Análisis del interpretador de expresiones matemáticas: `Java Mathematical Expression Parser (JEP)`. *Extensión a funciones de* $\mathbb{C}^n$ *en* $\mathbb{C}^m$, *y a funciones definidas a trozos*

EDWIN CAMILO CUBIDES GARZÓN

UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE CIENCIAS
DEPARTAMENTO DE MATEMÁTICAS
BOGOTÁ, D.C.
Diciembre de 2003

Análisis del interpretador de expresiones matemáticas: `Java Mathematical Expression Parser (JEP)`. *Extensión a funciones de $\mathbb{C}^n$ en $\mathbb{C}^m$, y a funciones definidas a trozos*

EDWIN CAMILO CUBIDES GARZÓN

Trabajo de grado para optar al título de
Matemático

Director
ÁLVARO MAURICIO MONTENEGRO DÍAZ, M.Sc.
Matemático

UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE CIENCIAS
DEPARTAMENTO DE MATEMÁTICAS
BOGOTÁ, D.C.
Diciembre de 2003

Título en Español

Análisis del interpretador de expresiones matemáticas: `Java Mathematical Expression Parser (JEP)`. Extensión a funciones de $\mathbb{C}^n$ en $\mathbb{C}^m$, y a funciones definidas a trozos.

Title in English

Analysis of the interpreter of mathematics expressions: `Java Mathematical Expression Parser (JEP)`. Extension to functions of $\mathbb{C}^n$ in $\mathbb{C}^m$, and to functions defined to chunks.

Resumen

En este trabajo se realizo un análisis del diseño y la implementación del interpretador de expresiones matemáticas `Java Mathematical Expression Parser (JEP)`, el cual fue diseñado por el estudiante de ciencias de la computación de la universidad de Alberta en Canadá, Nathan Funk. Para la implementación de este interpretador se hizo uso del lenguaje de programación `Java`, utilizando la metodología de la programación orientada a objetos. Como resultado de este trabajo se obtuvo un nuevo orden para el conjunto de intervalos compuestos por números de máquina y con base en este orden se realizó una extensión para evaluar funciones definidas a trozos, además se da un ejemplo de la forma de como utilizar el paquete `JEP` para evaluar funciones vectoriales de variable real y compleja.

Abstract

In this work is I accomplished an analysis of the design and the implementation of the interpreter of mathematics expressions `Java Mathematical Expression Parser (JEP)`, the one which was designed by the student of sciences the computation of the university of Alberta in Canada, Nathan Funk. For the implementation of this interpreter was made use of the programming language `Java`, using the methodology of the programming oriented to objects. In the wake of this work was obtained a new order for the set of intervals composed by numbers of machine and based on this order was accomplished a extension to evaluate functions defined to chunks, furthermore is given an example of the form of as using the package `JEP` to evaluate functions vectorial of real and complex variable.

Palabras claves: Java Mathematical Expression Parser — JEP — Evaluador — Interpretador — Evaluador de expresiones matemáticas — Evaluador de funciones complejas — Evaluador de funciones vectoriales — Evaluador de funciones definidas a trozos — Orden para intervalos

Keywords: Java Mathematical Expression Parser — JEP — Evaluator — Interpreted — Evaluator of mathematics expressions — Evaluator of complex functions — Evaluator of vectorial functions — Evaluator of functions defined to chunks — Order for intervals

Nota de aceptación

Jurado
Hector Manuel Mora Escobar

Jurado
Humberto Sarria Zapata

Director
Álvaro Mauricio Montenegro Díaz

Bogotá, D.C. Febrero 5 de 2004

A mi padres, hermanos, sobrinos, amigos y especialmente a Dennis F. Guerrero.

Agradecimientos

El autor expresa sus agradecimientos a:

Álvaro Mauricio Montenegro Díaz, Matemático y director del trabajo, quien siempre me aconsejo, oriento y me apoyo en la realización de este trabajo.

Rodrigo de Castro Korgi, Matemático, quien me ayudo y oriento en la edición del trabajo y la preparación de la defensa de éste.

Henry Barragan Gómez, Licenciado en matemáticas, quien con su gusto por esta ciencia me inspiro para estudiarla y me ha apoyado durante todo el transcurso de mi carrera.

Dennis Fabiola Guerrero Ospina, Diseñadora gráfica, quien siempre me ha acompañado, apoyado y me ha brindado su mano en los buenos y malos momentos.

"*El computador es increíblemente rápido, exacto y estúpido. El hombre es increíblemente lento, inexacto y brillante. El matrimonio de los dos es una fuerza más allá de cualquier cálculo.*"

Leo Cherne (economista)

Índice general

Índice de tablas

Índice de figuras

Introducción

Durante el transcurso de mi carrera, he tomado varios cursos de programación y algunos otros en los que se hace uso de ésta para resolver problemas matemáticos de forma numérica. Generalmente siempre existía un algoritmo para resolver estos problemas; pero en el programa en el cual se implantaba este algoritmo se tenía siempre la misma deficiencia: que sí quería trabajar con funciones, estás debían estar incluidas en el programa, haciendo que dicho programa solamente fuese utilizado para un carácter académico y no de una forma profesional. Debido a lo anterior, siempre me encontré insatisfecho con el trabajo realizado en estos cursos y cuando le preguntaba a los profesores la forma de como resolver ese problema ellos me respondían: "hay que trabajar así, a pesar de tener esa deficiencia". Cuando llegue ver los cursos avanzados de la línea de informática me encontré con el mismo inconveniente, pero en este momento, ya entendía que resolver este problema no era tarea fácil, aunque tenia la convicción de que ya estaba en la capacidad de intentar dar solución a mi inquietud.

Lo anterior me motivó para tomar la decisión de que el tema de mi proyecto de grado fuese estudiar y analizar una de la posibles soluciones a ese problema. La intención inicial al desarrollar este trabajo era la de construir un sencillo interpretador de expresiones matemáticas; pero durante el diseño del interpretador me topé con que ya alguien había pensado en resolver este problema y que lo había hecho en una forma excelente, implementando esta solución en el paquete (`JEP`) diseñado en el lenguaje de programación `Java`; por lo tanto, ya no valía la pena esforzarse en diseñar un interpretador, sabiendo que ya existía uno de libre uso; de lo cual, se tomó la decisión de realizar un análisis de la construcción, diseño y uso de este paquete para poder dar a conocer esta herramienta de programación a la comunidad matemática de la universidad, que podría hallar en ésta la solución a muchos problemas que se presentan cuando se trabaja en problemas de matemática aplicada en la cual se deseé realizar la evaluación de expresiones matemáticas de forma sencilla, segura, eficiente, confiable y gratuita. Otro de los objetivos es el de realizar una extension del paquete `JEP`, implementando un ejemplo de trabajo con funciones vectoriales y el diseño de un paquete con el cual se pudiese manipular funciones definidas a trozos.

Durante el diseño del paquete para trabajar con funciones definidas a trozos, se tuvo la necesidad de implementar un orden total para intervalos, pues en computación, trabajar con conjuntos desordenados implica un gran costo en el tiempo de ejecución de los algoritmos.

También es importante tener en cuenta de que como este trabajo tiene un carácter divulgativo, entonces uno de los principales objetivos es el de poner en la red "preferiblemente en la página WEB del departamento de matemáticas de la UN" los resultados obtenidos, los programas ejemplo y los paquetes que se describen en este documento, además de este texto.

El documento está compuesto por cinco capítulos distribuidos de la siguiente manera: en el primero se realiza una introducción a la teoría de los compiladores y las herramientas para construirlos en el lenguaje de programación `Java`, en el segundo se realiza un análisis del diseño y construcción del paquete `JEP`, en el tercero un manual para el uso del paquete `JEP`, el cual se puede estudiar directamente si el interés es de sólo utilizar esta herramienta, en el cuarto se hace la definición del orden para intervalos y la demostración de que es un orden total, en el quinto se encuentra un manual para el manejo de las funciones definidas a trozos, los intervalos y un ejemplo de la extensión de `JEP` a funciones vectoriales. Además de lo anterior, se cuenta con cuatro apéndices, entre los cuales se cuenta con un glosario de términos técnicos usados en `Java`, la gramática que especifica la sintaxis de `JEP`, una introducción a la teoría de los números de máquina y por último, la forma correcta de cómo se debe instalar el paquete `JEP`.

Al documento se encuentra anexo un CD que contiene el paquete `JEP`, las extensiones realizadas, los ejemplos ilustrados en el documento, el API de `Java` y el documento PDF en formato electrónico.

Capítulo 1

Introducción a los compiladores y a las herramientas para su construcción en `Java`

1.1. Introducción a los compiladores

1.1.1. Historia de los compiladores

La palabra compilador se atribuye con frecuencia a *Grace Murray Hopper*, quien visualizó la implementación de un lenguaje de alto nivel, en forma bastante precisa para la época, "como una compilación de una secuencia de subrutinas desde una librería". Los primeros esfuerzos para implementar lenguajes tipo inglés o castellano, fueron hechos de ésta manera.

Los primeros compiladores tipo traductor fueron escritos a finales de los 50's. Se ha otorgado ha FORTRAN (*formula translator*, fórmula traductora) el crédito de ser el primer lenguaje compilado con éxito. Tomó 18 años-hombre desarrollar el compilador FORTRAN, debido a que el lenguaje se estaba diseñando al mismo tiempo que se implementaba, y ya que fue por completo una primicia, el proceso de traducción no fue bien comprendido.

John Backus encabezó el equipo de diseño que inventó he implementó FORTRAN. Es difícil hoy en día apreciar lo significativo del paso dado por FORTRAN y su compilador.

En aquel tiempo, había un gran escepticismo con respecto a que un lenguaje semejante al inglés pudiera diseñarse y traducirse en algo que la máquina pudiera ejecutar eficazmente. El hecho de que los problemas pudieran escribirse mejor en lenguaje de alto nivel empleando sofisticadas estructuras de datos y estructuras de control estaban apenas comenzando a comprenderse. El refrán: *"Los verdaderos programadores utilizan lenguaje ensamblador"*; era firmemente aceptado (¡aunque no se expresaba de esta manera!) por muchas personas dedicadas a la computación en esa época. Es sabido que un buen progra-

mador de lenguaje ensamblador puede, incluso en la actualidad, escribir mejor código en algunos casos que el producido por un compilador, pero las ventajas de resolver un programa utilizando las abstracciones encontradas en lenguajes de alto nivel son muy superiores, excepto en una minoría de los casos.

El compilador FORTRAN fue diseñado para obtener el mejor código posible (es decir, que se ejecute en forma rápida) y así superar el escepticismo con respecto a que un lenguaje tipo inglés que seria ineficiente. FORTRAN se diseño con la necesidad de probarse a sí mismo, de modo que muchas de las características del lenguaje reflejan su diseño para la eficiencia. FORTRAN carece de las sofisticadas estructuras de datos (por ejemplo: registros, conjuntos, apuntadores, tipos enumerados) de los lenguajes más recientes como Pascal, Ada, C, Java; además, la recursividad no está en FORTRAN (debe simularse haciendo uso de una estructura de pila).

Ahora podemos diseñar e implementar un compilador mejor y más rápido por muchas razones. En primer lugar, los lenguajes se comprenden mejor. El lenguaje FORTRAN fue diseñado al mismo tiempo que su compilador. En segundo, se han desarrollado herramientas que facilitan algunas de las tareas realizadas (en particular, para las primeras fases de un compilador). En tercer lugar a través de los años se han desarrollado estructuras de datos y algoritmos para realizar tareas que son comunes a los compiladores. Finalmente, la experiencia con lenguajes y compiladores ha dado como resultado el diseño y características de lenguaje, adecuadas para las cuales es más fácil escribir un compilador.

1.1.2. Lenguajes en computación

Lenguajes de programación

Un lenguaje de programación permite al usuario crear programas que serán entendidos por el ordenador (directa o indirectamente) con el objetivo de realizar alguna tarea.

A grandes rasgos se pueden clasificar los lenguajes de programación en tres categorías: máquina, bajo nivel (ensamblador) y alto nivel.

Lenguaje de máquina

Los lenguajes de máquina son aquellos cuyas instrucciones son directamente entendibles por el ordenador sin necesidad de traducción alguna. Sus instrucciones no son más que secuencias de ceros y unos (bits). Estas especifican la operación a realizar, los registros del procesador y celdas de memoria implicados, etc.

Obviamente, este tipo de lenguajes serán fáciles de comprender para un ordenador pero muy difíciles para el hombre. Esta razón es la que lleva a buscar otro lenguaje para que los hombres se puedan comunicar con el ordenador.

Lenguajes de bajo nivel (ensamblador)

La programación en lenguaje máquina es difícil, por ello se necesitan lenguajes que permitan simplificar este proceso. Los lenguajes de bajo nivel han sido diseñados para este fin.

Estos lenguajes son generalmente dependientes de la máquina, es decir, dependen de un conjunto de instrucciones especificas del ordenador. Un ejemplo de este tipo de lenguajes es el *ensamblador*. En él, las instrucciones se escriben en códigos alfabéticos conocidos como mnemotécnicos (generalmente, abreviaturas de palabras inglesas).

Las palabras mnemotécnicas son mucho más fáciles de recordar que las secuencias de ceros y unos. Una instrucción típica de ensamblador puede ser: `ADD x,y,z`; ésta instrucción significaría que se deben sumar los números almacenados en las direcciones de memoria `x` y `y`, y almacenar el resultado en la dirección `z`. Pero aún así, a medida que los programas crecen en tamaño y complejidad, el ensamblador sigue sin ser una buena solución. Se pasa entonces así a los lenguajes de alto nivel.

Lenguajes de alto nivel

Los lenguajes de alto nivel son aquellos en los que las instrucciones o sentencias son escritas con palabras similares a las de los lenguajes humanos (en la mayoría de los casos, el Inglés). Esto facilita la escritura y comprensión del código al programador. Existen muchos lenguajes de alto nivel, por citar algunos:

- Ada
- SmallTalk
- Cobol
- Fortran
- Modula-2
- Pascal
- C
- C++
- Visual Basic
- Visual C++
- Delphi
- Java

Los programas escritos en lenguaje de alto nivel no son entendibles directamente por la máquina. Necesitan ser traducidos a instrucciones en lenguaje máquina que entiendan los ordenadores. Los programas que realizan esta traducción se llaman compiladores, y los programas escritos en lenguajes de alto nivel se denominan programas fuente.

Programas fuente y objeto

Un programa puede ser definido como un conjunto de instrucciones que pueden someterse como unidad a un ordenador y utilizarse para dirigir el comportamiento de éste.

Un *programa fuente* es aquel que nos permite escribir un algoritmo mediante un lenguaje formal. Por eso, al código desarrollado al realizar la programación se le llama código fuente. Un *programa objeto* es el resultado de traducir un programa fuente para obtener un lenguaje comprensible por la máquina.

1.1.3. Traductores de programas

Los traductores son un tipo de programas cuya función es convertir el código de un lenguaje en otro. Por ejemplo un compilador, que traduce código fuente en código objeto.

Existen distintos tipos de traductores, entre ellos destacan:

- Ensambladores
- Preprocesadores
- Intérpretes
- Compiladores

Ensambladores

Es un tipo de traductor que convierte programas escritos en lenguaje ensamblador a programas escritos en código de máquina.

Preprocesadores

Traduce un lenguaje de alto nivel a otro, cuando el primero no puede pasar a lenguaje de máquina directamente.

Interpretes

Son programas que ejecutan instrucciones escritas en un lenguaje de alto nivel. Existen varias formas de ejecutar un programa y una de ellas es a través de un interprete. Un interprete traduce instrucciones de un lenguaje de alto nivel a una forma intermedia la cual es ejecutada. Por el contrario los compiladores traducen instrucciones de alto nivel directamente a lenguaje de máquina. La ventaja de un interprete es que éste no necesita pasar por todas las etapas de un compilador para generar lenguaje de máquina, pero éste proceso puede ser consumidor de tiempo si el programa es muy grande. Por otro lado, el interprete puede ejecutar inmediatamente programas en lenguajes de alto nivel.

Compiladores

Son programas que traducen código fuente a programas escritos en código objeto. El compilador deriva su nombre de la manera en que trabaja, buscando en todo el código fuente, recolectando y reorganizando las instrucciones. Un compilador difiere de un interprete en que el interprete toma cada línea de código, la analiza y ejecuta, mientras que el compilador mira el código por completo.

Los compiladores requieren de un tiempo antes de poder generar un ejecutable, sin embargo los programas creados con compiladores se ejecutan mucho mas rápido que el mismo programa ejecutado con un interprete.

Dado que los compiladores traducen código fuente a código objeto, el cual es único para cada tipo de máquina, existen múltiples compiladores para un mismo lenguaje. Por ejemplo: el lenguaje C tiene un compilador para PC, otro para Apple Macintosh; además existen muchas casas que desarrollan compiladores para una misma plataforma y un mismo lenguaje, por ejemplo Microsoft y Borland tienen compiladores propios para C.

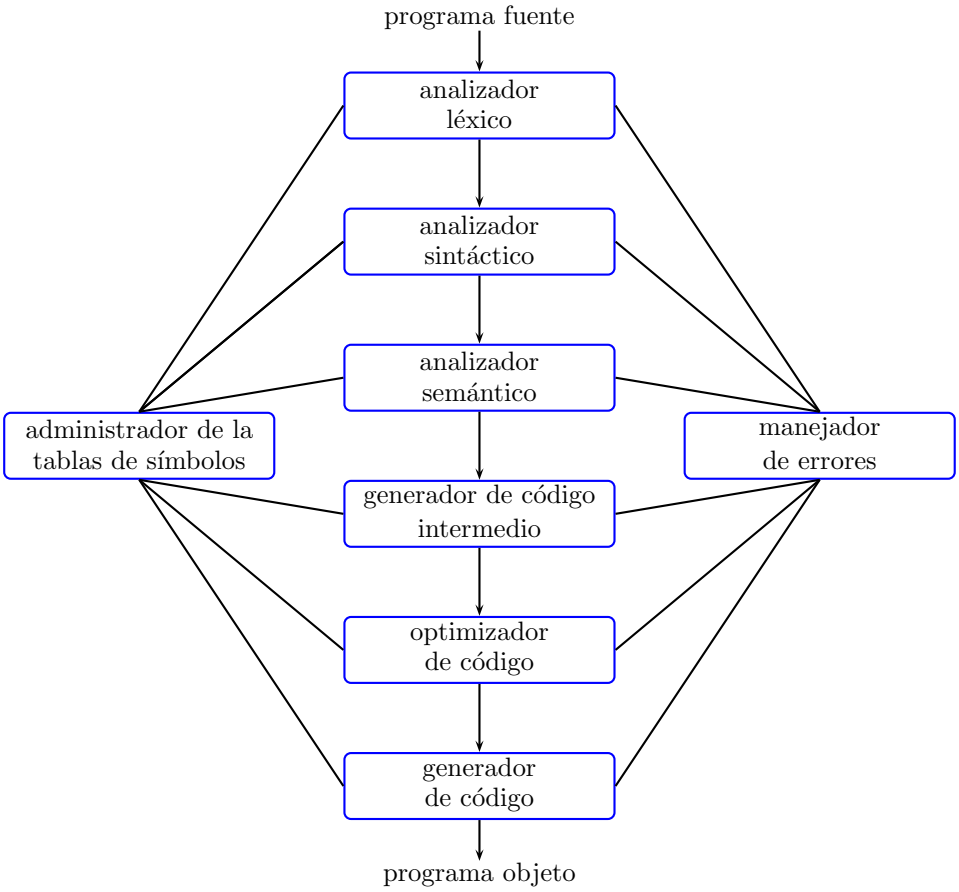


Figura 1.1: Fases de un compilador.

