

Para obtener una instancia de una clase en **HULK** se utiliza el nombre de la clase como si fuera un método, precedido la palabra clave `new`.

```
let p = new Point() in print(p.translate(5,3).length());
```

Si se desea inicializar los atributos de la clase, se pueden definir *argumentos de clase*, y su valor usarse en la inicialización de los atributos:

```
class Point(x:Number, y:Number) {
    x:Number = x;
    y:Number = y;
    # ...
}
```

Una vez definidos argumentos de clase, es obligatorio proporcionar su valor al construir la clase:

```
let p:Point = new Point(5,3) in print(p.length());
```

Redefinición y polimorfismo

En **HULK** todas las invocaciones a métodos de una instancia son polimórficas. Todos los métodos en **HULK** son virtuales, y pueden ser redefinidos, siempre que se mantenga la misma signatura (cantidad y tipo de los parámetros y retorno). La redefinición se realiza

implícitamente si se define en una clase heredera un método con el mismo nombre de una clase ancestro.

```
class Person(name:String) {
    name:String=name;
    greet() -> "Hello" @@ self.name;
}

class Colleague is Person {
    greet() -> "Hi" @@ self.name;
}
```

Al heredar de una clase se heredan por defecto las definiciones de los argumentos de clase. Por lo tanto, al instanciar una clase heredera, es obligatorio proporcionar los valores de los argumentos:

```
let p:Person = new Colleague("Pete") in print(p.greet()); # Hi Pete
```

Sin embargo, **no está permitido** usar estos argumentos de clase implícitos en la inicialización de atributos de una clase heredera. Si es necesario usarlos, se pueden redefinir explícitamente en la clase heredera. Por otro lado, siempre que se redefinan argumentos de clase en una clase heredera, será necesario indicar explícitamente cómo se evaluan los argumentos de la clase padre en términos de los argumentos de la clase heredera:

```
class Noble(title:String, who:String) is Person(title @@ who) { }

let p = new Noble("Sir", "Thomas") in print(p.greet()); # Hello Sir Thomas
```

Evaluando el tipo dinámico

La expresión `case` permite comparar el tipo dinámico de una expresión con una lista de tipos posibles. Su sintaxis es la siguiente:

```
case <expr> of {
    [<id>:<type> -> <body> ;]+
}
```

Esta expresión compara el tipo dinámico de `<expr>` contra cada uno de los tipos `<type>`, y ejecuta el `<body>` correspondiente a la rama del ancestro más cercano:

```
class A { }
class B is A { }
class C is B { }
class D is A { }

case new C() of {
    a:A -> print("A");
    b:B -> print("B"); # Se ejecuta esta rama
    d:D -> print("D");
}
```

En caso de ninguna rama ser válida en tiempo de ejecución, se lanza un error. En caso de poderse inferir el tipo de `<expr>`, se intentará validar la compatibilidad con los tipos `<type>`, y se lanzará un error semántico de existir. El cuerpo de una rama cualquiera puede ser una lista de expresiones entre `{` y `}` si fuera necesario, como sucede con las funciones.

Existe una versión compacta también de `case` cuando hay una sola rama, con la forma:

```
case <expr> of <id>:<type> -> <body>
```

Esta forma puede usarse para evaluar un "downcast" en **HULK**, cuando se conoce con certeza el tipo dinámico de un objeto.

Por ejemplo, el siguiente programa lanza error semántico pues `o` es de tipo estático `Object`, explícitamente declarado, por lo que no se puede sumar.

```
function dunno():Object -> 40;

let o:Object = something() in o + 2; # error semántico
```

Sin embargo, usando `case` se puede forzar al verificador de tipos a que infiera `Number` para esta expresión, lanzando error en tiempo de ejecución si realmente el tipo dinámico fuera otro.

```
function dunno():Object -> 40;

let o:Object = something() in case o of y:Number -> y + 2;
```

Hasta este punto, el lenguaje **HULK** básico definido representa un reto suficientemente interesante como proyecto de un semestre en un curso estándar de compilación. A continuación se definen un conjunto de extensiones que complejizan considerablemente el lenguaje en distintas dimensiones.

Extensiones a HULK

En esta sección definimos un conjunto de extensiones a **HULK**. Todas estas extensiones adicionan elementos sintácticos o semánticos al lenguaje. Todas las extensiones están diseñadas para que sean compatibles con el lenguaje **HULK** *básico*, y también compatibles entre sí. De este modo, un compilador de **HULK** puede decidir implementar cualquier subconjunto de estas extensiones.

Extensión: Arrays

Esta extensión introduce un tipo nativo con semántica similar al *array* de los lenguajes de programación de la familia C. Este *array* **no crece** dinámicamente y **no es covariante** en la asignación. El *array* es un tipo genérico que se declara con una sintaxis especial. Como es de esperar, un *array* se indiza con una expresión de tipo `Number` y el valor del primer índice es `0`. Si el valor del índice no es entero, se lanzará un error en tiempo de ejecución, al igual que si el índice sobrepasa los límites del array. Para los efectos del sistema de tipos, todo

tipo *array* hereda directamente de `Object`, y no se puede heredar de él. Todos los *arrays* en **HULK** tienen semántica de enlace por referencia.

```
let a:Number[] = {1, 2, 3, 4}, i:Number=0 in while (i < a.size()) {
    print(a[i]);
    i := i+1;
}
```

La forma de crear un array vacío es mediante la sintaxis:

```
let x:Number[] = new Number[20] in ...
```

Todo *array* se inicializa con el valor por defecto del tipo correspondiente, que es `0` para `Number`, `False` para `Boolean`, y `Null` para cualquier otro tipo. En **HULK** no existen *arrays* multidimensionales, pero sí es posible crear *arrays* de *arrays* (*ad infinitum*):

```
let x:Number[][] = new Number[][10] in ...
```

En este caso, el único *array* que se inicializa es el más externo. Cada valor de `x[i]` se inicializa con `Null` y debe asignarse explícitamente:

```
let x:Number[][] = new Number[][10], n=0:Number in while(n<x.size()) {
    x[i] := new Number[i+1];
}
```

El tipo de un *array* complejo como el anterior sería `Number[][]`.

Extensión: Inicialización automática de *arrays*

Cuando se instancia un *array*, es posible inicializarlo automáticamente mediante la sintaxis:

```
new <type>[<length>] { <index> -> <expr> }
```

Donde `<expr>` es una expresión que devuelve el valor del i-ésimo elemento, indizado por la variable `<index>`. Por ejemplo:

```
new Number[20]{ i -> 2*i+1 }
```

Es equivalente a:

```
let x:Number[] = new Number[20], i:Number=0 in while (i<x.size()) {
    x[i] := 2*i+1;
    i := i+1;
    x;
}
```

La expresión de inicialización es un atajo sintáctico para no escribir un ciclo de inicialización. Por tal motivo, se comporta exactamente igual que dicho ciclo. Esta expresión crea un nuevo contexto donde se define la variable `<index>`. Este contexto es hijo del contexto donde se crea el *array*. En particular, **no tiene acceso** a la variable que referencia al propio *array* (en caso de existir). Por ejemplo, la siguiente expresión **no** es válida:

```
# NO es válido
let x:Number[] = new Number[20]{ i -> if (i<=1) 0
```

```
                                    else x[i-1] + x[i-2] }
```

Ya que cuando se ejecuta la inicialización, la variable `x` aún no se ha definido. Para poder realizar una construcción de este tipo, introduciremos el uso de la palabra clave `self`, que servirá dentro de una expresión de inicialización para referirse al propio *array*:

```
# SI es válido
let x:Number[] = new Number[20]{ i -> if (i<=1) 0
                                    else self[i-1] + self[i-2] }
```