En la compilación se pueden detectar dos grandes fases muy bien diferenciadas, la fase del *análisis* y la fase de la *síntesis*, además de estas fases, un compilador realiza las tareas de administrar la tabla de símbolos y realizar el manejo de errores. Durante el análisis se determinan las operaciones que implica el programa fuente y se registran en una estructura jerárquica llamada árbol. A menudo se usa una clase especial de árbol llamado *árbol de sintaxis abstracta* (AST) (también se le conoce como *árbol de sintaxis abstracta* `(AAS)`), donde cada nodo representa una operación y los hijos son los argumentos de la operación.

La fase de análisis se encuentra compuesta por tres fases: El análisis léxico, el análisis sintáctico y el análisis semántico.

La parte de síntesis construye el programa objeto deseado a partir de la representación intermedia; esta fase también esta compuesta por otras tres fases: Generación de código intermedio, optimización del código y generación de código.

Metalenguajes

Un metalenguaje describe a otro lenguaje o algunos aspectos de otro lenguaje. Existen diversos metalenguajes que son muy conocidos, las expresiones regulares para describir tokens, la notación BNF (Forma de Backus-Naur) para describir la sintaxis de los lenguajes de programación o las gramáticas atribuidas para describir la semántica de los lenguajes añadiendo atributos y funciones semánticas a la notación BNF. Algunos metalenguajes pueden usarse para describirse a ellos mismos.

Las principales propiedades que debería tener un metalenguaje son:

- Fácil de leer, aprender y usar.
- Integrado, que tenga notaciones similares para las funciones a través de las diferentes partes del generador. Por ejemplo, el símbolo "=" debería ser usado como operador de asignación para la parte de las expresiones regulares, la sintaxis y funciones semánticas.
- De forma análoga a los lenguajes de programación, no debería incorporar construcciones que permitan cometer errores fácilmente.

1.1.4. Fases en la construcción de un compilador

Análisis lexicográfico

El analizador lexicográfico también conocido como *lexicador* o *escáner*, es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada, agruparlos en secuencias de caracteres que forman un componente léxico (lexemas), clasificar estos en categorías y elaborar como salida una secuencia de componentes léxicos (tokens), en el desarrollo de este trabajo realmente se pueden distinguir dos tareas separadas: *rastreo* y *filtrado*.

En el *rastreo* se mueve un apuntador a través de la entrada un carácter a la vez para hallar cadenas continuas de caracteres, las cuales constituyen elementos textuales individuales, y clasifica cada una de acuerdo con su tipo. En este punto cada vez que se encuentra un lexema, se crea un nuevo token, el cuál esta compuesto por su **tipo** de acuerdo a su clasificación y los **atributos** o **valores** que representen el lexema correspondiente; un token puede ser representado de la siguiente forma: **Token = (Tipo, Atributos)**.

En el *filtrado* se descartan algunos de los tokens encontrados por el rastreador (espacios en blanco, saltos de línea, tabulaciones y comentarios), además se determinan cuales otros son símbolos reservados (palabras clave, identificadores, operadores, etc), y coloca los demás tokens en la tabla de nombres.

El analizador lexicográfico pasa al analizador sintáctico cada uno de los tokens encontrados. En ocasiones, lo que es pasado es un apuntador a la tabla que contiene el valor del token. Puede haber tablas separadas para nombres, constantes numéricas, cadenas de caracteres constantes y operadores, o puede haber una tabla que incluya todo lo anterior.

Reconocimiento de componentes léxicos: Como el problema del cual se encarga el analizador lexicográfico es: Dada una cadena de caracteres, dividir esta cadena en los componentes léxicos que la conforman. Existen tres formalismos o modelos comunes para la descripción de los tipos de tokens:

1. *Expresiones regulares.* Las expresiones regulares describen los tokens como el conjunto de cadenas permitidas en un lenguaje.

2. *Diagramas de transición.* Los diagramas describen los tokens como las cadenas permitidas que tomen el diagrama desde un estado inicial hasta un estado final o de aceptación.

3. *Gramáticas regulares.* Las gramáticas regulares describen los tokens como las cadenas generadas por una Gramática Independiente del Contexto (GIC).

Análisis sintáctico o gramatical

El análisis sintáctico también conocido como *parsing* o *parser*, es el proceso de determinar si una cadena de componentes léxicos puede ser generada por una gramática. En el estudio de este problema, es útil pensar en construir un árbol de sintaxis abstracta, aunque, de hecho, un compilador no lo construya. Sin embargo, un analizador deberá poder construir el árbol, pues de otro modo, no se puede garantizar que la traducción sea correcta.

La mayoría de los métodos de análisis sintáctico se encuentran clasificados en dos clases: los métodos *descendentes* y *ascendentes*. Esto términos hacen referencia al orden en que se construyen los nodos del árbol de sintaxis abstracta. En el primero, la construcción se inicia en la raíz y avanza hacia las hojas, mientras que en el segundo, la construcción se inicia en las hojas y avanza hacia la raíz.

Gramáticas: Las gramáticas describen lenguajes, los lenguajes naturales como el español, o el inglés, son a menudo descritos por una gramática que agrupa palabras en categorías sintácticas tales como sujetos, predicados, frases preposicionales, etc.

Matemáticamente, una gramática es un dispositivo formal para especificar un lenguaje potencialmente infinito, en una manera finita, puesto que es imposible enumerar todas la cadenas de caracteres en un lenguaje, ya sea un lenguaje natural (español, inglés) o un lenguaje de programación (C, Pascal, Java). Al mismo tiempo, una gramática impone una estructura de sentencias en el lenguaje. Es decir, una gramática G, define un lenguaje $L(G)$ mediante la definición de una manera para derivar todas las cadenas en el lenguaje, es decir con una gramática se desea poder describir un lenguaje especifico, por lo tanto una gramática es un *metalenguaje*.

En la construcción de compiladores existe una clase particular de gramáticas denominadas gramáticas independientes del contexto (GIC), con éstas se puede verificar si, dada una cadena de componentes léxicos, entonces dicha cadena es sintácticamente correcta, de acuerdo a la construcción especificada en el lenguaje de programación en estudio.

Análisis sintáctico descendente: Como se explicó anteriormente en el análisis sintáctico descendente se puede considerar como un intento de construir un árbol de sintaxis abstracta para la entrada comenzando desde la raíz y creando los nodos del árbol en orden previo. Este tipo de analizadores frecuentemente se consideran como analizadores predictivos puesto que, en cualquier etapa, intentan predecir el nivel inferior siguiente del árbol. Esta predicción se hace examinando el siguiente token en la entrada y el árbol actual, eligiendo la producción para el siguiente nodo hijo.

Análisis sintáctico ascendente: En el análisis sintáctico ascendente la construcción del árbol sintaxis abstracta se realiza de abajo hacia arriba, es decir desde la hojas hasta la raíz. En este tipo de análisis se toma la cadena de componentes léxicos y se leen de derecha a izquierda, tomando el token que se desea analizar, intentando saber cual es el token padre de los que se encuentren a la izquierda de éste, así se forma el árbol si la cadena es sintácticamente correcta.

Análisis Semántico

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase del análisis sintáctico para identificar los operadores, operandos de expresiones y proposiciones. Con base en el árbol de sintaxis abstracta construido, en el cual los operadores aparecen como los nodos interiores y los operandos de un operador son los hijos del nodo para ese operador. El trabajo que se realiza en el análisis semántico mediante el uso de este árbol se denomina: traducción dirigida por la sintaxis.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del

lenguaje fuente.

Traducción dirigida por la sintaxis: Para traducir una construcción de un lenguaje de programación, un compilador puede necesitar tener en cuenta muchas características, además del código generado para construcción. Por ejemplo, puede ocurrir que el compilador necesite conocer el tipo de la construcción, la posición de la primera instrucción del código objeto o el número de instrucciones generadas. Por tanto, los *atributos* pueden representar cualquier cantidad, por ejemplo: un tipo, una cadena, una posición de memoria o cualquier otra cosa.

El formalismo llamado traducción dirigida por la sintaxis, sirve para especificar la traducción de una construcción de un lenguaje de programación en función de los atributos asociados con sus componentes sintácticos.

Generación de código intermedio

Después del análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa (que ya puede ejecutarse) para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir al programa objeto.

Optimización de código

La fase de optimización cambia el código intermedio transformándolo en una forma de la que se pueda producir código objeto más eficiente, es decir, en la fase final de generación de código se producirá código que se ejecute más rápido o que ocupe menos espacio (o ambas cosas). Las exigencias tradicionalmente impuestas a un compilador son duras. El código de salida debe ser correcto y de gran calidad, lo que significa que debe utilizar de forma eficaz los recursos de la máquina objeto. Además el generador de código con el que se éste construyendo el compilador también debe ejecutarse eficientemente.

Matemáticamente, el problema de generar código óptimo es "**indecidible**"[1]. En la practica, hay que conformarse con técnicas heurísticas que generan código bueno pero no siempre óptimo. La elección de las heurísticas es importante, ya que un algoritmo de generación de código cuidadosamente diseñado puede producir fácilmente código que sea varias veces más rápido que el producido por un algoritmo diseñado precipitadamente.

[1]Un problema en computación se dice "*indecidible*", sí no existe un algoritmo para resolver dicho problema. Para más detalles *véase* [JHM02].

Generación de código

La fase final de un compilador es la generación de código objeto, en ésta se traduce la representación intermedia a código de máquina (generalmente se usa un lenguaje ensamblador). Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a registros.

Administración de la tabla de símbolos

Una función esencial es registrar los identificadores utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, su tipo, su ámbito (la parte del programa donde tiene validez) y, en caso de nombres de procedimientos, cosas como el número y tipos de sus argumentos, el método de pasar cada argumento (por ej. por referencia o por valor) y el tipo que devuelve, si lo hay.

Una *tabla de símbolos* es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos del identificador. La estructura de datos permite encontrar rápidamente el registro de cada identificador y almacenar o consultar rápidamente datos de ese registro.

Manejador de errores

En cada fase en que se desarrolla un compilador se pueden encontrar errores. Sin embargo, después de detectar un error, cada fase debe tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de más errores en el programa fuente. Un compilador que se detiene cuando encuentra el primer error, no resulta tan útil como debiera.

Para el manejo de errores se pueden caracterizar cuatro procedimientos que debe realizar un compilador para tratar este problema:

- Creación de errores.
- Detección de errores.
- Informe de errores.
- Recuperación de errores.

La mayor parte de los compiladores gastan una gran cantidad de tiempo y espacio manejando los errores. Las fases de análisis sintáctico y semántico por lo general manejan una

gran porción de los errores detectables por el compilador. La fase léxica puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje. Los errores donde la cadena de componentes léxicos violan las reglas de estructura (sintaxis) del lenguaje son determinados por la fase de análisis sintáctico. Si se detecta una construcción incorrecta de la estructura sintáctica de un programa, en el análisis semántico se debe detectar este error; generalmente los errores de este tipo son los de uso de variables no declaradas y un incorrecto uso de las operaciones implícitas en el programa fuente.

1.2. Herramientas para la construcción de compiladores en `Java`

1.2.1. Una breve historia de `Java`

[2] En 1990, Sun Microsystems, Inc. financió un proyecto de investigación corporativo llamado Green, cuyo fin era el de desarrollar software destinado a electrónica de consumo. Para desarrollar este proyecto se conformo un equipo de investigación al cual pertenecían: *Patrick Naughton*, *Chris Warth*, *Ed Frank*, *Mike Sheridan* y *James Gosling*, este ultimo un veterano en el diseño clásico de software de red, y quien estaba a la cabeza del grupo.

En un comienzo se empezó a escribir software con C++ para incrustarlo en aparatos como tostadoras, videos y asistentes personales digitales o PDA's. El software incrustado se utilizó para crear electrodomésticos más inteligentes, añadiéndoles displays digitales o utilizando inteligencia artificial para controlar mejor los mecanismos. Sin embargo, pronto se dieron cuenta de que C++ era una herramienta inadecuada para este trabajo. C++ es lo suficientemente flexible para controlar sistemas controladores, pero es susceptible a errores que pueden detener el sistema. En particular, C++ utiliza referencias directas a los recursos del sistema y requiere, por parte del programador, seguir el rastro de cómo se manejan estos recursos lo que supone una carga significativa a los programadores. Esta carga en la gestión de los recursos del sistema es la barrera para la programación de software fiable y portátil, y fue un problema serio para la electrónica de consumo.

La solución dada a este problema fue un nuevo lenguaje llamado Oak. Oak preservó una sintaxis similar a la de C++ pero omite las características potencialmente peligrosas como referencias de recursos explícitas, La aritmética de punteros y la sobrecarga de operadores. Oak incorporaba gestión de memoria dentro del lenguaje de una forma directa, liberando al programador de concentrarse en tareas que el propio programa debería desempeñar. Para conseguir que fuese un lenguaje de programación de sistemas controladores eficiente, Oak necesitaba poder responder a eventos provenientes del exterior en cuestión de microsegundos. También era necesario que fuese portátil; ésto es, que fuese capaz de ejecutarse en

[2] Tomada del libro *La Biblia de Java*, *vease* [VPH97]

un determinado número de chips y entornos diferentes. Esta independencia del hardware podría proporcionar al fabricante de una tostadora cambiar el chip que utiliza para hacerla funcionar sin necesidad de cambiar el software. El fabricante podría utilizar también partes del mismo código que utiliza la tostadora para funcionar, digamos en un horno con gratinador; esto podría reducir costos, tanto de desarrollo como de hardware, aumentando también su fiabilidad. Al tiempo que Oak maduraba, la World Wide Web se encontraba en su periodo de dramático crecimiento y el equipo de desarrollo de Sun, se dio cuenta de que Oak era perfectamente adecuado para la programación en Internet. En 1994, completaron su trabajo en un producto conocido como WebRunner, un primitivo visor Web escrito en Oak. WebRunner se renombró posteriormente como Hotjava y demostró el poder de Oak como herramienta de desarrollo en Internet. HotJava es bien conocido en la industria. Finalmente, en 1995, Oak se cambió por `Java` (*por razones de marketing*) y se presentó en SunWorld 1995. Incluso antes de la primera distribución del compilador de `Java` en Junio de 1996, `Java` era ya considerado como estándar en la industria para la programación en Internet.

En los primeros seis meses de 1996, existían ya un gran número de compañías líderes, tanto de software como de hardware, con licencia de Sun de la tecnología `Java`. Entre ellas podríamos incluir Adobe, Asymetrix, Borland, IBM, Macromedia, Metrowerks, Microsoft, Novell, Oracle, Spyglass y Symantec. Estas y otras licencias de `Java`, incorporan a `Java` en sus productos de escritorio, sistemas operativos y herramientas de desarrollo.

1.2.2. Herramientas para la construcción de compiladores

Poco después de escribirse el primer compilador, aparecieron sistemas para ayudar en el proceso de escritura de compiladores. A menudo se hace referencia a estos sistemas como: *compiladores de compiladores*, *generadores de compiladores o sistemas generadores de traductores*. En gran parte, se orientan en torno a un modelo particular de lenguaje, y son más adecuados para generar compiladores de lenguajes similares al del modelo.

Por ejemplo, es tentador suponer que los analizadores léxicos para todos los lenguajes son en esencia iguales, excepto por las palabras clave y signos particularmente que se reconocen. Muchos compiladores de compiladores de hecho producen rutinas fijas de análisis léxico para usar en el compilador generado. Éstas rutinas sólo difieren en las listas clave que reconocen, y esta lista es todo lo que debe proporcionar el usuario. El planteamiento es válido, pero puede no ser funcional si se requiere que reconozca componentes léxicos no estándares, como identificadores que pueden incluir ciertos caracteres distintos de letras y dígitos.

Se han creado algunas herramientas generales para el diseño automático de componentes específicos de compilador. Estas herramientas utilizan lenguajes especializados para especificar e implantar el componente, y pueden utilizar algoritmos bastante complejos. Las herramientas más efectivas son las que ocultan los detalles del algoritmo de generación

y producen componentes que se pueden integrar con facilidad al resto del compilador.

Generadores de analizadores léxicos

Una forma sencilla de crear un analizador léxico consiste en la construcción de un diagrama que ilustre la estructura de los componentes léxicos del lenguaje fuente, y después hacer la traducción "a mano" del diagrama en un programa para encontrar los componentes léxicos, de esta forma, se pueden producir analizadores léxicos eficientes.

Escribir analizadores léxicos eficientes a mano puede resultar una tarea tediosa; para evitar este trabajo se han creado herramientas de software que generan automáticamente analizadores léxicos. Éstos generan un analizador léxico a partir de una especificación provista por el usuario a partir de una especificación basada en expresiones regulares o autómatas finitos.

Puede asegurarse que la herramienta del tipo mencionado mas conocida es `Lex`[3]. `Lex` es un generador de analizadores léxicos, originalmente incluido dentro del ambiente de desarrollo de UNIX usando a C como lenguaje huésped y posteriormente migrado a casi todas las plataformas y lenguajes. Otra herramienta que últimamente ha tenido gran difusión es `JTLex`, que usa a `Java` como lenguaje huésped y corresponde al compilador de compiladores `JavaCup`; mientras que algunos compiladores de compiladores actuales como `JavaCC` y `Eli`, integran la especificación del análisis léxico sin brindar un módulo específico. Todas estas herramientas para generar analizadores léxicos permiten definir la sintaxis de los símbolos mediante expresiones regulares, mientras que sus atributos deben ser computados luego del reconocimiento de una subcadena que constituya un símbolo del lenguaje, analizándola. `JTLex` en cambio permite expresar conjuntamente sintaxis y semántica al estilo de la traducción dirigida por la sintaxis. A su vez el proceso de cómputo de atributos es implementado por `JTLex` por un autómata finito traductor con las ventajas de eficiencia que esto supone. `JTLex` sigue el estilo de `Lex`, con la variante de que se basa en expresiones regulares. Cuando `JTLex` reconoce una subcadena va generando acciones apareadas a sus caracteres, se puede por lo tanto pensar en la secuencia de acciones generada como una traducción de la subcadena reconocida. La traducción se produce teniendo como alfabeto de entrada al conjunto de caracteres y como alfabeto de salida al conjunto de acciones. Por este motivo al tipo de formalismos que sustentan a `JTLex` se lo ha denominado expresiones regulares traductoras (ET). Una expresión regular traductora (ET) es una expresión regular en la cual los términos están constituidos por pares carácter-acción. Dentro de las ET podemos caracterizar las expresiones regulares con traducción única (ETU), aquellas a las que a cada cadena de entrada corresponde una única cadena de salida. En la figura 1.2 se ilustra como actúa un generador de analizadores léxicos, en particular `JTLex`, en éste el usuario realiza la especificación del analizador léxico y luego de que `JTLex` realiza su trabajo, se obtiene el analizador léxico, implementando el algoritmo

[3]Para detalles del uso de esta herramienta *véase* [ASU90].

general y la estructura de datos necesaria.