Pues es equivalente a:

```
let x:Number[] = new Number[20], i:Number=0 in while (i<x.size()) {
    x[i] := if (i<=1) 0 else x[i-1] * x[i-2];
    i := i+1;
    x;
}
```

De esta forma es posible inicializar *arrays* complejos en una sola expresión, como en el ejemplo siguiente:

```
new Number[][100] { i -> new Number[100] { j -> i*j } }
```

Que es equivalente a:

```
let x:Number[] = new Number[][100], i:Number=0 in while (i<x.size()) {
    x[i] := new Number[100];
    let j:Number=0 in while (j < x[i].size()) {
        x[i][j] := i * j;
        j := j+1;
    };
    i := i+1;
    x;
}
```

Extensión: Interfaces

Esta extensión introduce el concepto de *interfaz* en **HULK**. Una interfaz es básicamente una declaración de los métodos que debe contener un tipo:

```
interface Piece {
    canMove(dx:Number, dy:Number): Boolean;
}
```

Una vez declarada una interfaz, se puede utilizar como una anotación de tipo en cualquier lugar donde se requiera un tipo en **HULK**, es decir, en una variable, parámetro, tipo de retorno, etc.

Las interfaces en **HULK** son compatibles con cualquier tipo que implemente los métodos adecuados. **No** es necesario que una clase declare explíciticamente que es compatible con una interfaz.

Por ejemplo, la siguiente clase implementa la interfaz `Piece`.

```
class Bishop(x:Number, y:Number) {
    x:Number = x;
    y:Number = y;

    canMove(dx:Number, dy:Number): Boolean {
        abs(x - dx) == abs(y - dy);
    }
}
```

Para ser compatibile con una interfaz, una clase debe implementar **todos** los métodos. Cada método debe tener la misma cantidad de argumentos, aunque los nombres de los argumentos no importan. Los tipos de los argumentos son *covariantes* en la entrada y *contravariantes* en el retorno. Es decir, si una interfaz tiene un método:

```
interface I {
    method(arg1:T1, ..., argn:Tn): Tr;
}
```

Entonces una clase puede implementarla con un método:

```
class C {
    method(arg:T'1, ..., argn:T'n): T'r -> ...
}
```

Si se cumple que `T'i > Ti` y `T'r < Tr`. Es decir, recibe argumentos con tipos iguales o ancestros de los argumentos en la interfaz, y devuelve un tipo igual o descendiente.

Para el sistema de tipos, si una clase `C` implementa una interfaz `I` con estas condiciones, entonces se considera `C < I`. Por lo tanto, es posible usar como tipos de los argumentos de una interfaz a otras interfaces, mientras que las implementaciones pueden usar tipos concretos, u otras interfaces compatibles.

Extensión: Tipos funcionales

Esta extensión permite definir *tipos* que representan funciones, y pueden ser utilizadas como ciudadanos de primera clase dentro de la jerarquía de tipos (delegados).

Un *tipo funcional* se declara con la sintaxis siguiente:

```
(<type1>, <type2>, ...) -> <return>
```

Por ejemplo, para la función siguiente:

```
function fib(n:Number): Number -> if (n<=1) 1 else fib(n-1) + fib(n-2);
```

Una anotación de su tipo sería:

```
(Number) -> Number
```

Estas anotaciones sirven para declarar tipos funcionales que pueden ser usados entonces como argumentos o valores de retorno en otras funciones, o almacenados en variables:

```
let f:(Number)->Number = fib in ...
```

Nótese como se puede utilizar entonces el identificador `fib` (nombre de la función) como referencia a una instancia de un tipo funcional `(Number)->Number`. Una vez obtenida una referencia a un tipo funcional, se puede utilizar exactamente como cualquier otro valoren **HULK**, es decir, ser pasado como parámetro a un método, almacenado en un *array* (si se implementa esta extensión), o *invocado*.

Para invocar un tipo funcional se usa la misma sintaxis que para invocar directamente una función global:

```
let f:(Number)->Number = fib in print(f(6));
```

De la misma forma se pueden definir funciones con argumentos de tipos funcionales. Por ejemplo, si se implementan *arrays*, la siguiente función es posible:

```
function map(a:Number[], f:(Number)->Number):Number[] {
    new Number[array.size()] { i -> f(a[i]) };
}
```

Desde el punto de vista semántico, el tipo funcional anterior se comporta *como si* existiera una interfaz como la siguiente, y una clase respectiva que implementara la interfaz:

```
interface NumberNumberFunction {
    invoke(arg1:Number): Number;
}

class FibFunction {
    invoke(arg1:Number): Number -> fib(arg1);
}
```

Y el uso o invocación de este tipo funcional pudiera verse *como si* se empleara de la siguiente forma:

```
let f:NumberNumberFunction = new FibFunction() in print(f.invoke(6));
```

Con la particularidad de que, por supuesto, estas *supuestas* clases e interfaces no son accesibles por el código usuario directamente, sino solo a través de la sintaxis definida anteriormente.

Los tipos funcionales son invariantes tanto en los tipos de los argumentos como en el tipo de retorno.

Extensión: Funciones anónimas (expresiones *lambda*)

Esta extensión introduce funciones anónimas, también conocidas como expresiones *lambda*. Una función anónima es una expresión cuyo tipo es un funcional. Sintácticamente se declaran muy parecido a las funciones globales, excepto que no es necesario indicar un nombre. Semánticamente, las funciones anónimas son expresiones, por lo que pueden ser usadas donde se requiera un funcional. Por ejemplo, en una declaración de variables:

```
let f:(Number)->Number = function (x:Number):Number -> x * 2 in print(f(3));
```

O directamente como parámetro a una función o método:

```
function map(a:Number[], f:(Number)->Number) {
    new Number[]{ i -> f(a[i]) };
}

print(map(new Number[20]{ i -> i }, function (x:Number):Number -> x+2));
```

Al igual que las funciones globales, el cuerpo puede ser una expresión simple o un bloque de expresiones.

Extensión: Clausura funcional

Esta extensión adiciona a las expresiones *lambda* un mecanismo de clausura que permite capturar variables existentes en el contexto de la función automáticamente.

```
let x:Number=3, f:(Number)->Number = function (y:Number):Number -> y * x in
    print(f(5)); # 15
```

La captura se realiza **por copia** siempre, por lo que las variables capturadas **no mantienen su valor** luego de ejecutada la función. Sin embargo, la variable capturada visible dentro de la función anónima **si mantiente su valor** de una ejecución a la siguiente:

```
let x:Number=1 in
    let f:(Number)->Number = function (y:Number):Number {
        x := x + 1;
        y * x;
    } in {
        print(f(3)); # Imprime 6
        print(f(5)); # Imprime 15
        print(x);    # Imprime 1
    };
```

Semánticamente, la clausura se comporta como si en el momento de definir la función anónima se copiaran los valores de las variables capturadas a campos internos de una *supuesta* clase que representa el funcional. El ejemplo anterior sería equivalente semánticamente al siguiente código:

```
interface NumberNumberFunction {
    invoke(arg1:Number):Number;
}

class LambdaFunction1(x:Number) is NumberNumberFunction {
    x:Number = x;
    invoke(arg1:Number):Number {
        self.x := self.x + 1;
        y * self.x;
    }
}
```

```
let x:Number=1 in
    let f:NumberNumberFunction = LambdaFunction1(x) in {
        print(f.invoke(3)); # Imprime 6
        print(f.invoke(5)); # Imprime 15
        print(x);    # Imprime 1
    };
```

De esta forma se garantiza que el valor de `x` fuera de `f` no sea modificado, sin embargo, todas las ejecuciones de `f` comparten una misma referencia al `x` interno.

Extensión: Funciones genéricas

Esta extensión introduce funciones globales con argumentos de tipos genéricos. Un tipo genérico se define con la sintaxis `'T` donde `T` es un identificador. Los tipos genéricos pueden usarse en tipos complejos (por ejemplo, un *array*, un tipo funcional, etc.). Una función genérica puede declarar como tipo de uno o más de sus argumentos un tipo genérico.

```
function map(a:'T[], f:('T)->'R): 'R[] {
    new T'[] { i -> f(a[i]) };
}
```

Los tipos genéricos se consideran declarados en los argumentos de la función. Dentro del cuerpo de una función genérica se pueden utilizar los tipos genéricos como si fueran tipos concretos. En el momento en que se realice una invocación de la función, los tipos genéricos declarados se realizarán en un tipo concreto.

A los efectos semánticos, es como si cuando se realizara una invocación, se definiera en ese momento una versión de la función con los tipos concretos.

Extensión: Generadores (iteradores)

Esta extensión adiciona una sintaxis para usar generadores (también llamados iteradores). Un generador es cualquier clase que implemente dos métodos: `next()` y `current()`. El método `next()` devuelve `Boolean` y su función es avanzar el iterador. El método `current()` devuelve un tipo arbitrario y retorna el elemento "actual" del generador. Por ejemplo:

```
class Range(start:Number, end:Number) {
    i:Number = start;
    next():Boolean {
        self.i += 1;
        self.i <= end;
    }
    current():Number -> self.i - 1;
}
```

Cualquier clase que posea estos dos métodos puede ser usada con la siguiente sintaxis:

```
for (<var>:<type> in <generator>) <expr>
```

Por ejemplo:

```
for (x:Number in new Range(0,100)) print(x);
```

Esta expresión enlaza una variable `x` al valor `current()` del generador en cada iteración, y ejecuta la expresión correspondiente. Al igual que todas las expresiones con cuerpo (`let`, `while`), es posible usar una expresión simple o un bloque de expresiones.

La expresión `for` es sintácticamente equivalente a una construcción como la siguiente:

```
let _iter:<generator-type>=<generator> in while (_iter.next()) {
    let <var>:<type>=_iter.current() in <expr>;
}
```

Por ejemplo, en este caso:

```
let _iter:Range=new Range(0,100) in while(_iter.next()) {
    let x:Number=_iter.current() in print(x);
};
```

Extensión: Tipos generadores

Esta extensión adiciona una familia especial de tipos, los tipos generadores, que son covariantes con cualquier clase que implemente una interfaz de generador adecuada.

Un tipo generador se declara con la sintaxis:

```
<type>*
```

Donde `<type>` es un tipo cualquiera. Por ejemplo `Number*`, `Boolean*`, `Number[]*`, y por supuesto, `Number**` y cualquier otra composición. Los tipos generadores pueden usarse como tipo estático de una variable o parámetro donde se espere un generador, y son compatibles (covariantes) con cualquier tipo (clase) que cumpla la interfaz de generador correspondiente.

Por ejemplo, la siguiente clase cumple con la interfaz de un generador de tipo `Number*`:

```
class Range(start:Number, end:Number) {
    i:Number = start;
    next():Boolean {
        self.i += 1;
        self.i <= end;
    }
    current():Number -> self.i - 1;
}
```

Por lo tanto, puede ser usado en el siguiente fragmento:

```
function sum(items:Number*) {
    let s:Number=0 in for(x:Number in items) s:=s+x;
}

print(sum(new Range(1,100))); # 4950
```

De esta forma es posible escribir funciones "genéricas" con respecto a los tipos generadores, es decir, donde no sea necesario conocer de antemano el tipo concreto que implementa la interfaz de generador. Además, es posible escribir funciones que devuelvan tipos generadores y de esta forma simplificar la sintaxis:

```
function range(start:Number, end:Number):Number* -> new Range(start, end);

for (x in range(1,100)) print(x);
```

Las variables y parámetros declarados con un tipo generador pueden ser usados en una expresión `for`, pero además, es posible interpretarlos como una *supuesta* clase con métodos `next()` y `current()`, ya que la implementación "real" es esta. Por lo tanto, el siguiente código es válido:

```
let items:Number* = range(1,100) in items.next();
```

Aún cuando no sea posible acceder estáticamente al tipo concreto que implementa el generador, es sabido que tendrá al menos los métodos `next()` y `current()` con la semántica esperada, por lo que es posible implementar funciones importantes tales como:

```
function empty(items:Number*) -> !items.next();
```

O patrones como este:

```
class Take(items:Number*, k:Number) {
    items:Number* = items;
    k:Number = k;

    next():Boolean -> if (k>0) self.items.next() else False;
    current():Number -> self.items.current();
}

function take(items:Number*, k:Number):Number* -> Take(items,k);
```

Extensión: Expresiones generadoras

Esta extensión añade una sintaxis para definir expresiones generadoras, es decir, expresiones que pueden usarse con la sintaxis `for`, cuyo tipo estático será un tipo generador.

Una expresión generadora tiene la forma:

```
{ <expr> | <var> in <loop-expr> }
```

Donde `<expr>` es una expresión que involucra la variable `<var>`, y `<var>` captura el valor de retorno de `<loop-expr>`, que debe ser una de las siguientes expresiones:

- Un tipo generador.
- Un ciclo `while`.
- Un ciclo `for`.
- Una expresión `let` cuyo cuerpo es una de las anteriores.

Los siguientes ejemplos son todos válidos:

- Tipo generador explícito: `hulk { 2*x | x in new Range(1, 100) }`
- Función que devuelve un tipo generador: `hulk { x+1 | x in range(1, 100) }`
- Ciclo `while` (infinito en este caso): `hulk { exp(x) | x in while (True) random() }`
- Ciclo `for` (anidado en este caso): `hulk { x | x in for (i in range(1, 100)) for (j in range(1, 100)) i*j }`
- Expresión `let` con un ciclo en el cuerpo: `hulk { x | x in let i:Number=0 in while (i < 100) { i := i + 1; random(); }}`

La variable `<var>` tendrá sucesivamente todos los valores "producidos" para la expresión de iteración.

Un expresión generadora es semánticamente equivalente a una clase que implemente el tipo generador correspondiente (al tipo estático de `x`), y cuya implementación de los métodos `next()` y `current()` corresponda al comportamiento definido. El mecanismo exacto para lograr esta transpilación es demasiado complejo para formalizarlo en esta sección, pero daremos un ejemplo ilustrativo.

De manera general, el cuerpo del método `next()` corresponderá a la implementación de `<loop-expr>`, desenrrollada de forma tal que cada paso se ejecute en un llamado correspondiente, mientras que el cuerpo de `current()` corresponderá a la expresión `<expr>` de retorno del generador.

Por ejemplo, para la siguiente expresión:

```
{ exp(x) | x in let i:Number=0 in while (i < 100) {
                    i := i + 1;
                    random(); }}
```

Una posible implementación será el siguiente tipo generador:

```
class Generator {
    x:Number = 0; # Variable que almacenará current
    i:Number = 0; # Inicialización de la expresión let

    next():Boolean {
        if (!(self.i < 100)) False
        else {
            self.i := self.i + 1;
            self.x := random();
            True;
        };
    }

    current():Number -> exp(self.x);
}
```

En el caso de que el cuerpo de una expresión generadora sea un generador directamente, o un ciclo `for`, existen formas de convertirlo a un patrón `let-while` básico. De esta forma todas las expresiones generadoras pueden reescribirse como **HULK** básico.