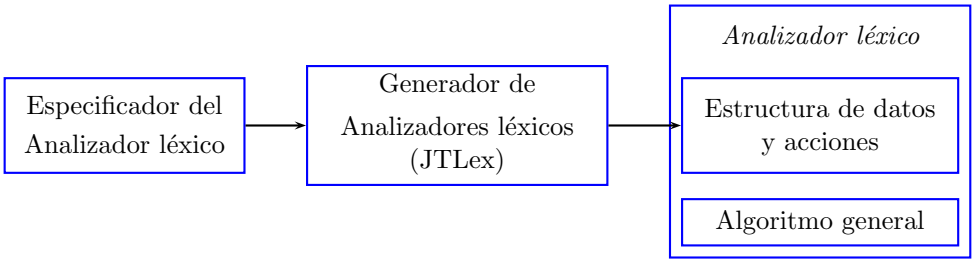


Figura 1.2: Entrada y salida del generador de analizadores léxicos `JTLex`

Generadores de analizadores sintácticos

Estos generadores producen analizadores sintácticos, normalmente a partir de una entrada fundamentada en una gramática independiente del contexto. En los primeros compiladores, el análisis sintáctico consumía no sólo parte del tiempo de ejecución del compilador, sino gran esfuerzo intelectual para escribirlo. Esta fase se considera actualmente una de las más fáciles de aplicar. Muchos de los generadores de analizadores sintácticos utilizan poderosos algoritmos de análisis sintáctico, que son demasiado complejos para realizarlos manualmente.

Uno de los analizadores sintácticos más utilizados es `YACC`[4] (*Yet Another Compiler-Compiler*, "aún otro compilador de compiladores"), el cual casi llegó a dominar la escena de los generadores de compiladores de manera tal, que muchas herramientas usan un lenguaje de especificación parecido al suyo, y otras son clones de `YACC` para la generación de compiladores con otros lenguajes de programación. Después del surgimiento de `YACC` se ha avanzado mucho en cuanto a lenguajes de programación, sistemas operativos, tecnologías de hardware y procesamiento de lenguajes; por tanto las razones que justificaron decisiones de diseño dependientes de la disponibilidad de recursos en alguna de estas áreas en la época de su surgimiento, son anacrónicas en momentos actuales.

Se basa en el modelo puro de gramáticas independientes del contexto, aunque permite el acceso a la pila semántica del analizador sintáctico, que es $LALR(1)$ con reglas de desambiguación. Éste toma una descripción concisa de una gramática y produce rutinas en C que pueden analizar esta gramática. La especificación es monolítica, y siguiendo la filosofía de UNIX de especialización de las herramientas, sólo genera el analizador sintáctico, el lexicográfico se debe generar por separado con `LEX`, su compañero inseparable, o programarlo por sí mismo. Normalmente `YACC` no es más rápido que uno generado manualmente, pero

[4]Escrito por S.C. Johnson en 1974. Para detalles del uso de esta herramienta *véase* [ASU90].

es más fácil de escribir y modificar.

`JavaCC`[5] es una nueva herramienta escrita en `Java` y creada por Sun Microsystems que genera analizadores sintácticos con los analizadores léxicos incluidos. `JavaCC` sigue el estilo de herramientas independientes, cuya interacción es por la línea de comandos. A diferencia de `YACC` no hay un solo símbolo inicial no-terminal. En `JavaCC`, se puede iniciar el análisis sintáctico con respecto a cualquier no-terminal en la gramática.

La producción BNF(forma Backur-Naur), es la producción estándar utilizada en la especificación de gramáticas en `JavaCC`. Cada producción BNF tiene un lado izquierdo, el cual es la especificación de un no-terminal; la producción BNF después define este no-terminal en términos de expansiones BNF en el lado derecho de la producción. En `JavaCC` los no-terminales son escritos exactamente como un método declarado en `Java`. El nombre del no-terminal es el nombre del método, y los parámetros y valores de retorno declarados son los mismos que pasan por arriba y abajo del árbol de sintaxis abstracta.

`JavaCC` usa expresiones regulares para el análisis lexicográfico; para el análisis sintáctico genera un **analizador sintáctico recursivo** $LL(k)$, utilizando gramáticas $LL(k)$. En los analizadores sintácticos generados por `JavaCC` se toman decisiones en puntos de selección basados en alguna exploración de componentes léxicos por adelantado en la cadena de entrada y entonces deciden que camino seguir, es decir, ningún retroceso es ejecutado, sólo una decisión es hecha. En un analizador sintáctico $LL(k)$, la k representa los componentes léxicos que se deben tomar por adelantado para llevar a cabo el análisis[6], y en `JavaCC` la k puede ser tan grande como sea necesario. Las gramáticas incluyen la operación de clausura y la estructura opcional, pudiéndose insertar las acciones semánticas en cualquier lugar de la producción. Para el reconocimiento usa "lookahead" local, que puede ser combinado con "lookahead" sintáctico y "lookahead" semántico. En la implementación se usa el método recursivo descendente, generándose una clase de `Java` la cual contendrá un método publico por cada no-terminal definido en la gramática.

Las acciones semánticas pueden escribirse en `Java` o cualquier extensión a este lenguaje, ya que el generador sólo chequea los delimitadores de las acciones semánticas. No hay necesidad de separar la especificación del analizador lexicográfico del sintáctico, aunque de estar presentes en el archivo, puede generarse código sólo para una de ellas si así se desea. Si se desea depurar errores en el analizador, en el momento de generarlo esto debe ser especificado. Esta hace que se imprima una traza de la ejecución, que puede pedirse para el analizador lexicográfico o el sintáctico de forma no exclusiva. En el caso del analizador sintáctico, puede pedirse que se incluyan en la traza las acciones de las operaciones de "lookahead".

`JavaCC` por defecto chequea la existencia de recursividad izquierda, ambigüedad con un "lookahead" limitado, uso de expansiones en la cadena vacía, entre otros. Opcionalmente

[5]Para una obtener la documentación completa de esta herramienta visite la página web http://javacc.dev.java.net/.

[6]A este procedimiento también se le conoce como "lookahead".

puede pedirse que se chequee la ambigüedad, pasando por alto las especificaciones de "lookahead" de la gramática.

JJTree de JavaCC

`JJTree` es un preprocesador para `JavaCC` que inserta árboles de sintaxis abstracta (AST) integrando acciones en varios lugares en la fuente `JavaCC`. La salida de `JJTree` es ejecutada a través de `JavaCC` para crear el analizador sintáctico.

Por defecto `JJTree` genera código para construir los nodos del AST para cada no-terminal en el lenguaje. Este comportamiento puede ser modificado de manera que algunos no-terminales no tengan nodos generados.

`JJTree` define una interface `Java` llamada `Node` que todos los nodos del AST deben implementar. La interface provee métodos para operaciones tales como: configurar el padre del nodo, agregar hijos y recuperarlos, además de que a cada nodo se le puede asociar un conjunto de atributos como un tipo, una cadena, una posición de memoria o cualquier otra cosa. De esta manera con `JJTree` se puede manipular al árbol como se desee, lo cual facilita llevar a cabo algunas tareas tales como una *definición dirigida por la sintaxis*.

En la figura 1.3 se muestra la representación del árbol obtenido al analizar sintácticamente la expresión matemática $u(x, y) = e^x \operatorname{sen} y + e^y \cos x$, el cuál se recorre para realizar el análisis semántico y la generación del código intermedio.

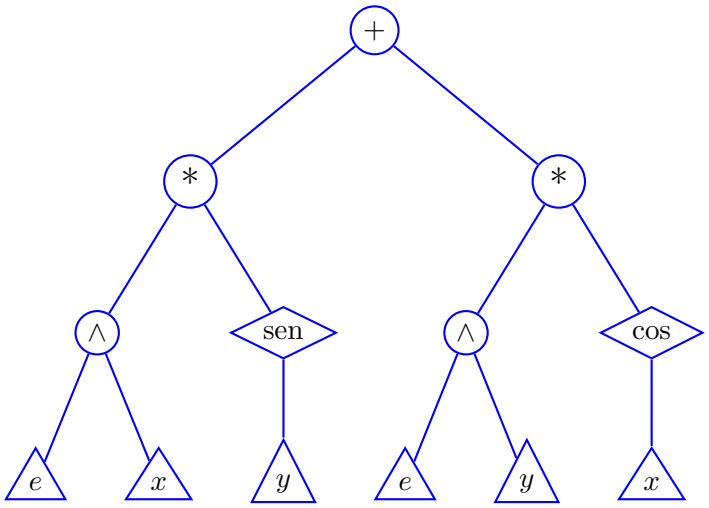


Figura 1.3: Árbol con el cual se puede representar la expresión $u(x, y) = e^x \operatorname{sen} y + e^y \cos x$

Dispositivos de traducción dirigida por la sintaxis

Estos producen grupos de rutinas que recorren el árbol de sintaxis abstracta, generando código intermedio. La idea básica es que se asocia una o más "traducciones" con cada nodo

del árbol de sintaxis abstracta, y cada traducción se define partiendo de traducciones en sus nodos vecinos en el árbol. Para el caso de `JavaCC`, éste genera la clase *ParseDumpVisitor* que sirve para recorrer el árbol de sintaxis abstracta.

Generadores automáticos de código

Tales herramientas toman un conjunto de reglas que definen la traducción de cada operación del lenguaje intermedio al lenguaje de máquina para la máquina objeto. Las reglas deben incluir suficiente detalle para poder manejar los distintos métodos de acceso posible a los datos; por ejemplo, las variables pueden estar en registros, en una posición fija (estática) de memoria o pueden tener asignada una posición en una pila. La técnica fundamental es la de "concordancia de plantillas". Las proposiciones de código intermedio se reemplazan por "plantillas" que representan secuencias de instrucciones de máquina, de modo que las suposiciones sobre el almacenamiento de la variables concuerden de plantilla a plantilla. Como suele haber muchas opciones en relación con la ubicación de las variables (por ej. en uno o varios registros o en memoria), hay muchas formas posibles de "cubrir" el código intermedio con un conjunto dado de plantillas, y es necesario seleccionar una buena cobertura sin una explosión combinatoria en el tiempo de ejecución del compilador.

Dispositivos para análisis de flujo de datos

Mucha de la información necesaria para hacer una buena optimización de código implica hacer un "análisis de flujo de datos", que consiste en la recolección de información sobre la forma en que se transmiten los valores de una parte de un programa a cada una de las otras partes. Las distintas tareas de ésta naturaleza se pueden efectuar esencialmente con la misma rutina, en la que el usuario proporciona los detalles relativos a la relación que hay entre proposiciones en código intermedio y la información que se está recolectando.

Capítulo 2

Análisis del paquete JEP

Figura 2.1: Nathan Funk

JEP fue diseñado por *Nathan Funk*. Estudiante graduado en Ciencias de la Computación en la Universidad Alberta en Canada. JEP es un Java API (Interfaz de programación de aplicaciones de Java) diseñado para la evaluación de expresiones matemáticas. Las expresiones son pasadas como argumentos de tipo cadena (*String*) y pueden ser evaluadas instantáneamente. JEP permite el uso de tantas variables como se encuentren contenidas en la expresión, además de constantes numéricas y la mayoría de las funciones reales y complejas (p. ej. sin(), log(), cosh(), arctan(), im(), etc.) y las constantes matemáticas más comúnmente utilizadas (p. ej. e, π).

JEP ha sido utilizado en un amplio rango de aplicaciones tales como: simulación, mercado de acciones, aplicaciones a la ingeniería, educación matemática entre otras. Además ha sido utilizado por una amplia gama de prestigiosas instituciones tales como: NASA Jet Propulsion Laboratory, Object Reservoir, ERIN Engineering and Research Inc., NovoDynamics Inc., Ebase Technology Ltda, ChemAxon Ltda, entre otras.

El paquete JEP se encuentra agrupado en la carpeta org.nfunk.jep la cual contiene las clases que permiten la verificación y evaluación de las expresiones matemáticas, y los paquetes org.nfunk.jep.type y org.nfunk.jep.function que permiten el manejo de diversos tipos numéricos y funciones matemáticas.

Jerarquía de los paquetes que intervienen en JEP

```
java.lang.Object
    |
    +--org.nfunk.jep
        |
        +--org.nfunk.jep.function
        |
        +--org.nfunk.jep.type
```

2.1. Descripción del paquete `org.nfunk.jep.type`

Este paquete es el que contiene las clases e interfaces que permiten el manejo del conjunto de números (reales, complejos y otros definidos por el usuario) con el cual puede trabajar `JEP`. Este paquete contiene una interfaz y dos clases que se describen a continuación:

Interfaces del paquete `org.nfunk.jep.type`	
Interfaz	Descripción
public interface *NumberFactory*	Esta interfaz es implementada para crear una fábrica de objetos de tipo numérico.

Clases del paquete `org.nfunk.jep.type`	
Clase	Descripción
public class *DoubleNumberFactory*	Esta clase implementa la interfaz *NumberFactory* y permite, por defecto, crear objetos numéricos de doble precisión.
public class *Complex*	Esta clase implementa la interfaz *NumberFactory* y sirve para representar números complejos donde las componentes real e imaginaria se representan con números de doble precisión e incluye los métodos que permiten la evaluación de funciones complejas.

La interfaz *NumberFactory* tiene el objetivo de actuar como una fabrica de números, actuando como una clase de envoltura de un tipo numérico. A continuación se describen los métodos declarados en esta interfaz.

Métodos declarados en la interfaz *NumberFactory*	
Método	Descripción
public java.lang.Object *createNumber*(double value)	Crea un objeto numérico e inicializa su valor con el parámetro *value*.

La clase *doubleNumberFactory* implementa la interfaz *NumberFactory*, actúa como una fábrica y como clase de envoltura del tipo primitivo *double*, con el fin de poder manipular este tipo como un objeto de tipo *java.lang.Object*. Pues las estructuras de datos estarán compuestas por objetos, como de *Object* se extienden todas las demás clases, entonces ésta se podrá convertir en cualquier otra clase en particular aquellas que contenga un tipo primitivo o un tipo de dato abstracto especifico; en este caso será un tipo *double*.

Constructores definidos en la clase *DoubleNumberFactory*	
Constructor	Descripción
public *DoubleNumberFactory*()	Constructor por defecto.

Métodos definidos en la clase *DoubleNumberFactory*	
Método	Descripción
public java.lang.Object *createNumber*(double value)	Crea un objeto *Double* inicializado con el parámetro *value*.

La clase *Complex* actúa como fábrica de números complejos para representar este conjunto como una pareja cuyas componentes son tipos *double*. Esta clase posee constructores de distintos formatos y una gran variedad de métodos de dos tipos; los primeros para manipular las componentes del número complejo representado, y los segundos para hacer la manipulación aritmética y la evaluación de las funciones de variable compleja básicas[1]. En la descripción de los métodos se encuentra la formula utilizada para la evaluación de cada una de las funciones[2], y se hace uso de un número complejo genérico definido como $z = re + i * im$.

Constructores definidos en la clase *Complex*	
Constructor	Descripción
public *Complex*()	Constructor por defecto, inicializa tanto la componente real como la imaginaria iguales a cero.
public *Complex*(Complex z)	Inicializa un objeto complejo al copiar las componentes real e imaginaria del objeto *z* en las correspondientes componentes del nuevo objeto.
public *Complex*(double re_in)	Inicializa un objeto complejo con su componente real igual al parámetro *re_in* y su componente imaginaria igual a cero.
public *Complex*(double re_in, double im_in)	Inicializa un objeto complejo con su componente real igual al parámetro *re_in* y su componente imaginaria igual a *im_in*.
public *Complex*(java.lang.Number re_in)	Inicializa un objeto complejo con su componente real igual al resultado retornado por el método *doubleValue*() del objeto *re_in* y su componente imaginaria igual a cero.

Métodos definidos en la clase *Complex*	
Método	Descripción
public double *re*()	Retorna la componente real de este objeto.
public double *im*()	Retorna la componente imaginaria de este objeto.
public void *set*(Complex z)	Establece las componentes real e imaginaria de este objeto como una copia de las componentes correspondientes del objeto *z*.

[1] El valor retornado es el valor principal de la función a evaluar; por ejemplo, al evaluar el argumento de un número complejo el valor retornado es un número entre $-\pi$ y π.