Al igual que en las funciones anónimas, las variables externas referenciadas dentro de la clausura de una expresión generadora se capturan **por copia**. Por lo tanto, su modificación dentro de una expresión generadora no se percibe fuera de la expresión:

```
let i:Number = 0,
    items:Number* = { x | x in while (i<10) i := i+1 }
in {
    for (x:Number in items) print(x); # Imprime 1 ... 10
    print(i);                         # Imprime 0
}
```

Extensión: Inicialización de *arrays* con generadores

Esta extensión permite inicializar un *array* a partir de un generador, mediante la sintaxis:

```
new <array-type>[<generator>]
```

Por ejemplo, usando directamente un tipo generador:

```
new Number[range(1, 100)]
```

O una expresión generadora (en caso de implementarse):

```
new Number[{ 2*x | x in range(1, 100) }]
```

Nótese que esta sintaxis **no es** equivalente a la inicialización automática de *arrays*. En primer lugar no es necesario indicar el tamaño del *array*, pues se ejecutará el generador para saber el tamaño real necesario. Por otro lado, no es posible referirse al propio *array* que se está inicializando en la expresión generadora, ya que el *array* conceptualmente no existe hasta que no se haya terminado de generar toda la expresión.

Extensión: Inferencia de tipos

Esta extensión introduce inferencia de tipos en **HULK**, de modo que no sea necesario especificar todas las anotaciones de tipos, si son inferibles dado el contexto. Formalizar precisamente en qué casos es posible inferir o no el tipo de una declaración es un problema complicado, y no lo intentaremos en este punto. Por el contrario, presentaremos algunos ejemplos donde es posible inferir el tipo, y otros ejemplos donde el mecanismo de inferencia de tipos de **HULK** no será capaz de deducirlo.

El caso más sencillo, es cuando en una declaración de variable se omite el tipo. En este caso, el tipo se infiere de la expresión de inicialización:

```
let x=3+2 in case x of y:Number -> print("Ok);
```

De igual forma sucede con los atributos de una clase, cuando pueden ser inferidos por el tipo de la expresión de inicialización:

```
class Point(x:Number, y:Number) {
    x=x;
    y=y;
    # ...
}
```

Un caso más complejo es cuando se deja sin especificar el tipo de retorno de una función, pero puede ser inferido a partir de su cuerpo:

```
function succ(n:Number) -> n + 1;
```

En el caso anterior, es fácil inferir el tipo de retorno de `succ` porque la expresión retorna exactamente el mismo tipo que un argumento. En estos casos, es posible incluso no especificar el tipo del argumento, ya que el operador `+` solo está definido para `Number`:

```
function succ(n) -> n + 1;
```

Sin embargo, a veces no es posible inferir el tipo de un argumento a partir de su uso *dentro del cuerpo* de una función. En el caso siguiente, aunque sabemos que el tipo del argumento `p` debe ser `Point` para aceptar la invocación, *no se garantiza* que el mecanismo de inferencia de tipos deba deducirlo (ya que en el futuro puede haber otras clases con un método `translate`). Dependiendo de la implementación, en estos casos se permite lanzar error semántico indicando que no fue posible inferir el tipo del argumento `p`.

```
function step(p) -> p.translate(1,1);

let p = new Point(0,0) in step(p); # Puede lanzar error semántico
```

Por último, especial complejidad acarrean las funciones recursivas:

```
function fact(n) -> if (n<0) 1 else n*fact(n-1);
```

El ejemplo anterior permite inferir simultáneamente el tipo del argumento `n` y del retorno, ya que se usa el retorno de la función recursiva en una operación `+` que solo está definida para `Number`. Sin embargo, en el ejemplo siguiente:

```
function ackermann(m, n) ->
    if   (m==0) n+1
    elif (n==0) ackermann(m-1, 1)
    else        ackermann(m-1, ackermann(m, n-1));
```

Como el tipo de retorno no se utiliza explícitamente en una operación matemática, no es trivial deducir que su tipo de retorno es `Number`, ya que `Object` funcionaría también como tipo de retorno. En estos casos, se desea que el mecanismo de inferencia deduzca *el tipo más concreto* para el retorno y *el tipo más abstracto* para los argumentos que sea posible.

Finalmente, dos funciones mutuamente recursivas:

```
function f(a, b) -> is (a==1) b else g(a+1, b/2);
function g(a, b) -> if (b==1) a else f(a/2, b+1);
```

En este caso, es posible teóricamente inferir que `f` y `g` deben ambos retornar tipo `Number`, pero dada la complejidad de manejar la inferencia de tipos en más de una función a la vez,

no se garantiza que sea posible deducir los tipos en este caso.

Varias de las extensiones de **HULK** introducen tipos nuevos, por ejemplo, *arrays*, tipos generadores y tipos funcionales. En caso de implementar algunas de estas extensión, la inferencia de tipos también se debe extender de forma correspondiente.

Cuando se crea un *array*, es posible inferir el tipo estático si se provee una expresión de inicialización:

```
let x = new [20]{ i -> (i+1)*(i+2) } in ... # Infiere Number
```

Sin embargo, si en la expresión de inicialización se usa el propio *array*, entonces no se garantiza que sea posible inferir el tipo:

```
let x = new [20]{ i -> if (i<=1) 1 else x[i-1] + x[i-2] } in ...
```

Cuando se define un tipo funcional, el tipo puede ser inferido a partir de la declaración de una función global:

```
function fib(n) -> if (n <= 1) 1 else fib(n-1) + fib(n-2);

let f=fib in print(f(3)); # f infiere (Number) -> Number
```

Cuando se define una función anónima, el tipo puede ser inferido a partir de su cuerpo:

```
let f = function (x) -> x % 2 == 0 in print(f(10));
```

En ocasiones, se puede inferir el tipo de un parámetro a partir del tipo de la variable donde es almacenada la función anónima:

```
let f:(Person)->Boolean = function (p) -> p.name() == "John" in ...
```

En este caso hipótetico el compilador infiere `Person` para el argumento `p` gracias a la declaración de `f`, por lo que reconoce `p.name` como un método válido en esta clase.

En una expresión `for` el tipo de la variable puede ser inferido a partir del tipo del generador:

```
function range(start:Number, end:Number):Number* {
    new Range(start,end);
}

for (x in range(1,100)) print(x); # x infiere Number
```

Para las expresiones generadoras, el tipo puede ser inferido a partir del tipo de la variable generada. Por ejemplo, en el siguiente caso, el tipo inferido para `x` es `Number`, pues es el tipo de retorno del ciclo `while` (función `random()`). El tipo inferido para la expresion `exp(x) < 0` es `Boolean`. Por lo tanto el tipo inferido para toda la expresión `Boolean*`.

```
# g infiere Boolean*
let g = { exp(x) < 0 | x in let i:Number=0 in
                            while (i < 100) {
                                i := i + 1;
                                random(); }} in ...
```

Extensión: *Null-safety*

Gracias a la existencia de la instrucción `with`, un compilador de **HULK** puede ser capaz de inferir para un programa si es seguro garantizar que no habrá errores en ejecución por variables `Null`. Es posible entonces que un compilador genere en estos casos un código más eficiente, al no tener que validar las referencias en todos los accesos y usos. En todos los casos en que no se pueda garantizar la *null-safety*, el compilador debe emitir una advertencia en tiempo de compilación (*warning*).

Opcionalmente, el compilador de **HULK** puede ejecutarse en modo **null-safe**. En este modo, las advertencias por violación de *null-safety* se convierten en errores de compilación.