[2] Para ver la deducción de estas formulas *véase* [CB92], y el modo de como se pueden implementar la evaluación en un programa *véase* [AJ95].

public void *set*(double re_in, double im_in)	Establece las componentes real e imaginaria de este objeto con el valor de los parámetros *re_in* e *im_in* respectivamente.
public void *setRe*(double re_in)	Establece la componente real de este objeto como igual al valor del parámetro *re_in*.
public void *setIm*(double im_in)	Establece la componente imaginaria de este objeto como igual al valor del parámetro *im_in*.
public java.lang.String *toString*()	Retorna una cadena (*String*) con el formato de un número complejo, sobre-escribiendo el método *toString*() de la clase *Object*, con el formato: "(*re,im*)".
■ Miscelánea de funciones y operadores	
public boolean *equals*(Complex b, double tolerance)	Establece si este objeto es lo suficientemente aproximado al objeto b, para ser considerados iguales, $z \cong b$. La tolerancia es la máxima magnitud diferencia que pueden tener las componentes respectivas.
public double *abs*()	Retorna el valor absoluto o magnitud de este número complejo $\|z\|$. Se calcula como: $\|z\| = \sqrt{re^2 + im^2}$.
public double *abs2*()	Retorna el cuadrado del valor absoluto de este número complejo. Se calcula como: $\|z\|^2 = re^2 + im^2$.
public double *arg*()	Retorna el argumento principal de este número complejo $\text{Arg}(z)$. Se obtiene dicho valor al conocer la segunda componente de la forma polar (r, θ) de este número complejo, para encontrar el valor θ se calcula $\tan^{-1}(\frac{im}{re})$ y su resultado se ubica en el intervalo $(-\pi, \pi]$, para $re \neq 0$ y $z \neq 0$
public Complex *neg*()	Retorna el opuesto aditivo de este número complejo. Se calcula como: $-z = (-re, -im)$.
public Complex *mul*(double b)	Retorna un complejo que es el resultado de multiplicar el parámetro b con este número complejo. Se calcula como: $b * z = (b * re, b * im)$.
public Complex *mul*(Complex b)	Retorna el resultado de multiplicar el parámetro b con este número complejo. Se calcula como: $(re * b.re - im * b.im, im * b.re + re * b.im)$.
public Complex *div*(Complex b)	Retorna el resultado de dividir este número complejo por el parámetro b. Se calcula como: $\frac{z}{b} = \left(\frac{re * b.re + im * b.im}{b.re^2 + b.im^2}, \frac{im * b.re - re * b.im}{b.re^2 + b.im^2}\right)$.

public Complex *power*(double exponent)	Retorna el valor principal de elevar este número complejo al parámetro real *exponent* z^a. Se calcula como: $z^a = \|z\|^a * (\cos(a * \text{Arg}(z)) + i * \text{sen}(a * \text{Arg}(z)))$.
public Complex *power*(Complex exponent)	Retorna el valor principal de elevar este complejo al parametro complejo *exponent* z^w. Se calcula como: $z^w = \exp(w * \text{Log}(z))$.
public Complex *log()*	Retorna el logaritmo principal de este número complejo $\text{Log}(z)$. Se calcula como: $\text{Log}(z) = \log(\|z\|) + i * \text{Arg}(z)$.
public Complex *sqrt*()	Retorna una de las raíces cuadradas de este número complejo $\sqrt{z}$. Se puede calcular de la siguientes formas: ▪ $\sqrt{z} = \sqrt{\|z\|} * \left[\cos\left(\frac{\text{Arg}(z)}{2}\right) + i * \text{sen}\left(\frac{\text{Arg}(z)}{2}\right)\right]$. ▪ $\sqrt{z} = z^{1/2} = \exp(\frac{1}{2}\text{Log}(z))$.
▷ Funciones trigonométricas	
public Complex *sin*()	Retorna el seno de este número complejo $\text{sen}(z)$. Se calcula como: $\text{sen}(z) = \frac{e^{iz} - e^{-iz}}{2i}$.
public Complex *cos*()	Retorna el coseno de este número complejo $\cos(z)$. Se calcula como: $\cos(z) = \frac{e^{iz} + e^{-iz}}{2}$.
public Complex *tan*()	Retorna la tangente de este número complejo $\tan(z)$. Se calcula como: $\tan(z) = \frac{\text{sen}(z)}{\cos(z)} = \frac{e^{iz} - e^{-iz}}{i * (e^{iz} + e^{-iz})}$.
▷ Funciones trigonométricas inversas	
public Complex *asin*()	Retorna el arcoseno principal de este número complejo $\text{sen}^{-1}(z)$. Se calcula como: $\text{sen}^{-1}(z) = -i * \text{Log}\left[iz + \sqrt{1 - z^2}\ \right]$.
public Complex *acos*()	Retorna el arcocoseno principal de este número complejo $\cos^{-1}(z)$. Se calcula como: $\cos^{-1}(z) = -i * \text{Log}\left[z + \sqrt{z^2 - 1}\ \right]$.
public Complex *atan*()	Retorna el arcotangente principal de este número complejo $\tan^{-1}(z)$. Se calcula como: $\tan^{-1}(z) = -\frac{i}{2} * \text{Log}\,\frac{i - z}{i + z}$.
▷ Funciones trigonométricas hiperbólicas	
public Complex *sinh*()	Retorna el seno hiperbólico de este número complejo: $\text{senh}(z)$. Se calcula como: $\text{senh}(z) = \frac{e^z - e^{-z}}{2}$.

public Complex *cosh*()	Retorna el coseno hiperbólico de este número complejo: $\cosh(z)$. Se calcula como: $\cosh(z) = \frac{e^z + e^{-z}}{2}$.
public Complex *tanh*()	Retorna la tangente hiperbólico de este número complejo: $\tanh(z)$. Se calcula como: $\tanh(z) = \frac{\text{senh}(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$.
▷ Funciones trigonométricas hiperbólicas inversas	
public Complex *asinh*()	Retorna el arcoseno hiperbólico principal de este número complejo $\text{senh}^{-1}(z)$. Se calcula como: $\text{senh}^{-1}(z) = \text{Log}\left[z + \sqrt{z^2+1}\ \right]$.
public Complex *acosh*()	Retorna el arcocoseno hiperbólico principal de este número complejo $\cosh^{-1}(z)$. Se calcula como: $\cosh^{-1}(z) = \text{Log}\left[z + \sqrt{z^2-1}\ \right]$.
public Complex *atanh*()	Retorna el arcotangente hiperbólico principal de este número complejo $\tanh^{-1}(z)$. Se calcula como: $\tanh^{-1}(z) = \frac{1}{2}\,\text{Log}\,\frac{1+z}{1-z}$.

2.2. Descripción del paquete `org.nfunk.jep.function`

En este paquete se definen las clases con las cuales se realiza la evaluación de las principales funciones matemáticas que se encuentran implementadas. Éste se encuentra compuesto por una interfaz y 31 clases. En la interfaz *PostfixMathCommandI* se declaran tres métodos: *getNumberOfParameters()*, *setCurNumberOfParameters(int n)* y *run(Stack aStack)*, esta interfaz es implementada por la clase *public PostfixMathCommand* de la cual se extienden todas aquellas clases que tienen el fin de permitir la evaluación de funciones[3] ya implementadas o aquellas que no están implementadas y se desea implementar por parte del usuario, realizando el análisis necesario para poder evaluar dicha función.

Interfaces del paquete `org.nfunk.jep.function`	
Interfaz	Descripción
public interface *PostfixMathCommandI*	Todas las clases de funciones deben implementar esta interfaz para asegurar que el método *run()* está definido.

[3]Para la evaluación se hace uso de una estructura de datos con el formato de una pila.

Clases del paquete `org.nfunk.jep.function`	
Clase	Descripción
public class *PostfixMathCommand*	Esta clase es la más importante de este paquete e implementa la interfaz *PostfixMathCommandI*, además todas las clases que implementan funciones matematicas se extienden de esta clase. Ésta incluye dos atributos: *int numberOfParameters* e *int curNumberOfParameters*. El primero es verificado al realizar el parsing de la expresión, este debe ser inicializado con el número de parámetros de la función, y si el número de parámetros es arbitrario, se debe iniciar el atributo con -1, el segundo es usado para saber el número de parámetros a usar en la invocación del siguiente *run()*.
▷ Las clases siguientes se extienden de *PostfixMathCommand*	
public class *Abs*	Sirve para inicializar un objeto, el cual posee un método que sirve para hallar el valor absoluto de un objeto ubicado en el tope de la pila y su resultado es puesto de nuevo en el tope.
public class *Add*	Sirve para inicializar un objeto, el cual posee un método que sirve para sumar los dos objetos ubicados en el tope de la pila y su resultado es puesto de nuevo en el tope.
public class *Angle*	Sirve para inicializar un objeto, el cual posee un método que retorna el parámetro θ del punto (r, θ) en coordenadas polares que corresponden a punto (x, y) en coordenadas cartesianas. Se cumple que $-\pi < \theta \leq \pi$.
public class *ArcCosine*	Sirve para inicializar un objeto, l cual posee un método que sirve para evaluar el arcocoseno de un objeto.
public class *ArcCosineH*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el arcocoseno hiperbólico de un objeto.
public class *ArcSineH*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el arcoseno hiperbólico de un objeto.
public class *ArcTangent*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el arcotangente de un objeto.
public class *ArcTanH*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el arcotangente hiperbólico de un objeto.
public class *Comparative*	Sirve para inicializar un objeto, el cual posee un método que sirve para comparar números reales con respecto al orden usual (p.ej. $x < y$, $x > y$, $x \leq y$, $x \geq y$, $x = y$, $x \neq y$), comparar sí dos números complejos son iguales (p.ej. $x + i * y = a * i * b$) o sí dos cadenas son iguales.

public class *CosineH*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el coseno hiperbólico de un objeto.
public class *Divide*	Sirve para inicializar un objeto, el cual posee un método que sirve para dividir números ú objetos (reales, complejos, vectores) entre números reales o complejos .
public class *Imaginary*	Sirve para inicializar un objeto, el cual posee un método que sirve para obtener la componente imaginaria de un objeto (p.ej. *Number* o *Complex*).
public class *Logarithm*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el logaritmo base 10 de un objeto.
public class *Logical*	Sirve para inicializar un objeto, el cual posee un método que sirve para implementar la lógica booleana (p.ej $1 \wedge 1 = 1$, $1 \wedge 0 = 0$, $0 \wedge 1 = 0$, $0 \wedge 0 = 0$, $1 \vee 0 = 1$, $1 \vee 1 = 1$, $0 \vee 1 = 1$, $0 \vee 0 = 0$).
public class *Modulus*	Sirve para inicializar un objeto, el cual posee un método que sirve para retornar el residuo de la división entre dos entre dos números (p.ej. $9{,}5 \,\%5 = 4{,}5$).
public class *Multiply*	Sirve para inicializar un objeto, el cual posee un método que sirve para multiplicar números ú objetos (reales, complejos, vectores) entre números reales o complejos.
public class *NaturalLogarithm*	Sirve para inicializar un objeto, él cual posee un método que sirve para evaluar el logaritmo natural de un objeto.
public class *Not*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar si un número es igual a cero o no (si $x = 0$ se retorna 1 sino retorna 0).
public class *Power*	Sirve para inicializar un objeto, el cual posee un método que sirve para elevar la base (real o compleja) a un exponente (real o complejo).
public class *Random*	Sirve para crear una clase de envoltura de un número aleatorio obtenido con el método *Math.random()*.
public class *Real*	Sirve para inicializar un objeto, el cual posee un método que sirve para obtener la componente real de un objeto (p.ej. *Number* o *Complex*).
public class *Sine*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el seno de un objeto.
public class *SineH*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar el seno hiperbólico de un objeto.
public class *SquareRoot*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar la raíz cuadrada de un objeto.
public class *Subtract*	Sirve para inicializar un objeto, el cual posee un método que sirve para sustraer objetos.

public class *Sum*	Sirve para crear un objeto, que es un ejemplo de una de función que acepta cualquier número de parámetros, retorna el resultado de sumar todos los nodos de una pila suponiendo que cada nodo es un objeto de tipo *Double*.
public class *Tangent*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar la tangente de un objeto.
public class *TanH*	Sirve para inicializar un objeto, el cual posee un método que sirve para evaluar la tangente hiperbólico de un objeto.
public class *UMinus*	Sirve para inicializar un objeto, el cual posee un método que sirve para retornar el opuesto aditivo de un objeto dado.

Las clases definidas en el paquete `org.nfunk.jep.function` se pueden agrupar en tres grandes grupos de clases:

1. El primer grupo esta compuesto solo por la clase *PostfixMathCommand* la cual actúa como la superclase de aquellas clases que sirven para implementar las funciones definidas en el evaluador y las definidas por el usuario.

2. El segundo grupo esta compuesto por las clases que implementan aquellas funciones que utilizan un solo parámetro.

Clases que actúan sobre un solo parámetro		
• Abs	• ArcCosine	• ArcCosineH
• ArcSine	• ArcSineH	• ArcTangent
• ArcTanH	• Cosine	• CosineH
• Imaginary	• Logarithm	• NaturalLogarithm
• Random	• Real	• Sine
• Not	• SineH	• SquareRoot
• Tangent	• TanH	• UMinus

   Estas clases se encuentran implementadas de una forma estándar, a continuación se hace una explicación de como se realizo dicha implementación describiendo detalladamente cada constructor y método utilizado. Para esta explicación se tomo como ejemplo la función seno implementada como la clase *Sine*.

   Cada clase posee un solo constructor, el constructor por defecto *Sine()* en este caso. En éste se establece el valor de la variable *numberOfParameters* de la superclase *public PostfixMathCommand* igual a uno (el número de parámetros).

```
public Sine()
 {
      numberOfParameters = 1;
 }
```

Se tiene además un método que sirve para realizar la evaluación de la función implementada, si el parámetro pasado es una instancia de la clase *Number*, se utiliza el método *sin(double arg)* de la clase *Math* para realizar la evaluación y se retorna el valor obtenido, si por el contrario el parámetro es una instancia de la clase *Complex*, se utiliza el método *sin()* de la clase *Complex* para realizar la evaluación y se retorna el valor obtenido, si no es ninguna de las instancias anteriores se lanza un objeto *ParseException* con el mensaje *"Invalid parameter type"*.

```
public Object sin(Object param) throws ParseException {
   if (param instanceof Number) {
      return new Double(Math.sin(((Number)param).doubleValue()));
   } else if (param instanceof Complex) {
      return ((Complex)param).sin();
   }
      throw new ParseException("Invalid parameter type");
}
```

Como el objetivo principal de extender una clase de la clase *PostfixMathCommand* es que esta clase tenga necesariamente incorporado el método *run(Stack inStack)*, por lo tanto este método debe ser sobrescrito en cada clase y las funciones que realiza son las de chequear que la pila no este vacía, obtener el objeto del tope de la pila quitándolo de ella y luego realiza una llamada al método anterior colocando el resultado de nuevo en el tope de la pila.

```
public void run(Stack inStack) throws ParseException {
      checkStack(inStack);// check the stack
      Object param = inStack.pop();
      inStack.push(sin(param));// push the result on the inStack
      return;
}
```

Existen 5 clases que no tienen implementado exactamente este formato, las clases *Imaginary*, *Real*, *Random*, *UMinus* y *Not*. Se mantiene un constructor y el método *run()*, el que cambia es el método que sirve para realizar la evaluación el que puede o no aparecer.

- En la clase *Imaginary* si el parámetro del método que sirve para realizar la evaluación es una instancia de la clase *Number* se retorna 0, en otro caso se comporta de forma análoga al método descrito.
- En la clase *Real* si el parámetro es una instancia de la clase *Number* se retorna el valor que encapsule este objeto, en otro caso se comporta de forma análoga al método descrito.
- La clase *Random* no tiene un método para la evaluación, sino que se obtiene un número aleatorio con el método *random()* de la clase *Math*, se crea un objeto de tipo *Double* y se coloca directamente en el tope de la pila.

- En la clase *UMinus* si el parámetro es una instancia de la clase *Number* se retorna menos el valor que encapsule esta clase, si por el contrario es un objeto de la clase *complex* entonces se hace un llamado del método *neg()* de esta clase, en otro caso se lanza una excepción.
- La clase *Not* no tiene un método para la evaluación, sino que verifica si el parámetro es un objeto de la clase *Number* y verifica si el valor encapsulado es igual a 0, si es así entonces retorna 1 en cualquier otro caso retorna 0. Si el parámetro no es un objeto de este tipo entonces se lanza una excepción.

3. El tercer grupo esta compuesto por las clases que implementan aquellas funciones que utilizan dos o más parámetros.

Clases que actúan sobre dos o más parámetros				
• Add	• Angle	• Comparative	• Divide	• Logical
• Modulus	• Multiply	• Power	• Subtract	• Sum

Todas estas clases excepto *add* y *sum* son utilizadas para simular los operadores binarios más comunes, en el constructor se establece el parámetro *numberOfParameters* igual a 2, de tal manera que al sobreescribir el método *run(Stack inStack)* se obtienen los dos objetos que se encuentran en el tope de la pila y de acuerdo a la clase de la cual son instancia se realiza la operación correspondiente.

En los constructores de las clases *add* y *sum* se establece *numberOfParameters* igual a -1, que indica que el número de parámetros es arbitrario, en el método *run(Stack inStack)* de la clase *add* se hace aplica el operador "+", dependiendo si los objetos contenidos en la pila son instancias de *Number*, *Complex* o *String* en los dos primeros se hace la suma de los números y en el ultimo se hace la concatenación de cadenas. La clase *sum* es un ejemplo de como se puede implementar una clase que actúa sobre cualquier número de parámetros y su objetivo es del de sumar los objetos de la pila suponiendo que todos los objetos son instancias de la clase *Number*.

2.3. Descripción del paquete `org.nfunk.jep`

Éste es el paquete principal de `JEP`, en él se encuentran contenidos los paquetes `org.nfunk.jep.function` y `org.nfunk.jep.type`, con los cuales se hace el manejo los tipos de datos y las funciones que actúan sobre éstos.

En este paquete se encuentran las clases que permiten realizar el análisis de léxico, sintáctico y semántico. Estas clases fueron obtenidas haciendo uso de la herramientas `JavaCC` y `JJTree` que las genera automáticamente. En el archivo `Parser.jjt` se especifica la gramática[4] con la cual el analizador léxico es generado por medio de `JavaCC` y su

[4]En el apéndice B, se encuentra consignada la gramática con la cual se especifica la sintaxis del interpretador de expresiones matemáticas `JEP`.

componente `JJTree`. Al procesar este archivo se genera el archivo `Parser.jj` el cual es usado por `JavaCC` para generar la clases que permiten realizar el análisis recursivo $LL(1)$.

También en este paquete se encuentra la clase *JEP*, la cual es la más importante del API `JEP` y es la que permite la evaluación de las expresiones matemáticas. En el capítulo 3 se realiza una descripción detallada de esta clase, donde se exhiben ejemplos y se explica la forma de usar los constructores y los métodos propios de la clase, en particular *parseExpression(String)*, el cuál permite el paso de las expresiones matemáticas al evaluador, y *getValue()* para obtener el resultado de evaluar la expresión pasada.

Interfaces del paquete `org.nfunk.jep`	
Interfaz	Descripción
public interface *Node*	Generada por `JJTree`. Todos los nodos del árbol de sintaxis abstracta implementan esta interfaz. Esta provee la maquinaria básica para la construcción de las relaciones entre los nodos hijos y padres.
public interface *ParserConstants*	Es generada por `JJTree` y `JavaCC`. En esta interfaz se definen las constantes utilizadas por el analizador sintáctico.
public interface *ParserTreeConstants*	Generada por `JJTree`. En esta interfaz se definen las constates utilizadas en los nodos del árbol de sintaxis abstracta AST.
public interface *ParserVisitor*	Generada por `JJTree`. En esta interfaz se declaran los métodos con los cuales se realiza las visitas a los nodos.

Clases del paquete `org.nfunk.jep`	
Clase	Descripción
public class *ASTConstant*	Generada por `JJTree`, se extiende de *SimpleNode* e implementa la interfaz *Node*. Ésta actúa como un nodo del árbol de sintaxis abstracta, que sirve de envoltura de una constante.
public class *ASTFunNode*	Generada por `JJTree`, se extiende de *SimpleNode* e implementa la interfaz *Node*. Ésta actúa como un nodo del árbol de sintaxis abstracta, que sirve de envoltura de las funciones matemáticas definida en *JEP* y los operadores binarios $(+, -, *, /, \ldots)$.
public class *ASTStart*	Generada por `JJTree`, se extiende de *SimpleNode* e implementa la interfaz *Node*. Ésta actúa como el nodo raíz del árbol de sintaxis abstracta.
public class *ASTVarNode*	Generada por `JJTree`, se extiende de *SimpleNode* e implementa la interfaz *Node*. Ésta actúa como un nodo del árbol de sintaxis abstracta, que sirve de envoltura de una variable de la expresión matemática interpretada.

public class *FunctionTable*	Se extiende de *java.util.Hashtable*. En ésta se guarda la información de las funciones matemáticas que pueden ser utilizadas en la expresión que se quiere evaluar.
public class *EvaluatorVisitor*	Se extiende de *Object* e implementa *ParserVisitor*. Esta clase es usada para la evaluación de la expresión. Ésta utiliza un visitador con un patron diseñado para recorrer el árbol y evaluar la expresión utilizando una pila. Los nodos que envuelven funciones necesitan que se evalúen primero todos sus nodos hijos para entonces poder aplicar la función que éstos envuelven. Los nodos que envuelven variables y contantes son evaluados para poner sus valores en el tope de la pila.
public final class *JavaCharStream*	Generada por `JavaCC` y se extiende de *java.lang.Object*. Es una implementación de la interfaz *CharStream*, donde el flujo se asume que contiene solo caracteres ASCII (con java-like unicode para el proceso de escape).
public class *JEP*	Se extiende de *java.lang.Object*. La clase *JEP* es la interfaz principal con la cual el usuario debe interactuar. Ésta contiene todos los métodos necesario para el análisis sintáctico y la evaluación de las expresiones matemáticas. Los métodos más importantes son *parseExpression(String)*, para el paso de las expresiones matemáticas al evaluador, y *getValue()* para obtener el resultado de evaluar la expresión.
public class *Parser*	Es generada por `JTTree` y `JavaCC`, y se extiende de *java.lang.Object*. En esta clase es donde se encuentra el analizador sintáctico principal.
public class *ParserDumpVisitor*	Se extiende de *java.lang.Object* e implementa la interfaz *ParserVisitor*. Ésta sirve para recorrer el árbol de sintaxis abstracta y descargarlo para imprimirlo en la consola de forma que se pueda ver el la configuración padre-hijo de éste.
public class *ParserTokenManager*	Es generada por `JTTree` y `JavaCC`, se extiende de *java.lang.Object* e implementa la interfaz *ParserConstants*. Esta clase actúa como un gestor de símbolos terminales para el analizador sintáctico contenido en la clase *Parser*.
public class *SimpleNode*	Generado por `JJTree`, se extiende de *java.lang.Object* e implementa la interfaz *Node*. Ésta es la superclase de aquellas clases que actúan como constantes, variables, funciones y de la que sirven de nodo raíz.
public class *SymbolTable*	Se extiende de *java.util.Hashtable*. En esta se guarda la información de las variables y contantes que son utilizadas en la expresión que se quiere evaluar.

public class *Token*	Es generada por `JavaCC` y se extiende de la clase *java.lang.Object*. Esta clase encapsula los símbolos terminales que el analizador léxico se encarga de obtener para luego pasarlos al analizador sintáctico.

Excepciones en el paquete `org.nfunk.jep`	
Excepción	Descripción
public class *ParseException*	Generada por `JavaCC`, se extiende de la clase *java.lang.Exception* e implementa la interfaz *java.io.Serializable*. Es la excepción que se lanza cuando el analizador sintáctico encuentra algún problema.

Errores en el paquete `org.nfunk.jep`	
Error	Descripción
public class *TokenMgrError*	Es generada por `JavaCC`, se extiende de la clase *java.lang.Error* e implementa la interfaz *java.io.Serializable*. Es la excepción que se lanza cuando se detectan problemas con el reconocimiento de los tokens, además se encarga de que los mensajes de error presenten suficiente información al usuario.

Capítulo 3

Manejo y uso de *JEP*

3.1. Manejo y uso de la clase *JEP*

La clase *JEP* es la más importante del API `JEP`, pues es ésta la que permite el uso de toda la maquinaria diseñada para la interpretación de expresiones matemáticas. A continuación se explican los métodos, opciones propias de *JEP*, se muestra el manejo y uso correcto del interpretador mediante algunos ejemplos, también se explican y se dan ejemplos para la implementación de nuevas funciones de una o de varias variables.

Campos en la clase *JEP*	
Campo	Descripción
protected boolean *allowUndeclared*	Opción que permite no declarar las variables de la expresión, el interpretador las identifica automáticamente y las inicializan con 0. El valor por defecto es *false*.
protected java.util.Vector *errorList*	Estructura de datos en la cual se listarán los errores que se produzcan durante el análisis sintáctico o la evaluación de la expresión matemática. En condiciones normales debe estar vacía.
protected FunctionTable *funTab*	Estructura de datos en la cuál se cargarán las funciones matemáticas que se pueden incluir en la expresión matemática.
protected boolean *implicitMul*	Opción que permite utilizar la multiplicación implícita entre los factores de la expresión a evaluar. El valor por defecto es *false*.
protected SymbolTable *symTab*	Estructura de datos en la cuál se cargarán las variables que se encuentren incluidas en la expresión matemática. Además se cargan la constates matemáticas $\{e, \pi\}$.

private boolean *traverse*	Opción que permite recorrer e imprimir el árbol de sintaxis abstracta en la consola. El valor por defecto es *false*.
private NumberFactory *numberFactory*	Es un objeto que actúa como una fabrica de tipos numéricos por ejemplo: enteros, reales, complejos, etc. Por defecto utiliza *DoubleNumberFactory*.

Constructores de la clase *JEP*	
Constructor	Descripción
public *JEP*()	Crea una nueva instancia de *JEP* con la configuración por defecto. *traverse* = *false*; *allowUndeclared* = *false*; *implicitMul* = *false*; *numberFactory* = *DoubleNumberFactory*;
public *JEP*(boolean traverse_in, boolean allowUndeclared_in, boolean implicitMul_in, NumberFactory numberFactory_in)	Crea una nueva instancia de *JEP* con la configuración especificada por los argumentos de este constructor. Si el parámetro *numberFactory_in* es *null*, entonces se usa el valor por defecto *DoubleNumberFactory*. *traverse* = *traverse_in*; *allowUndeclared* = *allowUndeclared_in*; *implicitMul* = *implicitMul_in*; *numberFactory* = *numberFactory_in*;

Métodos de la clase *JEP*	
Métodos	Descripción
public void *initSymTab*()	Crea un nuevo objeto *SymbolTable* y lo asigna a *symTab*.
public void *initFunTab*()	Crea un nuevo objeto *FunctionTable* y lo asigna a *funTab*.
public void *addStandardFunctions*()	Adiciona a la estructura *funTab*, las funciones matemáticas estándar que se encuentran implementadas en el paquete `org.nfunk.jep.function` , tales como sin(), cos(). Si se utiliza alguna de estas funciones en la expresión matemática a evaluar y no se ha cargado a la tabla de funciones entonces se produce una excepción ‘‘`Unrecognized function...`’’. Lo más adecuado en caso de utilizar funciones matemáticas, es hacer el llamado a este método inmediatamente después de instanciar la clase *JEP*.
public void *addStandardConstants*()	Adiciona a la estructura *symTab* las constantes π y e, para poderlas usar en la expresión a evaluar. Como en el método anterior, este debe ser llamado inmediatamente después de instanciar la clase *JEP*.

public void *addComplex*()	Debe ser llamada en caso de que la expresión a evaluar involucre números complejos. Se especifica a i como unidad imaginaria y se adiciona a la estructura que contiene la constantes *symTab*; además se agregan las funciones re() e im() a la estructura de funciones *funTab*.
public void *addFunction*(String functionName, PostfixMathCommandI function)	Debe ser llamada en caso de que sea necesario utilizar una función matemática que no éste implementada en `org.nfunk.jep.function`. *functionName* es el nombre que se le da a la nueva función en la expresión a evaluar y *function* es la clase donde se define la nueva función. Este método debe ser llamado antes de que se utilice la nueva función. En la subsección 3.1.2 se explica y ejemplariza la forma de diseñar e implementar nuevas funciones.
public Double *addVariable*(String name, double value)	Adiciona una nueva variable a la estructura que contiene las variables o actualiza el valor de una variable existente. Debe llamarse antes de realizarse la evaluación, para que se opere sobre los valores correctos. *name* es el nombre de la variable que se utiliza en la expresión a evaluar, y *value* es el valor que queremos asignarle a la variable. Retorna un objeto *Double* que envuelve a *value*.
public Complex *addComplexVariable*(String name, double re, double im)	Adiciona una nueva variable compleja a la estructura que contiene las variables o actualiza el valor de una variable existente. Debe llamarse antes de realizarse la evaluación, para que se opere sobre los valores correctos. *name* es el nombre de la variable que se utiliza en la expresión a evaluar, *re* e *im* son la parte real e imaginaria del complejo respectivamente. Retorna un objeto *Complex* con parte real e imaginaria *re* e *im* respectivamente.
public void *addVariableAsObject*(String name, Object object)	Adiciona un objeto como una nueva variable a la estructura que contiene las variables o actualiza el valor de una variable existente. Debe llamarse antes de realizarse la evaluación, para que se opere sobre los valores correctos. *name* es el nombre de la variable que se utiliza en la expresión a evaluar, y *object* es un objeto que envuelve el valor de la variable *name*.
public Object *removeVariable*(String name)	Debe ser llamada cuando se quiera eliminar una variable existente. Retorna el objeto que envuelve el valor de la variable, y en caso de que no exista, retorna *null*.
public Object *removeFunction*(String name)	Debe ser llamada cuando se quiera eliminar una función existente. Retorna el objeto que sirve para evaluar la función y en caso de que no exista retorna *null*.

public void *setTraverse*(boolean value)	Si *value* es igual a *true* se activa la opción de imprimir en la consola el árbol de sintaxis abstracta en el momento en el que se pasa la expresión al analizador sintáctico. Este método debe llamarse antes de pasar la expresión al analizador sintáctico. La opción por defecto es *false*.
public void *setImplicitMul*(boolean value)	Si *value* es igual a *true* se activa la opción de la multiplicación implícita, es decir una expresión como 2 3 y $2pi$ son interpretadas como $2*3$ y $2*pi$ respectivamente. La opción por defecto es *false*.
public void *setAllowUndeclared*(boolean value)	Si *value* es igual a *true* se activa la opción de reconocer las variables de la expresión de forma automática, se adicionan las variables detectadas a la estructura de variables con el valor inicial igual a 0. La opción por defecto es *false*.
public void *parseExpression*(String expression_in)	Es el método al cual se le debe pasar la expresión a evaluar. Si la expresión esta incorrectamente escrita, entonces se adiciona la excepción a la lista de errores *errorList*.
public double *getValue*()	Evalúa y retorna el valor de la expresión. Si el valor es complejo, se retorna la componente real. Si ocurre un error durante la evaluación, entonces el valor retornado es 0.
public Complex *getComplexValue*()	Evalúa y retorna el valor de la expresión como un número complejo. Si ocurre un error durante la evaluación, entonces se retorna *null*.
public Object *getValueAsObject*()	Evalúa y retorna el valor de la expresión como un objeto. Si ocurre un error durante la evaluación, entonces se retorna *null*.
public boolean *hasError*()	Retorna *true*, si a ocurrido un error durante la acción más reciente (análisis sintáctico o evaluación).
public String *getErrorInfo*()	Reporta la información sobre los errores ocurridos durante la más reciente acción (análisis sintáctico o evaluación). Se retorna la lista de los errores ocurridos, si no ocurren errores, entonces se retorna *null*.
public Node *getTopNode*()	Retorna la raíz del árbol de sintaxis abstracta obtenido al analizar la expresión. Con este nodo se puede evaluar la expresión manualmente ya que desde éste se puede llegar a cualquier nodo del árbol.
public SymbolTable *getSymbolTable*()	Retorna la tabla en la que se encuentran almacenadas las constates y variables.
public NumberFactory *getNumberFactory*()	Retorna el objeto *NumberFactory* que se especificó en el constructor de la clase.

3.1.1. Ejemplo general del uso de *JEP*

La clase que se muestra a continuación es un ejemplo clásico del uso de *JEP*; ésta es la forma correcta de utilizar *JEP* en la mayoría de los casos. Es importante notar que la multiplicación es explícita, las variables debe ser adicionadas por el usuario y la revisión de errores se hace en cada posible caso en que puedan ocurrir.

```
import org.nfunk.jep.*;

public class FuncionEjem {

    public FuncionEjem() {
    }

    public static void main(String[] args) {
        // crea una nueva instancia de JEP con la configuración por defecto
        JEP funcion = new JEP();
        String expresion = "e^x*sin(y)+e^y*cos(x)";
        double value;
        System.out.println(
                "Ejemplo del uso de JEP con la función: " + expresion);
        funcion.addStandardFunctions(); // adiciona las funciones matemáticas
        funcion.addStandardConstants(); // adiciona las constantes matemáticas
        // adiciona las variables y sus valores iniciales
        funcion.addVariable("x", 2.0);
        funcion.addVariable("y", -2.0);
        funcion.parseExpression(expresion); // paso de la expresión a evaluar
        // revisar si han ocurrido errores durante el análisis de la expresión
        if (funcion.hasError()) {
            System.out.println("Error durante el análisis sintáctico");
            System.out.println(funcion.getErrorInfo());
            return;
        }
        // obtener el resultado de evaluar la expresión
        value = funcion.getValue();
        // revisar si han ocurrido errores durante la evaluación de la expresión
        if (funcion.hasError()) {
            System.out.println("Error durante la evaluación");
            System.out.println(funcion.getErrorInfo());
            return;
        }
        // imprime la expresión evaluada y el resultado obtenido al evaluarla
        System.out.println(expresion + " = " + value);
        // cambiar el valor de los paramentros para evaluar la expresión
        funcion.addVariable("x", -2.0);
        funcion.addVariable("y", 2.0);
        value = funcion.getValue();
```

```
        System.out.println(expresion + " = " + value);
    }
}
```

El resultado obtenido al ejecutar este programa es:

```
Ejemplo del uso de JEP con la función: e^x*sin(y)+e^y*cos(x)%
e^x*sin(y)+e^y*cos(x) = -6.775169047420377    %
e^x*sin(y)+e^y*cos(x) = -2.951872295833582
```

3.1.2. Definición e implementación de nuevas funciones

JEP permite la adición de nuevas funciones que sean definidas por el usuario. Para poder realizar esto, es necesario determinar el número de variables que tiene la función, definir un nueva clase extendida de *PostfixMathCommand*, de la que en su constructor por defecto se especifique el número de parámetros que necesita para la evaluación y además sobre-escribir el método *run()* de la superclase, el cuál tiene por objetivo proveer los parámetros para evaluar la nueva función y poner el resultado de la evaluación en la pila, que es una estructura de datos auxiliar que ayuda a realizar la evaluación de una expresión matemática en notación posfija.

Para especificar el número de parámetros que son necesarios para realizar la evaluación en el constructor por defecto, se debe asignar al campo *numberOfParameters* el número de variables de la función y en caso de que este número no sea constante, por ejemplo, en el caso de una sumatoria, se debe asignar a *numberOfParameters* el valor -1.

Para poder sobre-escribir el método *run()*, es necesario saber como se encuentran ubicados los parámetros con los cuales se realiza la evaluación de la nueva función. Por ejemplo, si se tiene una función $f(x_1, x_2, \ldots, x_{n-1}, x_n)$, entonces los parámetros $x_1, x_2, \ldots, x_{n-1}, x_n$, se encuentran ubicados en orden inverso desde el tope de la pila, así como se muestra en la figura 3.1.

Para obtener los parámetros se revisa que la pila no se encuentre vacía con el método *checkStack()* y luego se obtiene el tope de la pila con el método *pop()* propio de la clase *Stack*; esto se realiza por cada uno de los parámetros. Luego se debe poner el resultado de la evaluación de la función en el tope de la pila con el método *push()* propio de la clase *Stack*.

En la pila cada nodo es un objeto y como en general para la evaluación de la función se diseña un nuevo método, entonces para cada objeto obtenido de la pila es necesario saber de que tipo es (p. ej. *Double*, *Complex*), para así poder realizar la evaluación correctamente. En `Java` existe el operador *instanceof* que permite saber de que tipo es un objeto, y tiene el formato:

`objeto` *instanceof* `tipo`

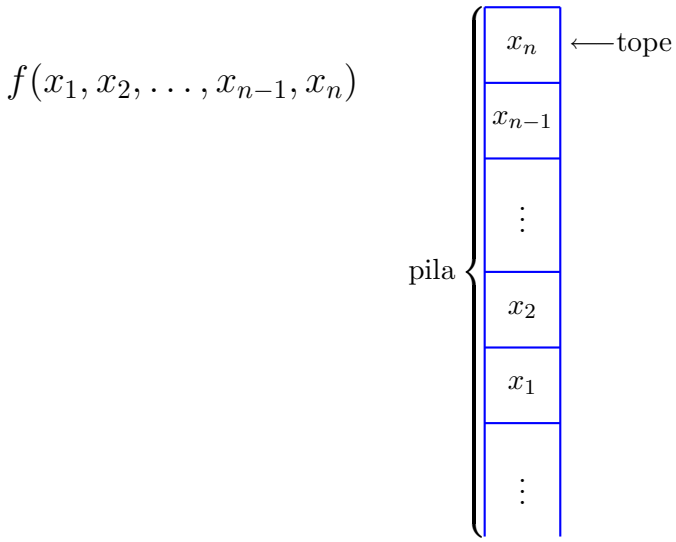


Figura 3.1: Ubicación de los parámetros de una función en un pila.

este operador retorna *true* o *false* en caso de que sea o no una instancia de la clase especificada (*tipo*). Mas adelante se mostrará como se debe utilizar este operador en la implementación.

A continuación se muestran dos ejemplos de como se deben implementar nuevas funciones en *JEP*.