Nótese que no es estrictamente necesario usar `with` en todos los casos para garantizar la *null-safety*. Hay casos en que por el contexto es posible garantizar la seguridad. El caso más sencillo es cuando se usan variables de tipo `Number` o `Boolean`, que por definición no pueden contener un valor `Null`.

Un caso más interesante es cuando se puede inferir por la inicialización de una variable, por ejemplo:

```
let x = new Person("John Doe") in print(x.greet());
```

En este caso se puede inferir que `x` nunca será `Null` pues no existe ninguna asignación, y el valor de la expresión `new Person(...)` nunca es `Null`.

Por el contrario, si se usa un ciclo o una expresión `if` sin parte `else`, no es posible garantizar *null-safety*, incluso cuando realmente el valor de la expresión no pueda ser `Null`.

```
let i=10, x=(while (i > 0) i:=i-1) in print(x);
```

Así mismo, en los argumentos de una función nunca será posible garantizar *null-safety*.

De manera general el compilador de **HULK** hará todo lo posible por inferir si cada uso es *null-safe*, incluso cuando no se introduzca una expresión `with`. En los casos en que no sea posible inferirlo, el programador siempre podrá introducir una expresión `with` para satisfacer al compilador.

Extensión: Macros

Esta extensión introduce un sistema limitado de macros en tiempo de compilación con verificación semántica. Una función macro es similar a una función estándar (global) en sintaxis, pero se define con la palabra clave `define`:

```
define dmin(x:Number, y:Number):Number -> if (x <= y) x else y;
```

La diferencia más directa radica en que las funciones macro se evalúan en tiempo de compilación, y se expanden directamente en el lugar donde se usan. Por ejemplo, para el caso anterior, si se usa de la siguiente manera:

```
print(min(3,4));
```

En tiempo de compilación se sustituirá el código anterior directamente por el resultado de expandir la función macro, lo que sería equivalente a haber escrito:

```
print(if (3 <= 4) 3 else 4);
```

Una diferencia adicional entre las funciones macro y las funciones estándar, es que en las funciones macro los parámetros **no se evalúan**, sino que se expanden directamente en el cuerpo del macro. Por ejemplo:

```
print(min(3+2, 4+5));
```

Expande a:

```
print(if ((3+2) <= (4+5)) (3+2) else (4+5));
```

Otra diferencia que se desprende directamente de esta definición, es que los parámetros de una función macro **no son *l-values***, por lo que no es posible asignarles una expresión.

Esto cambia considerablemente el comportamiento en comparación con una función estándar global si un parámetro tiene efectos colaterales. Por ejemplo, supongamos las siguientes declaraciones:

```
function fmin(x:Number, y:Number):Number -> if (x<y) x else y;
define dmin(x:Number, y:Number):Number -> if (x<y) x else y;

function f(x:Number) {
    print(x);
    x;
}
```

Aparentemente `fmin` y `dmin` son equivalentes, sin embargo:

```
fmin(f(3), f(4));
```

Imprime una vez `3` y una vez `4`, ya que los argumentos se evalúan antes de ser pasados como parámetros a `fmin`. Por el contrario:

```
dmin(f(3), f(4));
```

Se expande directamente en el siguiente código:

```
if (f(3) < f(4)) f(3) else f(4);
```

Por lo que `f(3)` es llamado 2 veces. Por tal motivo, las funciones macro deben usar en una instrucción `let` para expandir los argumentos una sola vez, a menos que se desee explícitamente ejecutar el efecto colateral potencial de una función más de una vez (por ejemplo, si la intención es realizar un ciclo).

Las funciones macro también **son verificadas semánticamente**, siguiendo las mismas reglas semánticas de las funciones globales, por ejemplo, con respecto a la consistencia en los tipos de los parámetros, el tipo de retorno y el cuerpo del macro. Esto garantiza que las funciones macro son seguras de utilizar en cualquier contexto donde una función estándar con los mismos tipos sería válida.

Un ejemplo más sobre las diferencias semánticas entre una función macro y una función estándar radica en que los argumentos en las funciones globales se pasan por copia. Sin embargo, como las funciones macro no son invocadas, sino expandidas **en el mismo contexto**,

realmente no hay un paso de parámetros. Por tal motivo, los efectos colaterales sobre las variables del contexto donde de expande un macro funcionan de forma diferente.

Por ejemplo, si tenemos la siguiente función global:

```
function repeat(x:Number, y:Number):Number {
    while (x > 0) {
        x := x - 1;
        y;
    };
}

let y:Number=0 in print(repeat(10, y:=y + 1));
```

El resultado de invocar a `repeat` es `1` pues **primero** se evalúa `y := y + 1` y luego se pasa este valor por copia a `repeat`. Sin embargo, si se define como una función macro:

```
define repeat(x:Number, y:Number):Number {
    let i:Number=x in while (i > 0) {
        i := i - 1;
        y;
    };
}

let y:Number=0 in print(repeat(10, y:=y + 1));
```

En este caso la expansión de `repeat` genera un código semánticamente equivalente al siguiente:

```
let y:Number=0 in print(
    let i:Number=10 in while (i > 0) {
        i := i - 1;
        y := y + 1;
    }
);
```

En este caso el valor de `y` será `10`, pues la expresión `y := y + 1` se ejecuta **todas** las iteraciones del ciclo, en **el mismo contexto** donde se expande el macro. Este efecto simplemente no es posible en **HULK** usando solo funciones, ni siquiera con expresiones *lambda*, pues el paso de parámetros siempre es **por copia**.

Una función macro al expandirse puede generar un bloque de expresiones. Aunque en **HULK** no es permitido al usuario escribir directamente un bloque de expresiones en cualquier lugar (por ejemplo, como valor de un parámetro), esta es una restricción sintáctica, impuesta por el *parser*. Desde el punto de vista semántico los bloques de expresión se comportan como expresiones cuyo tipo y valor de retorno son los de la última expresión del bloque.

La expansión de un macro **no es** una simple sustitución léxica del cuerpo del macro en el lugar donde se invoca. Por el contrario, la expansión es sintáctica, y se realiza a nivel de AST. Esto significa que no es necesario que una definición de macro se preocupe por parentizar

expresiones o poner ; adicionales "por si acaso", pues en el momento de la expansión del macro, los elementos léxicos ya no importan.

Por otro lado, las funciones macros deben ser *higiénicas*, esto significa que **no pueden contaminar** el contexto donde se expanden con símbolos que puedan cambiar la semántica. Por este motivo, **todas las variables** declaradas dentro de un macro (ya sea en expresiones `let`, `for`, etc.) deben ser renombradas a símbolos internos que no puedan coincidir con los símbolos escritos por el programador. Si esto no se hiciera, entonces pudiera suceder que nombres de símbolos expandidos por el macro ocultaran símbolos del contexto. Por ejemplo:

```
let i:Number=0 in repeat(10, i := i + 1);
```

Generaría:

```
let i:Number=0 in
    let i:Number=10 in while (i>0) {
        i := i - 1;
        i := i + 1;
    }
```

Con lo cual el ciclo no terminaría nunca. Por este motivo realmente la sustitución debe generar un código más parecido al siguiente:

```
let i:Number=0 in
    let #var0:Number=10 in while (#var0>0) {
        #var0 := #var0 - 1;
        i := i + 1;
    }
```

Donde `#var0` es un identificador generador dinámicamente que no puede ser escrito por el usuario (por restricciones del *lexer*). De esta forma se garantiza que la expansión de un macro nunca introduzca símbolos que oculten otros símbolos en el contexto donde se usen. Esta sustitución **debe hacerse automáticamente**, es decir, en la definición del macro se puede declarar un símbolo `i` que será automáticamente sustituido por `#varX` o cualquier otro convenio similar.

El proceso de expansión de un macro puede terminar en una expresión que aún contenga macros. Estos macros volverán a ser expandidos, recursivamente, hasta que no quede ningún macro por expandir. Si este proceso produce una recursión infinita (en la práctica, una cantidad elevada de expansiones), se lanzará un error en tiempo de compilación. Idealmente, si es posible detectar durante el chequeo semántico del macro la posibilidad de una expansión infinita, se debe lanzar un error o *warning*.

Extensión: Bloques macro

Esta extensión permite invocar a las funciones macro con una sintaxis especial, donde el último parámetro puede ser definido fuera de los paréntesis de la invocación. Siguiendo con el mismo ejemplo de la sección anterior (macro `repeat`), esta extensión permitiría invocarlo de la forma:

```
let y:Number=0 in repeat (10) y:=y+1;
```

O de la forma:

```
let y:Number=0 in repeat (10) {
    y:=y+1;
};
```

Es decir, la expresión (o bloque de expresiones) que sigue a una función macro se usa como el último parámetro del macro. Esto permite simular en **HULK** construcciones sintácticas como la anterior, que parecen nativas del lenguaje, sin perder la verificación semántica ni la inferencia de tipos (opcional) que funciona en el resto de **HULK**. Veamos algunas macros interesantes que se pueden implementar con esta idea.

Es posible simular expresiones condicionales distintas, como la siguiente:

```
define unless (cond:Boolean, expr:Object):Object -> if (!cond) expr;
```

Que se puede usar de la siguiente forma:

```
let msg:String=read() in unless (msg=="Exit") {
    print("Hello World");
};
```

También es posible simular distintos sabores de ciclos, como este:

```
define until (cond:Boolean, expr:Object):Object -> while(!cond) expr;
```

Que se podría usar de la siguiente manera:

```
let x:Number=10 in until (x == 0) {
    x := x - 1;
    print(x);
};
```

Extensión: Patrones de expresiones en macros

Esta extensión añade un mecanismo de *pattern matching* para expresiones dentro de los macros. Este mecanismo permite definir macros cuya expansión dependa de la estructura sintáctica de los argumentos. Para ello introduciremos una expresión `match` que **solo** puede ser usada en macros. Esta expresión permite *deconstruir* la estructura sintáctica de una expresión arbitraria `e` y tomar decisiones en función de los elementos que la componen.

Veamos un ejemplo un tanto esotérico para introducir los elementos básicos de la sintaxis de patrones. Supongamos que queremos definir un macro `simplify` que recibe una expresión aritmética y devuelve una expresión de igual valor pero simplificada. Las reglas para simplificar serán muy sencillas: si es una expresión `+`, entonces si una de las dos partes es `0`, me quedo con la otra, y si es una expresión `*` entonces si una de las dos partes es `1` me quedo con la otra. En otro caso la expresión se mantiene igual.

```
define simplify(e:Number):Number {
    match e with {
        e1:Number + Number(0) -> simplify(e1);
```

```
        e1:Number * Number(1) -> simplify(e1);
        Number(0) + e1:Number -> simplify(e1);
        Number(1) * e1:Number -> simplify(e1);
        e1:Number             -> e1;
    };
}
```

En esta sintaxis, similar a `case`, la expresión `e` se compara *estructuralmente* (en términos del AST) con cada uno de los patrones (las expresiones en la parte izquierda de `->`). Para el **primer** patrón que sea compatible se expande entonces la expresión en la parte derecha de `->`. Si ningún patrón es compatible, se lanzará un error en tiempo de compilación durante la expansión del macro, indicando que ninguna de las formas esperadas era compatible. Para indicar *valores literales* (`Boolean`, `Number` y `String`) usaremos una sintaxis como la mostrada (e.g., `Number(4)` o `String("Hello World")`).

La compatibilidad se realiza mediante una *unificación* recursiva del AST de `e` con el AST de cada patrón. En este proceso, las variables (`e1` en el ejemplo) que existan en el patrón se unificarán con los sub-árboles correspondientes del AST. Las "variables" `e1`, `e2`, etc., que unificarán con sub-árboles del AST, deben anotarse con un tipo que indica el tipo más abstracto esperado. Es decir que `e:Object` unifica con cualquier árbol.

Por ejemplo:

```
simplify (1 * 3 + 0);
```

En este caso el primer patrón es compatible, siendo `e1` unificado con `(1 * 3)`, por lo que se obtiene la expansión:

```
simplify(1 * 3);
```

Como en esta expansión aún quedan macros, se vuelve a expandir `simplify` siendo compatible el cuarto patrón, `e1` unificando con `3`. La expansión final queda entonces:

```
3;
```

Nótese que la unificación puede funcionar si se asigna a la misma variable ASTs con estructura idéntica. Por ejemplo, si deseamos añadir como patrón que `x + x` se convierte en `2*x`, podemos definir el macro como:

```
define simplify(e:Number):Number {
    match e with {
        # ... los casos vistos anteriormente
        e1:Number + e1:Number -> 2 * e1;
        e1:Number             -> e1;
    };
}
```

En este caso se unificará solamente si ambas partes de la expresión `+` tienen exactamente la misma estructura, por ejemplo `3*2 + 3*2` unifica `e1` con `3*2`, pero `3*2 + 2*3` no funciona, ya que aunque semánticamente ambas sub-expresiones tendrán el mismo valor, no tienen exactamente la misma estructura sintáctica para los respectivos AST.

Todas las expresiones de **HULK** *que no definen símbolos* son usables como patrones en una expresión `match`, y para cada una la sintaxis como patrón se corresponde naturalmente con la sintaxis como expresión. Es decir, `let`, `for`, las expresiones generadoras, las expresiones *lambda* y las expresiones de inicialización de *array* **no pueden** ser usadas como patrones. También es posible unificar con una expresión compleja. En este caso, se verificará recursivamente cada parte de la expresión. Algunos ejemplos:

```
# ...
match e with {
    e1:Number + e2:Number                   -> ... # Expresiones
    e1:Number * e2:Number                   -> ... # aritméticas simples o
    e1:Number + (e2:Number - e3:Number)     -> ... # complejas,
    while (e1:Boolean) e2:Object            -> ... # Ciclos,
    if (c:Boolean) e1:Object                -> ... # Condicionales sin o
    if (c:Boolean) e1:Object else e2:Object -> ... # con parte else,
    x:Number                                -> ... # Variables,
    Number(5)                               -> ... # Literales,
    Person(n:String)                        -> ... # Instanciación,
    e1:Object := e2:Object                  -> ... # Asignación,
    fib(e1:Number)                          -> ... # Invocación a funciones
    e1:Person.greet()                       -> ... # y métodos de instancia.
}
```

Para el caso de los bloques de expresiones, es posible unificar directamente con un bloque de expresión que tiene una cantidad exacta de elementos:

```
match e with {
    { e1:Object ; e2:Object ; e3:Object } -> ...
}
```

Sin embargo, en ocasiones no se sabe de antemano la cantidad de expresiones que tendrá el bloque. Para estos casos, definiremos una sintaxis especial que permite *deconstruir* un bloque de expresiones con **más de 1 elemento** en 2 partes: la primera expresión, y un bloque de expresiones restante. Esto permite aplicar recursivamente un macro a un bloque de expresiones de longitud arbitraria:

```
define simplify(e:Number):Number {
    match e with {
        # ... los casos vistos anteriormente
        { e1:Number | e2:Number } -> { simplify(e1) ; simplify(e2) };
    }
}
```

En esta nueva sintaxis, el operador `|` sirve para indicar la concatenación de la primera expresión con el resto del bloque de expresiones. Nótese que para que esto funcione, `e2` no puede ser un bloque vacío, es decir, `e` originalmente contenía al menos 2 expresiones. Nótese que en esta situación el código expandido tendría la forma `{ e1 ; { e2; { e3; ...} } }`, es decir, los bloques se irían anidando a medida que se ejecuta el macro recursivamente. Sin embargo, `{ e1; { e2; } }` es semánticamente equivalente a `{ e1; e2 }` siempre, por

lo que el compilador es libre de "desenredar" esta anidación de bloques de expresiones de forma automática. Además, para que el caso base funcione, el compilador debe considerar un bloque con una sola expresión `{ e; }` equivalente a la misma expresión `e` por si sola.