Ejemplo Implementación de la función exponencial (*Exponential()*) de variable real y compleja, la cual no se encuentra implementada entre las definidas en el paquete `org.nfunk.jep.function`.

```
import java.util.*;
import org.nfunk.jep.*;
import org.nfunk.jep.type.*;
import org.nfunk.jep.function.*;

public class Exponential extends PostfixMathCommand {
    /**
    * Crea una nueva instancia de Exponential y establece el número
    * de parámetros
    */
    public Exponential() {
        numberOfParameters = 1;
    }
     /**
     * Permite obtener el parámetro de la pila y poner el resultado de la
     * evaluación de la función exponencial.
     */
```

```
    public void run(Stack inStack) throws ParseException {
        // chequear si la pila está vacía
        checkStack(inStack);
        Object param = inStack.pop(); // obtener el tope de la pila
        // poner el resultado de la evaluación en la tope de la pila
        inStack.push(exp(param));
    }
    /**
     * Realiza la evaluación de la función exponencial dependiendo de si el
     * argumento es un número real o un número complejo.
     */
    public Object exp(Object param) throws ParseException {
        if (param instanceof Number) {
            return new Double(Math.exp(((Number)param).doubleValue()));
        }else if (param instanceof Complex) {
            //e^z = e^x(cos y + i*sen y); z=(x,y)
            Complex z = (Complex)param;
            double x = z.re();
            double y = z.im();
            return new Complex(Math.exp(x)*Math.cos(y), Math.exp(x)*Math.sin(y));
        }
        throw new ParseException("Invalid parameter type");
    }
}
```

Ejemplo Se ilustra la implementación una función que tiene mas de una variable, esta función es el ejemplo ilustrado en figura 1.3, $u(x, y) = e^x \operatorname{sen} y + e^y \cos x$.

```
import java.util.*; import org.nfunk.jep.*; import
org.nfunk.jep.type.*; import org.nfunk.jep.function.*;

public class FuncionXY extends PostfixMathCommand {
    /** Crea una nueva instancia de FuncionXY */
    public FuncionXY() {
        numberOfParameters = 2;
    }
    /**
     * Permite obtener los 2 argumentos de la pila y poner el resultado
     * de la evaluación de la función.
     */
    public void run(Stack inStack) throws ParseException {
        // arreglo de objetos con el número de parámetros definidos
        Object param[] = new Object[this.numberOfParameters];
        // obtener los objetos en orden inverso al de la pila
        for(int i=this.numberOfParameters-1; i >= 0 ; i--){
            checkStack(inStack); // chequear que la pila no este vacía
```

```
            param[i] = inStack.pop(); // obtener el tope de la pila
        }
        // poner el resultado en el tope de la pila
        inStack.push(funcionXY(param[0],param[1]));
    }
    /**
     * Realiza la evaluación de la función verificando que los parámetros sean
     * números reales.
     */
    public Object funcionXY(Object param1, Object param2) throws ParseException {
        if (param1 instanceof Number && param2 instanceof Number) {
            double x = ((Number)param1).doubleValue();
            double y = ((Number)param2).doubleValue();
            return new Double(Math.exp(x)*Math.sin(y) + Math.exp(y)*Math.cos(x));
        }else
            throw new ParseException("Invalid parameter type");
    }
}
```

3.1.3. Ejemplos del uso de los métodos de *JEP*

Uso de *void parseExpression(String expression_in)*

Este método es el utilizado para pasarle al analizador sintáctico la expresión matemática a evaluar.

Por ejemplo en la clase ilustrada en la sección 3.1.1, se pasa la expresión matemática $e^x \sin(y) + e^y \cos(x)$ de la siguiente forma:

```
String expresion = "e^x*sin(y)+e^y*cos(x)";
funcion.parseExpression(expresion); // paso de la expresión a evaluar
```

Uso de *Double addVariable(String name, double value)*

Al hacer un llamado a este método se adiciona la variable *"name"* y se inicializa con el valor *value*, o en caso de que ya se encuentre adicionada, se actualiza su valor con *value*. Este método retorna el objeto *Double(value)*.

En la clase ilustrada en la sección 3.1.1, se hace la adición de las variables `x` y `y`, de la siguiente forma:

```
// adiciona las variables y sus valores iniciales
funcion.addVariable("x", 2.0);
funcion.addVariable("y", -2.0);
```

en caso de que se quiera cambiar el valor de alguna de las variables para realizar una

nueva evaluación se debe escribir la misma línea, pero después de pasar la expresión al analizador sintáctico de la siguiente forma:

```
// adiciona las variables y sus valores iniciales
funcion.addVariable("x", 2.0);
funcion.addVariable("y", -2.0);
funcion.parseExpression(expresion); // paso de la expresión a evaluar
funcion.addVariable("x", 4.0); // actualiza la variable x con el valor 4.0
```

y el nuevo resultado obtenido es:

```
Ejemplo del uso de JEP con la función: e^x*sin(y)+e^y*cos(x) %
e^x*sin(y)+e^y*cos(x) = -49.73441837914594
```

Uso de *Complex addComplexVariable(String name, double re, double im)*

Al hacer un llamado a este método se adiciona la variable compleja *"name"* y se inicializa la componente real con *re* y la componente imaginaria con *im* o en caso de que ya se encuentre adicionada, se actualiza su valor. Este método retorna el objeto *Complex(re,im)*. Para ejemplarizar el uso de este método se utilizará la igualdad:

$$\text{Log}(-ei) = 1 - \frac{\pi}{2}i$$

Para obtener un valor aproximado de $\text{Log}(-ei)$ se utilizan las siguientes líneas:

```
funcion.addComplexVariable("x", 0, -Math.E);
funcion.parseExpression("ln(x)");
```

Uso de *void addVariableAsObject(String name, Object object)*

Con este método se pueden adicionar objetos como variables. Por ejemplo para operar con cadenas, complejos o vectores se puede o es aconsejable utilizar este método. Para realizar las siguientes operaciones se utiliza este método que permite adicionar los operandos:

- "*a*"+"*b*"+"*c*"+"*d*"="*abcd*"; (concatenación)
- $(1 + 3i) + (1 - 2i) = (2 + i)$; (suma de complejos)
- $[3, 4, 5] * 2 = [6, 8, 10]$; (multiplicación de un vector por un escalar)

Uso de *void addStandardFunctions()*

Al hacer un llamado a este método se adicionan al analizador sintáctico las funciones más comúnmente utilizadas en matemáticas. En la tabla 3.1 se muestran las funciones

adicionadas y los objetos sobre los cuales pueden operar éstas. Es importante tener en cuenta que la invocación de este método debe hacerse antes de pasar la expresión al analizador sintáctico como en la clase implementada en la sección 3.1.1, por otro lado si no se hace la invocación de este método, entonces la utilización de alguna de las funciones definidas producirá una excepción.

Funciones adicionadas con *addStandardFunctions()*					
Función	**Nombre**	**Double**	**Complex**	**String**	**Vector**
Seno	sin()	✓	✓		
Coseno	cos()	✓	✓		
Tangente	tan()	✓	✓		
Arco Seno	asin()	✓	✓		
Arco Coseno	acos()	✓	✓		
Arco Tangente	atan()	✓	✓		
Seno Hiperbólico	sinh()	✓	✓		
Coseno Hiperbólico	cosh()	✓	✓		
Tangente Hiperbólico	tanh()	✓	✓		
Arco Seno Hiperbólico	asinh()	✓	✓		
Arco Coseno Hiperbólico	acosh()	✓	✓		
Arco Tangente Hiperbólico	atanh()	✓	✓		
Logaritmo Natural	ln()	✓	✓		
Logaritmo Base 10	log()	✓	✓		
Angle	angle()	✓			
Valor absoluto / Norma	abs()	✓	✓		
Número Aleatorio (entre 0 y 1)	rand()				
Modulo	mod()	✓			
Raíz Cuadrada	sqrt()	✓	✓		
Suma Finita	sum()	✓			

Tabla 3.1: Funciones adicionadas al analizador sintáctico.

Uso de *void addComplex()*

Al hacer un llamado a este método se habilita al analizador sintáctico para que opere con valores complejos. Se adiciona la constate imaginaria "i", y las funciones que actúan sobre objetos complejos se muestran en la tabla 3.2.

Uso de *void addStandardConstants()*

Al hacer un llamado a este método se adicionan al analizador sintáctico las aproximaciones definidas en `Java` de la constantes reales matemáticas más importantes y comúnmente

Funciones adicionadas con *addComplex()*					
Función	**Nombre**	**Double**	**Complex**	**String**	**Vector**
Componente Real	re()		✓		
Componente Imaginaria	im()		✓		

Tabla 3.2: Funciones complejas adicionadas al analizador sintáctico.

usadas π y e. Los valores de doble precisión adicionados son:

$$\pi \approx \text{Math.PI} = 3{,}141592653589793$$
$$e \approx \text{Math.E} = 2{,}718281828459045$$

Uso de *void addFunction(String functionName, PostfixMathCommandI function)*

Al hacer un llamado a este método se adiciona una nueva función adicional a las adicionadas por *addStandardFunctions()*. Por ejemplo si queremos adicionar la función exponencial a la clase definida en la sección 3.1.1, de modo que pueda ser utilizada, para evaluar la misma expresión de esta clase se debe adicionar la siguiente linea:

```
funcion.addFunction(‘‘exp’’, new Exponential());
```

debe llamarse antes de pasar la expresión a evaluar. El argumento *"exp"* es el nombre con que se identifica la función en la expresión a evaluar, y *Exponential()* es el la implementación de la función definida en el primer ejemplo de la sección 3.1.2. y ahora podemos sustituir la expresión `‘‘e^x*sin(y)+e^y*cos(x)’’` por la nueva expresión `‘‘exp(x)*sin(y)+exp(y)*cos(x)’’`, con la cual ahora se obtienen los mismos resultados que los mostrados en la sección 3.1.1

Uso de *double getValue()*

Este método es utilizado para evaluar la expresión pasada al analizador sintáctico y retornar el valor obtenido. Debe ubicarse después del método `parseExpression`. En caso de que el valor sea un complejo, se retorna la componente real, y en caso de que haya ocurrido un error durante la evaluación se retorna 0.

Por ejemplo en la clase ilustrada en la sección 3.1.1, se retorna el valor de la siguiente forma:

```
String expresion = "e^x*sin(y)+e^y*cos(x)";
funcion.parseExpression(expresion); // paso de la expresión a evaluar
value = funcion.getValue();
```

Uso de *Complex getComplexValue()*

Este método es utilizado para evaluar la expresión pasada al analizador sintáctico y retornar el valor en forma de un objeto *Complex*, Debe ubicarse después del método *parseExpression*. En caso de que el valor sea un real, se retorna un objeto *Complex* cuya componente imaginaria es igual a 0, y en caso de que allá ocurrido un error durante la evaluación se retorna *null*.

Un ejemplo de la ubicación de los métodos y la forma en que se debe retornar el valor se encuentra a continuación, se utiliza la expresión $\text{Log}(-ei) = 1 - \frac{\pi}{2}i$

```
funcion.addComplexVariable("x",  0, -Math.E);
funcion.parseExpression("ln(x)");
Complex complejo = funcion.getComplexValue();
System.out.println(complejo);
```

el resultado obtenido es:

```
(1.0, -1.5707963267948966)
```

Uso de *Object getValueAsObject()*

Este método es utilizado para evaluar y retornar el resultado como un objeto, es usado internamente porque para *JEP* los operandos son siempre objetos. Puede ser utilizado cuando el resultado no es un tipo primitivo o no es una instancia de *Complex*. Por ejemplo cuando el resultado es un vector o una cadena.

Uso de *Object removeVariable(String name)*

Este método es utilizado para eliminar y retornar una variable o constante matemática del analizador sintáctico, si no existe dicha variable o constante entonces se retorna *null*. Por ejemplo, si se tiene implementada la función definida en el primer ejemplo de la sección 3.1.2 (*Exponential()*), es de suponer que no se quiera ya tener la constate e por lo que se utilizará el método de la siguiente forma:

```
Object removeVariable("e");
```

Uso de *Object removeFunction(String name)*

Este método es utilizado para eliminar y retornar una función de las adicionadas al analizador sintáctico, si no existe dicha función entonces se retorna *null*. Por ejemplo, si se desea eliminar la función *sin()* que se encuentra escrita es inglés, y se desea adicionar la misma función pero con su nombre en castellano *sen()*, se debe escribir las siguientes líneas.

```
Object removeFunction("sin");
addFunction("sen", new Sine());
```

Uso de *void setTraverse(boolean value)*

Con este método se recorre el árbol de sintaxis abstracta y se imprime éste en la consola si *value* es igual a *true*. Este método debe ubicarse antes de pasar la expresión al analizador sintáctico.

Por ejemplo el árbol generado por la expresión ‘‘e^x*sin(y)+e^y*cos(x)’’ que se muestra en la figura 1.3 y el cual se imprime en la consola de la siguiente forma:

```
Function ""+""
    Function ""*""
        Function ""^""
            Variable: "e"
            Variable: "x"
        Function "sin"
            Variable: "y"
    Function ""*""
        Function ""^""
            Variable: "e"
            Variable: "y"
        Function "cos"
            Variable: "x"
```

Uso de *void setImplicitMul(boolean value)*

Con este método se permite activar la multiplicación implícita si *value* es igual a *true*. Este método debe ubicarse antes de pasar la expresión al analizador sintáctico.

En la multiplicación implícita se entiende que si entre dos operandos no hay operador, entonces el operador se sobrentiende que es el de multiplicación, por ejemplo expresiones como ‘‘2 x’’ y ‘‘2x’’ son interpretadas como ‘‘2*x’’. En el caso anterior es importante tener en cuenta que en general la multiplicación no es conmutativa por ejemplo ‘‘2x’’ no es lo mismo que ‘‘x2’’, pues la primera es una multiplicación y la segunda es una variable que contiene dígitos; por lo que es aconsejable en general utilizar la multiplicación explícita y la notación usual del álgebra, donde el coeficiente se ubica primero que la variable.

Uso de *void setAllowUndeclared(boolean value)*

Con este método se permite el analizador sintáctico que detecte automáticamente las variables que se encuentran en la expresión y las inicializa con 0. Por ejemplo, en la expresión $e^x \sin(y) + e^y \cos(x)$, el analizador sintáctico detectará las variables x y y, y las iniciará con el valor 0 de la siguiente manera:

```
"x" = 0.0;
"y" = 0.0;
```

Uso de *boolean hasError()*

Este método retorna *true* si ha ocurrido algún error durante el análisis sintáctico o la evaluación. Por ejemplo, si en el ejemplo de la sección 3.1.1 en vez de pasarle la expresión original ‘‘e^x*sin(y)+e^y*cos(x)’’, le pasáramos la expresión ‘‘e^x sin(y)+e^y cos(x)’’, en la consola se imprimiría

```
Ejemplo del uso de JEP con la función: e^x sin(y)+e^y cos(x)
Error durante el análisis sintáctico
```

porque la expresión ‘‘e^x sin(y)+e^y cos(x)’’ es sintácticamente correcta si la opción de la multiplicación implícita se encuentra activada, en caso contrario producirá un error, como lo es este caso.

Uso de *String getErrorInfo()*

Retorna el informe de los errores ocurridos durante la mas reciente acción (análisis sintáctico o evaluación). Por ejemplo, si en el ejemplo de la sección 3.1.1 en vez de pasarle la expresión original ‘‘e^x*sin(y)+e^y*cos(x)’’, le pasáramos la expresión ‘‘e^x sin(xy)+e^y cos(xy)’’, en la consola se imprimiría

```
Ejemplo del uso de JEP con la función: e^x sin(xy)+e^y cos(xy)
Error durante el análisis sintáctico
Unrecognized symbol "xy"
Syntax Error (implicit multiplication not enabled)
```

En donde la dos ultimas líneas corresponden al informe obtenido al hacer uso de este método. Los errores se produjeron porque la multiplicación implícita no se encuentra activada y además la variable `xy` no es reconocida por el analizador sintáctico.

Uso de *SymbolTable getSymbolTable()*

Este método retorna la tabla de símbolos. Por ejemplo, si se imprimiera la *SymbolTable* obtenida al utilizar este método después de adicionar las variables en el ejemplo de la sección 3.1.1. Entonces se obtendría además de las líneas comunes la siguiente línea en la consola

```
{x=2.0, e=2.718281828459045, pi=3.141592653589793, y=-2.0}
```

3.1.4. Operadores definidos en *JEP*

En *JEP* se encuentran definidos la mayoría de los operadores utilizados en computación. Existen operadores que se encuentran sobrecargados, de tal manera que con el mismo símbolo (operador) se pueden operar distintos objetos. Por ejemplo, el operador "+" sirve para sumar objetos *Double* o *Complex*, y concatenar cadenas *String*. En la tabla 3.3 se muestran los operadores permitidos en `JEP` y los objetos sobre los cuales operan.

Operadores y los TDA sobre los cuales pueden operar					
Operador	**Símbolo**	**Double**	**Complex**	**String**	**Vector**
Potencia	$\hat{}$	✓	✓		
No Lógico	!	✓			
Mas Unario, Menos Unario	$+x, -x$	✓	✓		
Modulo	%	✓			
División	/	✓	✓		✓
Multiplicación	$*$	✓	✓		✓
Adición, Sustracción	$+, -$	✓	✓	✓(solo +)	
Menor o Igual, Mayor o Igual	$<=, >=$	✓			
Menor que o Mayor que	$<, >$	✓			
No es Igual, Igual	$!=, ==$	✓	✓	✓	
Y Lógico	&&	✓			
O Lógico	$\|\|$	✓			

Tabla 3.3: Operadores definidos en `JEP`.

Capítulo 4

Un orden total para intervalos del conjunto de números de máquina

En este capitulo se definirán unos conjuntos muy especiales llamados intervalos, los cuales se encuentran conformados por números de máquina, estos conjunto conformarán un conjunto en él cual se define una relación de orden que resultará siendo un orden total.

4.1 Definición. (**Orden parcial**) Sea S un conjunto. Un *orden parcial* de S es una relación, denotada $\leq$, que ésta definida para algunos pares ordenados de elementos de S y satisface tres condiciones:

(i) $x \leq x$ para todo $x \in S$ **Ley reflexiva**

(ii) si $x \leq y$ y $y \leq x$, entonces $x = y$ **Ley antisimétrica**

(iii) si $x \leq y$ y $y \leq z$, entonces $x \leq z$ **Ley transitiva**

Sí S tiene un orden parcial definido sobre éste, entonces se dice que S es un *conjunto parcialmente ordenado* y es denotado por $(S, \leq)$.

4.2 Definición. (**Orden total**) Sea S un conjunto que posee un orden parcial $\leq$. Se dice que $\leq$ es un *orden total* sí dados cualesquiera elementos x y y de S, se tiene que $x \leq y$ o $y \leq x$.

4.3 Definición. (**Conjunto de los números de máquina**[1]) Se denomina al conjunto de números reales en forma de punto flotante que pueden ser representados por una máquina. A este conjunto lo denotaremos con $\mathcal{M}$.

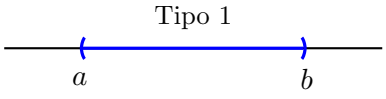
Nota. El conjunto $\mathcal{M}$ es un subconjunto propio finito del conjunto de los números reales formado únicamente por números racionales, es decir $\mathcal{M} \subset \mathbb{Q}$.

[1] En el apéndice C se hace una descripción más detallada de la teoría de los números de máquina.

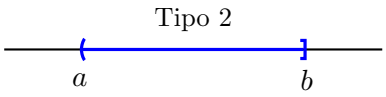
4.4 Definición. (**Conjunto de intervalos de $\mathcal{M}$**) Sean $a, b \in \mathcal{M}$, en la figura 4.1 se definen 4 tipos "distintos" de conjuntos en $\mathcal{M}$, se dice que un conjunto $I \subseteq \mathcal{M}$ es un intervalo, si I cumple todas las condiciones exhibidas en alguno de los conjuntos caracterizados en la figura 4.1.