El *pattern matching* en macros que hemos definido en esta extensión es uno de las técnicas más poderosas de meta-programación que se pueden lograr en un lenguaje con tipado estático. Nótese que el compilador puede tener que realizar un procesamiento potencialmente ilimitado (excepto por efectos prácticos en los límites de la recursividad). Aunque no lo hemos demostrado formalmente, el mecanismo de macros definido con *pattern matching* es Turing-completo. Una manera relativamente fácil de entender por qué, es notar que es posible codificar cualquier máquina de Turing como una secuencia de expresiones, representando los estados y los valores de la cinta mediante constantes. A base de *pattern matching* y expansiones de macro recursivas, es posible simular cualquier máquina de Turing *en tiempo de compilación*.

A modo de ejemplo veamos cómo resolver el conocido problema de identificar si una secuencia de números `0` o `1` representa un número en binario divisible por `3`. Este es un conocido lenguaje regular, cuyo autómata tiene `3` estados que representan los posibles restos con `3`. La función macro siguiente simula este autómata (la presentaremos sin demostración):

```
define multiple3(state:Number, value:Number):Boolean {
    match state with {
        Number(0) -> match value with {
            Number(0)                   -> True;
            Number(1)                   -> False;
            { Number(0) | rest:Number } -> multiple3(0, rest);
            { Number(1) | rest:Number } -> multiple3(1, rest);
        };
        Number(1) -> match value with {
            Number(0)                   -> False;
            Number(1)                   -> True;
            { Number(0) | rest:Number } -> multiple3(2, rest);
            { Number(1) | rest:Number } -> multiple3(0, rest);
        };
        Number(2) -> match value with {
            Number(0)                   -> False;
            Number(1)                   -> False;
            { Number(0) | rest:Number } -> multiple3(1, rest);
            { Number(1) | rest:Number } -> multiple3(2, rest);
        };
    };
}
```

Este macro se puede ejecutar de la siguiente forma:

```
multiple3 (0) { 1; 1; 0; };
```

Y se expandiría directamente al literal `True`, computando *en tiempo de compilación* un problema de la palabra en un lenguaje regular. No es muy complicado ver como extender esta

idea a simular una máquina de Turing. Solo es necesario "llevar la cuenta" de la posición del cabezal.

Formalización del lenguaje HULK

En esta sección presentaremos una descripción formal de **HULK**, en términos sintácticos, semánticos y operacionales. Esta sección debe servir como referencia para la construcción de un compilador de **HULK**, pero recomendamos las secciones anteriores que explican de forma más intuitiva todos los elementos relevantes del lenguaje.

Sintaxis de HULK

Una gramática posible para **HULK** se muestra a continuación. Nótese que por motivos didácticos, esta gramática es ambigua. En particular, la asociatividad y prioridad de los distintos tipos de expresiones se deja sin resolver, para dar espacio a los estudiantes a que resuelvan estos problemas.

Los terminales de **HULK** son:

```
NUMBER  := [0-9]+(.[0-9]+)?
STRING  := " UNICODE* " # Todos los caracteres unicode válidos
BOOLEAN := True | False
LITERAL := NUMBER | STRING | BOOLEAN
ID      := [a-zA-Z_][a-zA-Z0-9]*
```

Un programa en **HULK** tiene tres partes: declaraciones de clases, declaraciones de funciones y una expresión opcional terminada en punto y coma (`;`):

```
<program> := [<class>]* [<function>]* [<expr> ;]
```

Una clase contiene atributos y métodos, y opcionalmente declaraciones de argumentos de clases:

```
<class>  := class ID [(<params>)] [is ID [(<args>)]] { [<attr> ;]* [<method>]* }
<params> := ID [: ID] [, ID [: ID]]*
          | epsilon
<args>   := <expr> [, <expr>]*
          | epsilon
```

Los atributos tiene un nombre, opcionalmente un tipo y una inicialización obligatoria.

```
<attr> := ID [: ID] = <expr>
```

Un método tiene un nombre, argumentos, un tipo de retorno opcional y un cuerpo. Hay 2 tipos de notaciones para métodos, una donde el cuerpo es una expresión simple (y termina en `;`) y otra donde el cuerpo es una lista de expresiones.

```
<method> := ID ( <params> ) [: ID] <body>
<body>   := -> <expr> ;
          | { [<expr> ;]+ }
```

Una función global tiene una signatura muy parecida a un método, pero requiere la palabra clave `function`:

```
<function> := function ID ( <params> ) [: ID] <body>
```

Finalmente las expresiones se dividen en 8 tipos fundamentales:

```
<expr> := <let-expr>
        | <if-expr>
        | <while-expr>
        | <case-expr>
        | <assign-expr>
        | <array-expr>
        | <inst-expr>
        | <elem-expr>
```

Una expresión de tipo `let` se compone de un bloque de inicializaciones, y un cuerpo. Igual que las funciones, este cuerpo puede ser simple o compuesto por una lista de expresiones:

```
<let-expr>  := let <decls> in <expr-body>
<decls>     := <decl> [, <decl>]*
<decl>      := ID [: ID] = <expr>
<expr-body> := <expr>
             | { [<expr> ;]+ }
```

Una expresión de tipo `if` tiene un conjunto de condiciones y opcionalmente una cláusula `else`:

```
<if-expr> := if ( <expr> ) <expr-body>
             [elif ( <expr> ) <expr-body>]*
             [else <expr-body>]
```

Una expresión de tipo `while` tiene una condición, un cuerpo, y opcionalmente una cláusula `else`:

```
<while-expr> := while ( <expr> ) <expr-body>
                [else <expr-body>]
```

Una expresión de tipo `case` tiene una expresión y un conjunto de ramas compuestas por identificador, tipo, y expresión de retorno:

```
<case-expr> := case <expr> of <case-body>
<case-body> := ID : ID -> <expr-body>
             | { [ID : ID -> <expr-body> ;]* }
```

Una asignación tiene una locación a la izquierda y una expresión a la derecha:

```
<assing-expr> := <loc> ':=' <expr>
<loc>         := <loc> '[' <expr> ']'
               | <loc> . <loc>
               | ID
```

Una expresión de creación de *array* tiene un tipo opcional, una cantidad, y opcionalmente una cláusula de inicialización:

```
<array-expr> := new ID? '[' <expr> ']' [{ ID -> <expr> }]?
```

Una expresión de instanciación tiene un tipo y un conjunto de argumentos de clase:

```
<inst-expr> := new ID ( <args> )
```

Las expresiones elementales se componen de todas las operaciones lógicas, aritméticas, etc., además de la invocación a funciones globales, métodos, y atributos:

```
<elem-expr> := <expr> == <expr> | <expr> != <expr>
             | <expr> < <expr>  | <expr> > <expr>
             | <expr> <= <expr> | <expr> >= <expr>
             | <expr> & <expr>  | <expr> '|' <expr> | !<expr>
             | <expr> @ <expr>  | <expr> @@ <expr>
             | <expr> + <expr>  | <expr> - <expr>
             | <expr> % <expr>  | <expr> * <expr>   | <expr> / <expr>
             | <expr> [ '[' <expr> ']' ]?
             | [<expr> .] ID [( <args> )]?
             | -<expr> | ( <expr> )
```

NOTA: Como se ha explicado al inicio de la sección, la gramática anterior no tiene en cuenta la asociatividad ni precedencia de los diferentes tipos de expresiones. Esto se ha hecho a propósito, para permitir a los estudiantes resolver los problemas de ambigüedad resultantes del modo que consideren oportuno.

Semántica de tipos

En **HULK** todas las expresiones tienen asociado un tipo estático, que debe ser inferido por el compilador. Cada expresión o instrucción de **HULK** tiene reglas de consistencia de tipos que deben ser verificadas por el compilador. En esta sección asumiremos que todos los tipos están explícitamente declarados. La sección siguiente explica cómo realizar la inferencia de tipos cuando existan declaraciones sin anotaciones de tipo. Para definir el tipo inferido y las restricciones de consistencia de cada tipo de expresión usaremos la notación definida en el capítulo Semántica de Tipos.

Reglas para la inferencia de tipos

Semántica operacional

Implementando un Compilador de HULK

La implementación de un compilador de **HULK** tiene varios detalles y retos interesantes. A continuación queremos discutir algunas cuestiones que facilitarán esta tarea.

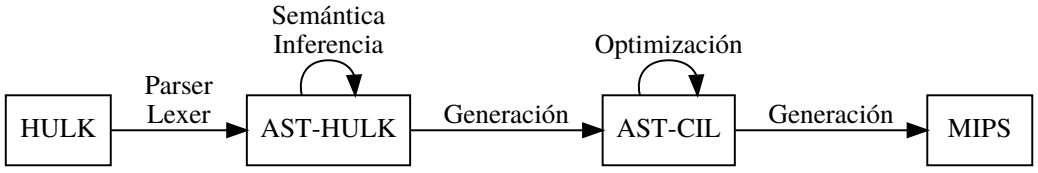
Consideraciones generales

El compilador de **HULK** es un proyecto complejo, que debe ser dividido convenientemente en subproblemas, para ser atacado de forma efectiva. Si el proyecto se realiza en un equipo, existen algunos puntos importantes donde la carga puede ser divida.

El punto de división más evidente es el AST semántico, que separa todo el proceso de *parsing* de las fases de verificación semántica y generación de código. Si se define primero una jerarquía para el AST y se usa el patrón *Visitor*, es posible dividir el trabajo al menos en 2 fases independientes: *parsing* y chequeo semántico. Mientras un miembro del equipo construye la gramática y el *parser*, otro miembro puede implementar toda la verificación semántica, sin estorbarse mutuamente, pues la interfaz de comunicación es el AST. Así mismo, un tercer miembro puede a partir del AST semántico implementar toda la generación de código, asumiendo que la verificación semántica es correcta, sin importar que aún no esté implementada.

Para la fase de generación de código, recomendamos que se utilice un lenguaje intermedio, similar a **CIL**. Esto permitirá nuevamente dividir el trabajo en 2 fases bien separadas. Primero se define un AST de este lenguaje intermedio, que debe ser mucho más sencillo que un AST para **HULK**. A partir de este punto, 2 personas diferentes pueden trabajar en 2 tareas: la transformación del AST semántico al AST de **CIL**, y la transformación de este AST a MIPS.

De modo que existen al menos 4 tareas del compilador que pueden implementarse en paralelo, estableciendo como interfaces de comunicación 2 ASTs, 1 para **HULK** y uno para **CIL**. Las tareas opcionales de inferencia de tipos y optimización de código también ser incluidas en este esquema de forma no disruptiva. Esto queda resumido en la siguiente gráfica:



Flujo de trabajo

El lenguaje **HULK** ha sido diseñado de forma que su compilador pueda implementarse *bottom-up*. Esto es, en vez de implementar cada fase del compilador (*lexer*, *parser*, semántico, generación de código) de forma secuencial, proponemos que se tomen las características del lenguaje (expresiones, funciones, clases) y se vayan adicionando, en cada momento implementando las modificaciones que cada fase requiera. A continuación proponemos un orden para implementar las características del lenguaje. En cada paso, sugerimos implementar *todas* las fases, es decir, introducir los tipos de *token* nuevos, producciones en la gramática, nodos del AST, reglas de verificación semántica y generación de código.

Para simplificar aun más la organización, proponemos dividir el desarrollo del compilador en 2 grandes fases: *frontend* y *backend*. La fase de *frontend* termina con la verificación semántica en el AST de HULK y la fase de *backend* comienza justo en la generación de código de HULK a CIL. Ambas fases pueden ser implementadas en paralelo (por personas diferentes), o en serie. En esta sección asumiremos que estas 2 fases se realizan en serie, y por tanto primero ejemplificaremos como implementar todo el *frontend* y luego todo el *backend*, pero es importante recordar que ambas fases son prácticamente independientes y se pueden ir desarrollando a la par.

La ventaja de comenzar de esta manera, es que muy rápidamente se tocan todos los puntos claves del *frontend* del compilador y se comienza a trabajar en todas las fases, aunque en cada una es muy sencillo lo que debe implementarse. Esto no quiere decir que más adelante no sea necesario regresar y revisar decisiones de diseño que en este punto no fueron previstas, pero eso es inevitable en cualquier caso. Al obligarse a comenzar el proyecto implementando un primer prototipo funcional *completo*, habrás garantizado "chocar" con la mayoría de los obstáculos temprano.

Veamos entonces una propuesta de organización.

Paso 1: Expresiones Aritméticas

Implementar los operadores `+`, `-`, `*`, `/` y `%`, y el tipo `Number`. En este punto tu compilador debe ser capaz de *interpretar* programas como el siguiente:

```
(34.1 * (123.42 - 208)) / (24 + 9);
```

Para resolver este paso deberás:

- Implementar un tokenizador básico de expresiones aritméticas.
- Diseñar una gramática no-ambigüa de expresiones.
- Construir un AST con soporte para expresiones.
- Construir un visitor para interpretar el AST.

Paso 2: Funciones globales

Implementar la invocación (**solo la invocación**) a funciones globales (`print`, `parse`, `max`, `min`, `sin`, `cos`, etc.). En este punto tu compilador debe ser capaz de interpretar programas como el siguiente:

```
print(sin(2 * 3.1415) + cos(1 / (4.54 - 6.72)));
```

Para resolver este paso deberás:

- Adicionar las reglas y producciones a la gramática.
- Añadir un nodo de invocación a funciones.
- Implementar un *visitor* de verificación semántica que chequee la cantidad de argumentos pasados a una función.
- Añadir al intérprete la implementación de las funciones elementales.

Paso 3: Declaración de funciones

Implementar la declaración de funciones, potencialmente *solo* con la notación compacta. En este punto tu compilador debe ser capaz de interpretar programas como el siguiente:

```
function tan(x) -> sin(x) / cos(x);

print(tan(2 * 3.1415));
```

Para resolver este paso deberás:

- Adicionar reglas y producciones para la declaración.
- Comprobar que la cantidad de parámetros declarados coincida con la invocación.

Paso 4:

Casos de prueba