Al valor a se le denomina el extremo inferior del intervalo, y al valor b se le denomina el extremo superior del intervalo.

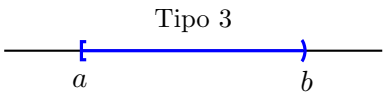
Al conjunto formado por estos intervalos se denomina conjunto de intervalos de $\mathcal{M}$ y se denotará con el símbolo $\mathcal{I}$.



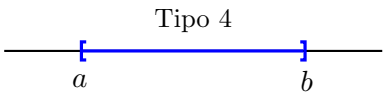
El *intervalo abierto* (a, b) en $\mathcal{M}$, es el conjunto de x tales que $a, b, x \in \mathcal{M}$ y $a < x < b$.



El *intervalo semiabierto* $(a, b]$ en $\mathcal{M}$, es el conjunto de x tales que $a, b, x \in \mathcal{M}$ y $a < x \leq b$.



El *intervalo semicerrado* $[a, b)$ en $\mathcal{M}$, es el conjunto de x tales que $a, b, x \in \mathcal{M}$ y $a \leq x < b$.



El *intervalo cerrado* $[a, b]$ en $\mathcal{M}$, es el conjunto de x tales que $a, b, x \in \mathcal{M}$ y $a \leq x \leq b$.

Figura 4.1: Definición de los cuatro tipos de intervalos en $\mathcal{M}$.

$X = Y$				
$\frac{Y \rightarrow}{X \downarrow}$	(c, d)	$(c, d]$	$[c, d)$	$[c, d]$
(a, b)	$a = c, \wedge, b = d$			
$(a, b]$		$a = c, \wedge, b = d$		
$[a, b)$			$a = c, \wedge, b = d$	
$[a, b]$				$a = c, \wedge, b = d$

Tabla 4.1: Condiciones para tener la igualdad entre intervalos.

4.5 Definición. Sean $x, y \in \mathcal{I}$, se dice que x "es igual a" y (lo denotaremos por $x = y$), si y solo si, x y y son del mismo tipo de intervalo y si además tanto los extremos inferiores como los superiores coinciden.

En la tabla 4.1, se caracterizan las condiciones para que dos intervalos dados sean iguales; para esto se debe tener en cuenta los tipos de los intervalos y las condiciones dadas en cada celda. En caso de que la celda correspondiente este vacía ésto significará que los intervalos comparados son distintos.

4.6 Definición. Sean $x, y \in \mathcal{I}$, si a y c son los extremos inferiores de x y y respectivamente, y si b y d son los extremos superiores de x y y respectivamente, entonces, se dice que x "es estrictamente menor que" y (lo denotaremos por $x < y$), si y solo si, se cumple alguno de los siguientes casos:

Caso 1: Se comparan los extremos inferiores, si $a < c$ entonces $x < y$.

Caso 2: Si los extremos inferiores son iguales ($a = c$), y si $a \in x$ y $c \notin y$ entonces $x < y$.

Caso 3: Si los extremos inferiores son iguales ($a = c$), y si estos pertenecen o no simultáneamente al intervalo respectivo, y si $b < d$ entonces $x < y$.

Caso 4: Si los extremos inferiores son iguales ($a = c$), y si estos pertenecen o no simultáneamente al intervalo respectivo, y si se tiene que los extremos superiores son iguales ($b = d$), y si $b \notin x$ y $d \in y$ entonces $x < y$.

En la tabla 4.2, se caracterizan las condiciones para que dados dos intervalos x y y se pueda deducir que $x < y$; para esto se debe tener en cuenta las condiciones dadas en cada celda de acuerdo al tipo en el cual clasifiquen x y y respectivamente.

$X < Y$				
$\frac{Y \rightarrow}{X \downarrow}$	(c, d)	$(c, d]$	$[c, d)$	$[c, d]$
(a, b)	$(a < c)$ $\vee$ $(a = c, \wedge, b < d)$	$(a < c)$ $\vee$ $(a = c, \wedge, b \leq d)$	$a < c$	$a < c$
$(a, b]$	$(a < c)$ $\vee$ $(a = c, \wedge, b < d)$	$(a < c)$ $\vee$ $(a = c, \wedge, b < d)$	$a < c$	$a < c$
$[a, b)$	$a \leq c$	$a \leq c$	$(a < c)$ $\vee$ $(a = c, \wedge, b < d)$	$(a < c)$ $\vee$ $(a = c, \wedge, b \leq d)$
$[a, b]$	$a \leq c$	$a \leq c$	$(a < c)$ $\vee$ $(a = c, \wedge, b < d)$	$(a < c)$ $\vee$ $(a = c, \wedge, b < d)$

Tabla 4.2: Condiciones para la desigualdad "estrictamente menor que" entre intervalos.

4.7 Definición. Sean $x, y \in \mathcal{I}$, se dice que x "es estrictamente mayor que" y (lo denotaremos por $x > y$), si y solo si, x no es igual a y y x no es estrictamente menor que y, es decir si x y y no cumplen simultáneamente las dos condiciones para la igualdad y tampoco cumplen simultáneamente las condiciones de alguno de los cuatro casos descritos en la definición 4.6.

Nota. De las definiciones 4.5, 4.6 y 4.7 se puede observar que $\mathcal{I}$ posee la propiedad de tricotomía, es decir dados $x, y \in \mathcal{I}$ se verifica una y sólo una de las tres relaciones $a = b$, $a < b$, $a > b$. Esto se tiene, porque si dos intervalos son iguales, entonces no se puede dar que el primero sea estrictamente menor que el segundo y tampoco que el primero sea estrictamente mayor que el segundo. Ahora si el primero es estrictamente menor que el segundo entonces no se puede dar que sean iguales y tampoco que el primero sea estrictamente mayor que el segundo. Y por ultimo, si el primero es estrictamente mayor que el segundo, entonces esto ocurre porque no son iguales y el primero no estrictamente menor que el segundo.

4.8 Definición. Sean $x, y \in \mathcal{I}$, se define la relación en $\mathcal{I}$ denotada por $\preceq$ de la siguiente forma:

$$x \preceq y \Longleftrightarrow x < y, \vee, x = y$$

4.9 Teorema. *Si $\preceq$ es la relación exhibida en la definición 4.8, entonces, $\preceq$ define un orden parcial sobre $\mathcal{I}$. Es decir la relación $\preceq$ es reflexiva, antisimétrica y transitiva en $\mathcal{I}$.*

Demostración. Sean $x, y, z \in \mathcal{I}$. Si a, a', a'' denotan los extremos inferiores de los intervalos x, y, z respectivamente y b, b', b'' denotan los extremos superiores de los intervalos x, y, z respectivamente. Para demostrar que $\preceq$ define un orden parcial, se demostrará cada una de las propiedades (reflexividad, antisimetría, transitividad) por aparte.

i. Reflexividad: Dado que $x \in \mathcal{I}$, entonces x es de un solo tipo de intervalo, y como también los extremos son únicos por definición, entonces por la definición de igualdad de intervalos, se tiene que $x = x$, por lo tanto $x \preceq x$, es decir la relación $\preceq$ es reflexiva.

ii. Antisimetría: A continuación se analizarán los casos para que se pueda dar simultáneamente que $x \leq y$, y, $y \leq x$. Si $x < y$ entonces por definición no se puede dar que $y < x$; si $x < y$ entonces por definición no se puede dar que $y = x$; si $x = y$ entonces por definición no se puede dar que $y < x$; si $x = y$ se tiene que $y = x$ y por lo tanto $x = y$. De los razonamientos anteriores se puede observar que para que se pueda dar simultáneamente que $x \preceq y$ y $y \preceq x$, se debe tener que $x = y$; luego se tiene que $\preceq$ es una relación antisimétrica.

iii. Transitividad: Si $x \preceq y$ y $y \preceq z$, entonces por definición $x = y$ o $x < y$ y $y = z$ o $y < z$, por lo tanto.

1. Si $x = y$ y $y = z$, entonces el tipo del intervalo x es el mismo tipo del intervalo z, porque el tipo del intervalo en único, y como $a = a'$, $a' = a''$ entonces $a = a''$, análogamente $b = b''$, luego $x = z$ y por lo tanto $x \preceq z$.

2. Si $x < y$ y $y = z$, entonces $x < z$ y por lo tanto $x \preceq z$.
3. Si $x = y$ y $y < z$, entonces $x < z$ y por lo tanto $x \preceq z$.
4. Si $x < y$ y $y < z$, entonces como dados $x, z \in \mathcal{I}$ se tiene por la propiedad de tricotomía que o $z = x$ o $z > x$ o $z < x$. Supongamos que $z < x$ entonces z y x deben satisfacer alguno de los casos de la definición 4.6, entonces analicemos cada uno de los casos:

   Caso 1: Supongamos que $a'' < a$, entonces como $x < y$ y $y < z$ se tiene por definición que $a \leq a'$ y $a' \leq a''$ es decir $a \leq a''$, lo que contradice que $a'' < a$.

   Caso 2: Supongamos que $a'' = a$, $a'' \in z$ y $a \notin x$, entonces como se debe tener que $a \leq a' \leq a''$, entonces $a = a' = a''$, ahora si $a' \in y$ entonces $y < x$ lo cual contradice que $x < y$, sino es así es decir $a' \notin y$ entonces $z < y$ lo que contradice que $y < z$.

   Caso 3: Supongamos que $a'' = a$, que a'' y a pertenecen o no simultáneamente a z y x respectivamente, y $b'' < b$. Se tiene por hipótesis que como $x < y$ entonces $a \leq a'$ y como $y < z$ entonces $a' \leq a''$, y como además $a'' = a$ entonces $a'' = a' = a$, por otro lado si $a \in x$ entonces $a' \in y$, si por el contrario $a \notin x$ entonces $a' \notin y$. Por los resultados anteriores se tiene que como $x < y$ entonces $b \leq b'$ y como $y < z$ entonces $b' \leq b''$, por lo tanto $b \leq b''$, pero lo anterior contradice que $b > b''$.

   Caso 4: Supongamos que $a'' = a$, que a'' y a pertenecen o no simultáneamente a z y x respectivamente, y que $b'' = b$. Razonando de forma análoga a la anterior, se tienen los siguientes resultados: $a'' = a' = a$, si $a \in x$ entonces $a' \in y$, si no, entonces $a' \notin y$, y como por hipótesis $b'' = b$ entonces $b'' = b' = b$, de lo anterior y como $z < x$ la única opción que queda es que $b'' \notin z$ y $b \in x$, pero si $b' \in y$ entonces $z < y$ lo que contradice que $y < z$, o si $b' \notin y$ entonces $y < x$ lo que contradice que $x < y$.

De los análisis hechos a los casos 1, 2, 3, 4 se tiene que si $x < y$, $y < z$ y $z < x$ entonces se encuentra contradicción en todos los casos y por lo tanto se tiene que $z \not< x$, es decir se debe tener que $z > x$ o $z = x$ de donde $x \preceq z$. Con lo cual se han abarcado todos los casos posibles y por lo tanto $\preceq$ es una relación transitiva. ■

4.10 Teorema. *Sea $\preceq$ la relación de orden dada en la definición 4.8, entonces, $\preceq$ define un orden total sobre $\mathcal{I}$. Es decir dados x y y en $\mathcal{I}$ se tiene que o $x \preceq y$ o $y \preceq x$.*

Demostración. Dado que por definición x es igual a y si y solo si son intervalos del mismo tipo y coinciden en sus extremos, entonces si no se tiene eso se debe verificar si $x < y$ con base en la definición de la desigualdad "menor que", y en caso de que no se tenga este caso se dice que $y < x$, entonces para cualesquiera intervalos x y y se tiene que $x = y$ o $x < y$ o $y < x$, entonces se tiene que se cumple siempre que o $x \preceq y$ o $y \preceq x$, por lo tanto se tiene que $\preceq$ es un orden total. ■

Capítulo 5

Manejo y uso de las extensiones de *JEP*

5.1. Ejemplo general de una extensión de *JEP* a funciones de $\mathbb{C}^n$ en $\mathbb{C}^m$

Realizar una extensión de *JEP* a funciones vectoriales es bastante fácil, pues las expresiones que *JEP* puede interpretar pueden tener un número arbitrario de variables; por lo tanto para realizar la generalización es suficiente trabajar con un arreglo que tenga el tamaño igual al número de funciones componentes.

En el siguiente ejemplo se ilustra una de las posibles formas de como se puede realizar dicha extensión tomando como ejemplo la función:

$$F : \mathbb{C}^3 \longrightarrow \mathbb{C}^4$$
$$(x, y, z) \longmapsto F(x, y, z) = (f_0, f_1, f_2, f_3) = (ze^x + e^y, xe^y \cos z, 2xy \tan z, x^2 + y^2 + z^2)$$

la cual se evaluará en el vector $(x, y, z) = (7 - 4i, -5 + 2i, 1 + \frac{1}{2}i)$.

```
import org.nfunk.jep.*;%
import org.nfunk.jep.type.*;

public class FuncionCnCmEjem {

    /** Crea un nueva instancia de FuncionCnCmEjem */
    public FuncionCnCmEjem() {
    }

    /**
     *  parámetros de los argumentos de la linea de comandos
     */
    public static void main(String[] args) {
        String nombre[] = {"x","y","z"};
```

```
        Complex vectorCn[] = {
                    new Complex(7,-4), new Complex(-5,2), new Complex(1,0.5)};
        String expresiones[] = {
                  "z*e^x+e^y", "x*e^y*cos(z)", "2*x*y*tan(z)", "x^2+y^2+z^2"};
        Complex vectorCm[] = new Complex[expresiones.length];
        System.out.println(
                     "Ejemplo de la extensión de JEP a funciones vectoriales");
        System.out.println("       F:Cn --> Cm");
        System.out.println("    (x,y,z) --> F(x,y,z)=(f_0,f_1,f_2,f_3)\n");
        JEP F[] = {new JEP(), new JEP(), new JEP(), new JEP()};
        int i,j;
        for(i=0; i < expresiones.length; i++) {
            F[i].addStandardFunctions();
            F[i].addStandardConstants();
            F[i].addComplex();
        }
        for(i=0; i < expresiones.length; i++)
            for(j=0; j < vectorCn.length; j++)
                F[i].addVariableAsObject(nombre[j], vectorCn[j]);
        for(i=0; i < expresiones.length; i++)
            F[i].parseExpression(expresiones[i]);
        for(i=0; i < expresiones.length; i++) {
            vectorCm[i] = F[i].getComplexValue();
        }
        for(i=0; i < expresiones.length; i++)
            System.out.println("f_"+ i + "(x,y,z) = " + vectorCm[i]);
    }
}
```

El resultado obtenido al ejecutar éste programa es:

```
Ejemplo de la extensión de JEP a funciones vectoriales
       F:Cn --> Cm
    (x,y,z) --> F(x,y,z)=(f_0,f_1,f_2,f_3)
f_0(x,y,z) = (-1131.7774277776412, 471.537203316802)           %
f_1(x,y,z) = (0.02669644880252003, 0.0308234953924039)         %
f_2(x,y,z) = (1.4451953035057272, -114.48386978422312)         %
f_3(x,y,z) = (54.75, -75.0)
```

5.2. Descripción del paquete `Intervalo`

El paquete `Intervalo` fue diseñado para poder trabajar con intervalos de números de máquina; en el capitulo 4.8 se definieron 4 tipos de intervalos, en este paquete se considera un tipo de intervalo adicional, los singleton, que son un caso particular de un intervalo cerrado donde el extremo inferior es igual al extremo superior.

La clase *Intervalo* es un TDA que tiene la siguiente configuración:

Tipo	Extremo Inferior	Extremo Superior	Cadena
(int)	*(double)*	*(double)*	*(String)*

Se compone de un entero que indica el tipo de intervalo en el cual clasifica, dos valores de doble precisión, uno que representa el extremo inferior y el otro que representa el extremo superior, y una cadena que es utilizada para representar con símbolos matemáticos al intervalo en estudio.

Un intervalo puede ser iniciado con base de una cadena que representa matemáticamente al intervalo. Para especificar los extremos de un intervalo se pueden utilizar números de máquina y las constantes[1] *pi, min* y *max* que representan la aproximación de π, el valor mínimo negativo y el máximo positivo representado en `Java`, los valores correspondientes son:

- pi = 3.141592653589793;
- min = -1.7976931348623157E308;
- max = 1.7976931348623157E308;

A continuación se realiza la descripción de las fases de un compilador que son utilizadas para realizar la interpretación de un intervalo representado por una cadena.

5.2.1. Análisis lexicográfico en intervalos

También conocido como *escáner*, es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada, agruparlos en secuencias de caracteres que forman un componente léxico (lexemas), clasificar estos en categorías y elaborar como salida una secuencia de componentes léxicos (tokens). Para el caso de un intervalo se clasifican los lexemas en cinco categorías:

Constantes: Son secuencias de caracteres alfabéticos (p. ej. *max, Min, PI*).

Signos de agrupación: Son los caracteres [,], (,).

Números sin signo: Son aquellos números de doble precisión sin signo (*double*), reconocidos por `Java` incluyendo números en notación científica o notación de punto flotante (p. ej. 2, 2.0, 3.1415, 5.89E-6, 0.0349).

[1] El interpretador de intervalos no es sensible a minúsculas y mayúsculas. Por ejemplo las siguientes cadenas {pi, Pi, pI, PI} representan la misma constante, y valores como 5.89E-6 y 5.89e-6 representan el mismo número de máquina.

Operadores: Son los caracteres que actúan como operadores binarios o unarios: $*$ para los binarios y $+$, $-$ para los unarios.

Signos de puntuación Son los caracteres utilizados para separar el extremo inferior y el superior en el intervalo, para este caso el único signo de puntuación es la coma ",".

Para el reconocimiento de los lexemas se utiliza un diagrama de transiciones; en la figura 5.1 se ilustra un autómata finito determinista (AFD) que permite visualizar como se puede realizar el reconocimiento. q_0 denota el estado inicial, las aristas denotan transiciones, las etiquetas los símbolos que se procesan, los círculos estados y los círculos dobles denotan los estados de aceptación.

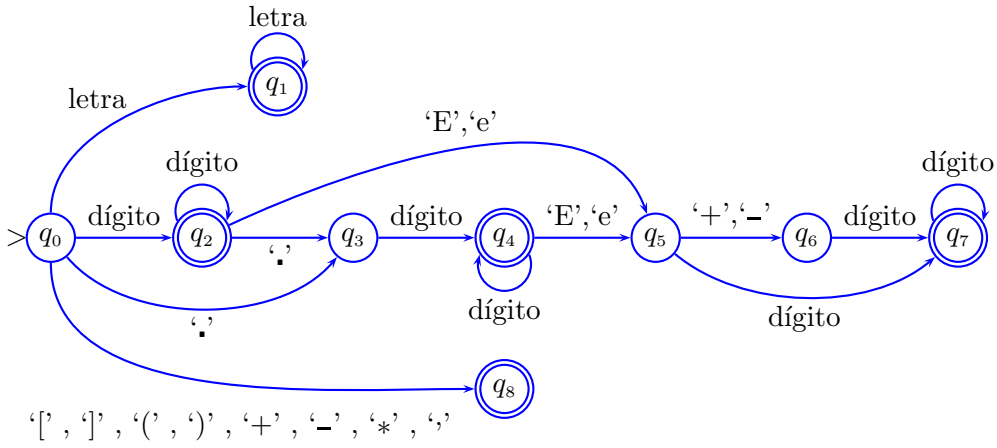


Figura 5.1: Diagrama de transiciones para el reconocimiento de lexemas.

La clase *Intervalo* del paquete `Intervalo` (adjunto), posee el método *escaner()* en el cual se implementa el analizador lexicográfico, la construcción esta basada en una rutina *switch* donde cada *case* representa un estado y una palabra es aceptada, es decir es un lexema, si se llega a un estado de aceptación, en caso contrario se lanza una excepción *IntervalException*.

5.2.2. Análisis sintáctico en intervalos

También conocido como *parsing* o *parser*, es el proceso de determinar si una cadena de componentes léxicos puede ser generada por una gramática. Para el caso de los intervalos estos tiene la misma notación usada en matemáticas, es decir es un número o tienen la forma:

⟨*signo de agrupación que abre*⟩ *expre_arit* ⟨,⟩ *expre_arit* ⟨*signo de agrupación que cierra*⟩

por ejemplo $(-1, 1)$, $[0, 2 * pi)$, $[min, max]$, (-2.0pi,2.0pi].

expre_arit es una expresión aritmética compuesta por un número o una expresión de la forma:

$$\langle unario \rangle \# * \langle cte \rangle$$

unario hace referencia a un operador unario opcional {+, −}, # representa un número sin signo de tipo *double*, ∗ representa el operador binario de multiplicación que es opcional, es decir, ésta puede ser explícita o implícita, y *cte* representa alguna de las constantes {max, min , pi}. Como ejemplos de expresiones aritméticas validas se tienen: 2.0, +5, 1E+10, −0.5*pi, 2pi, −max, min.

Para el análisis sintáctico generalmente se hace uso de una gramática independiente del contexto, para este caso se diseño un autómata finito determinista el cual es equivalente a una gramática independiente del contexto[2], el autómata se encuentra ilustrado en las figuras 5.2, 5.3, 5.4.

En la figura 5.2 se encuentra la parte del autómata con la cual se hace el reconocimiento de expresiones aritméticas. En la figura 5.3 se encuentra la parte del autómata con la cual se hace el reconocimiento del extremo inferior del intervalo y en la figura 5.4 se encuentra la parte del autómata con la cual se hace el reconocimiento del extremo superior del intervalo.

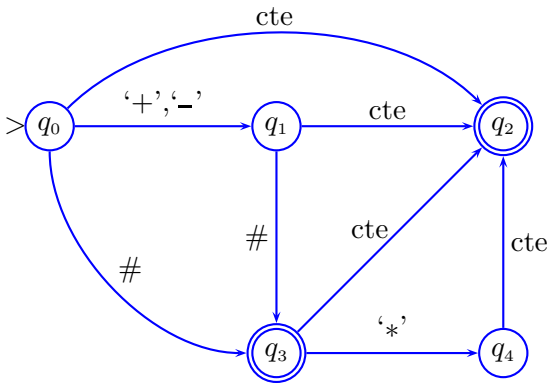


Figura 5.2: Diagrama de transiciones para el parsing de expresiones aritméticas

La clase *Intervalo* del paquete `Intervalo` (adjunto), posee el método *parsing()* en el cual se implementa el analizador sintáctico, la construcción esta basada en una rutina *switch* donde cada *case* representa un token a analizar y una secuencia de tokens es aceptada, es decir es un intervalo, si se llega a un estado de aceptación, en caso contrario se lanza una excepción *IntervalException*.

[2]Dado un autómata finito determinista AFD $M = (Q, \Sigma, q_0, F, \delta)$, existe una gramática independiente del contexto GIC regular $G = (V, \Sigma, S, P)$, tal que $L(M) = L(G)$. Para ver una demostración de éste resultado *véase* [Kor04].

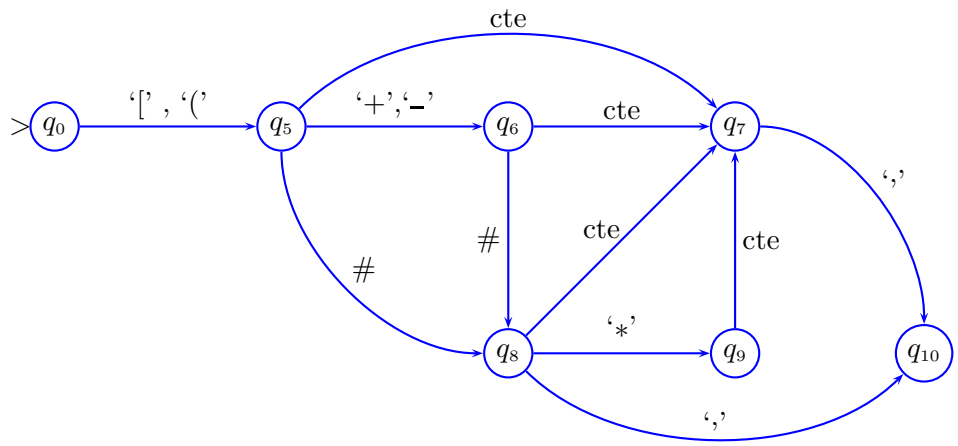


Figura 5.3: Diagrama de transiciones para el parsing del extremo inferior

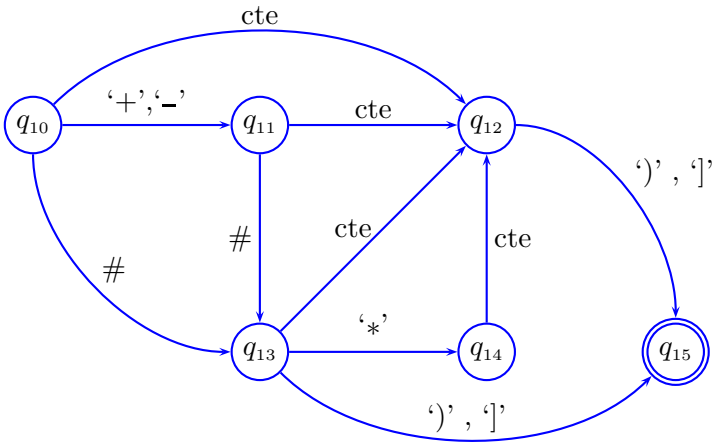


Figura 5.4: Diagrama de transiciones para el parsing del extremo superior

5.2.3. Notación posfija

Las personas generalmente escriben expresiones como $3+4$, $7 \wedge 9$ y $(6+2)*5-8/4$ en las que los operadores ($+$, $-$, $*$, $/$, $\wedge$) se escriben entre sus operandos; esto se denomina *notación infija*. La forma en que los compiladores evalúan expresiones aritméticas compuestas exclusivamente de constantes, operadores y paréntesis, es la *notación posfija* en la que el operador se escribe a la derecha de sus dos operandos. Las expresiones infijas anteriores se escribirían en notación posfija como $3\ 4\ +$, $7\ 9\ \wedge$ y $6\ 2\ +\ 5\ *\ 8\ 4\ /\ -$, respectivamente.

La *notación posfija* de una expresión E se puede definir de manera inductiva como sigue:

1. Si E es una variable o una constante, entonces la notación posfija de E es E.

2. Si E es una expresión de la forma $A \bigstar B$, donde $\bigstar$ es cualquier operador binario, entonces la notación posfija de E es $A'\ B'\ \bigstar$, donde A' y B' son las notaciones posfijas de A y B, respectivamente.

3. Si E es una expresión de la forma (A), entonces la notación posfija de A, es también la notación posfija de E.

La notación posfija no necesita paréntesis, porque la posición y el número de argumentos de los operadores permiten solo una decodificación de una expresión posfija.

Para evaluar una expresión infija compleja, lo primero que hace un compilador es convertir la expresión a notación posfija, y luego evaluaría la versión posfija de la expresión. Estos algoritmos requieren un solo recorrido de izquierda a derecha de la expresión. Cada algoritmo usa una estructura de datos (pila) para apoyar su funcionamiento, aunque en cada uno la pila se usa para un propósito distinto.

5.2.4. Análisis semántico en intervalos

En la fase del análisis semántico se identifican los operadores, operandos de expresiones y proposiciones. Para el caso de los intervalos primero se identifica si el intervalo es un singleton o no; en caso de ser un singleton se evalúa la expresión convirtiéndola primero a notación posfija y luego evaluándola mediante el uso de una pila; si no es un singleton, se obtienen las expresiones aritméticas que representan el extremo inferior y el extremo superior del intervalo, y luego se evalúan individualmente utilizando la notación posfija.

La clase *Intervalo* del paquete `Intervalo` (adjunto), posee los métodos *semantica()* y *evaluaExpresion()* en los cuales se implementa el analizador semántico, el primero sirve para determinar que clase de intervalo es y lo clasifica de acuerdo a su tipo, el segundo sirve para convertir expresiones aritméticas de notación infija a posfija y luego las evalúa.

5.2.5. Descripción de las clases del paquete `Intervalo`

Clases del paquete `Intervalo`	
Clase	Descripción
public class *Intervalo*	Esta clase es la principal del paquete `Intervalo`, es utilizada como un TDA que actúa como un intervalo en $\mathcal{M}$ cuyos elementos son números de máquina.
public class *Registro*	Es utilizada para crear los registros que forma la secuencia de tokens que se la pasan al analizador sintáctico y que son obtenidos al identificar cada lexema en el análisis léxico a la cadena que representa al intervalo.

Comparadores del paquete `Intervalo`	
public class *IntervalComparator*	Implementa la interfaz *Comparator*, en esta clase se sobre-escribe el método *compare* haciendo uso del orden definido en el capitulo 4.8. El fin de hacer el uso de este comparador es de realizar la ordenación total de un conjunto de intervalos, de forma que se pueda establecer si dados dos intervalos, estos son disyuntos o no.

Excepciones en el paquete `Intervalo`	
public class *IntervalException*	Se extiende de *Exception*, y es la excepción lanzada cuando se produce un error durante las fases en que se interpreta (análisis léxico o análisis sintáctico) la cadena que representa al intervalo.

Campos en la clase *Intervalo*	
private int *tipo*	Hace referencia al tipo de intervalo que encapsula este objeto.
private double *inferior*	Es el valor del extremo inferior del intervalo que representa este objeto.
private double *superior*	Es el valor del extremo superior del intervalo que representa este objeto.
private String *entrada*	Es la cadena que sirve para ilustrar la forma usual en que se notan los intervalos en matemáticas.
public static final int *SINGLETON*	Igual a 0, sirve para identificar a los intervalos que son singleton, tiene el formato $\{a\}$.
public static final int *ABIERTO*	Igual a 1, sirve para identificar a los intervalos que son abiertos, tiene el formato (a, b).
public static final int *SEMIABIERTO*	Igual a 2, sirve para identificar a los intervalos cuyo extremo inferior es abierto y el superior cerrado, $(a, b]$.
public static final int *SEMICERRADO*	Igual a 3, sirve para identificar a los intervalos cuyo extremo inferior es cerrado y el superior abierto, $[a, b)$.
public static final int *CERRADO*	Igual a 4, sirve para identificar a los intervalos que son cerrados, tiene el formato $[a, b]$.

Constructores definidos en la clase *Intervalo*	
public *Intervalo*()	Inicializa el objeto calificándolo de tipo *SINGLETON* y asigna a los extremos el valor 0.
public *Intervalo*(Intervalo intervalo)	Inicializa este objeto con los atributos propios del objeto *intervalo*
public *Intervalo*(double valor)	Inicializa el objeto calificándolo de tipo *SINGLETON*, y establece el extremo inferior y superior igual a *valor*

public *Intervalo*(int in_tipo, double in_inferior, double in_superior)	Este método tiene la capacidad de lanzar excepciones lanzando un objeto *IntervalException*, establece: *tipo = in_tipo*; *inferior = in_inferior*; *superior = in_superior*;
public *Intervalo*(String cadena)	Este método tiene la capacidad de lanzar excepciones lanzando un objeto *IntervalException*, el parámetro *cadena* es interpretado para poder saber en que tipo de intervalo clasifica y para obtener el extremo inferior y el extremo superior del intervalo representado por *cadena*.

Métodos definidos en la clase *Intervalo*	
public void *setTipo*(int tipo)	Establece el tipo de intervalo de este objeto, *tipo* debe cumplir que $0 \leq tipo \leq 4$, en caso contrario se lanza una excepción *IntervalException*.
public void *setInferior*(double inferior)	Establece el extremo inferior del intervalo igual a *inferior*; se debe cumplir que *inferior* debe se menor o igual que el extremo superior dependiendo del tipo de intervalo, en caso contrario se lanza un objeto excepción de tipo *IntervalException*.
public void *setSuperior*(double superior)	Establece el extremo superior del intervalo igual a *superior*; se debe cumplir que *superior* debe se mayor o igual que el extremo inferior dependiendo del tipo de intervalo, en caso contrario se lanza un objeto excepción de tipo *IntervalException*.
public int *getTipo*()	Retorna el tipo de intervalo de este objeto.
public double *getInferior*()	Retorna el extremo inferior del intervalo.
public double *getSuperior*()	Retorna el extremo superior del intervalo.
public boolean *pertenece*(double valor)	Retorna *true* si *valor* pertence al intervalo, en caso contrario retorna *false*.
public boolean *intercepta*(Intervalo intervalo)	Retorna *true* si éste objeto se intercepta con el objeto *intervalo*, en caso contrario retorna *false*.
public String *toString*()	Retorna la cadena que representa a este intervalo, en caso de que se haya pasado una cadena al constructor se retorna ésta cadena.

5.3. Descripción del paquete `FuncionTrozos`

Una función definida a trozos de $\mathbb{R}$ en $\mathbb{R}$, es una función de $\mathbb{R}$ en $\mathbb{R}$ en la cual se utilizas más de una expresión para poder determinarla de forma univoca. Éstas tienen grandes aplicaciones en economía, electrónica, estadística, ciencias sociales y muchas más ares del conocimiento, por lo tanto es importante que se pueda hacer una manipulación

computacional de éstas para poder dar solución a problemas específicos que se encuentran en las aplicaciones diarias de cada area en especifico.

En esta sección se estudia la clase *FuncionTrozos* que es una extensión de la clase ya estudiada *JEP*, y en la cual se realiza un trabajo con interpretes para poder utilizar la notación matemática usual al máximo, de tal manera que una función definida a trozos sea determinada por las expresiones y los intervalos sobre los cuales se definan dichas expresiones. Estos intervalos son los mismos que se estudian e interpretan en la clase *Intervalo* y que su notación es la misma que se usa en matemáticas.

A continuación se realiza un análisis del paquete `FuncionTrozos` y se hace énfasis en la clase *FuncionTrozos* que es la que el usuario usará generalmente en sus aplicaciones.

Clases del paquete `FuncionTrozos`	
Clase	Descripción
public class *ComponenteFuncion*	Esta clase es utilizada para crear una instancia de la clase *JEP* para cada uno de los trozos que componen la función definida a trozos. Se utiliza el constructor por defecto *JEP()* en el cual se especifica que la multiplicación es explícita y que la variables deben ser declaradas
public class *FuncionTrozos*	Ésta clase es utilizada para implementar una función definida a trozos y tiene los métodos con los cuales se evalúa la función y se hace el manejo de errores

Comparadores del paquete `FuncionTrozos`	
Comparador	Descripción
public class *TrozosComparador*	Implementa la interfaz *Comparator* y es utilizada para comparar y ordenar los trozos definidos en la clase *FuncionTrozos*. En esta clase se sobre-escribe el método *compare* de la interfaz *Comparator* utilizando el comparador *IntervalComparator* utilizado para comparar intervalos.

Constructores definidos en la clase *ComponenteFuncion*	
Constructor	Descripción
public *ComponenteFuncion*(String expresion, String intervalo, String variable)	Este constructor tiene la capacidad de dar lugar a excepciones, se utiliza el constructor por defecto de *JEP*, se adicionan la funciones estándar y las constantes estándar, se crea una instancia de la clase *Intervalo* pasándole al constructor el parámetro *intervalo* y se adiciona *variable* a la instancia de *JEP* inicializando la variable con el valor 0.

public *ComponenteFuncion*(ComponenteFuncion componente)	Inicializa una instancia de la clase *ComponenteFuncion* con los atributos propios del objeto *componente*.

Métodos definidos en la clase *ComponenteFuncion*	
Método	Descripción
public void *addVariable*(String name, double value)	Adiciona la variable *name* con el valor *value* al objeto *JEP* de éste objeto.
public double *getValue*()	Retorna el valor obtenido al evaluar la expresión pasada a la instancia de *JEP*
public Intervalo *getIntervalo*()	Retorna la instancia de la clase *Intervalo*.
public String *getExpresion*()	Retorna la expresión pasada a la instancia de *JEP*.
public JEP *getJEP*()	Retorna la instancia de *JEP* que se inicializó en este objeto.
public boolean *pertenece*(double value)	Retorna *true* o *false* si el valor *value* pertenece o no al intervalo especificado en este objeto.

Campos en la clase *FuncionTrozos*	
Campo	Descripción
private ComponenteFuncion[] *componente*	Arreglos que encapsula cada uno de los trozos (expresión e intervalo) que sirven para definir la función.
private int *numTro*	Guarda el número de expresiones utilizadas para definir la función.
private Vector *errorList*	Estructura de datos donde se guardan la excepciones y errores que puedan ocurrir durante le análisis sintáctico o la evaluación de la expresión. En condiciones normales debe estar vacía.
private String *nombre*	Es el nombre con el cual se imprimirá la función. El valor por default es "*F*".
private String *variable*	Es el nombre de la variable utilizado en las expresiones.

Constructores definidos en la clase *FuncionTrozos*	
Constructor	Descripción
public *FuncionTrozos*(String[][] funciones, String variable)	Inicializa un arreglo de *ComponenteFuncion* del tamaño igual al número de funciones pasadas a éste constructor, adiciona la variable *variable* a cada una de las componentes, ordena el arreglo de acuerdo al orden de los intervalo y revisa que los intervalos sean disyuntos asegurando que efectivamente la expresión pasada es una función.

public FuncionTrozos(String[][] funciones, String variable, String nombre)	Realiza las mismas acciones que el constructor anterior y especifica que el nombre utilizado para imprimir la función sea *nombre*.

Métodos definidos en la clase *FuncionTrozos*	
Método	Descripción
public double *evaluar*(double valor)	Retorna el valor obtenido al evaluar la función a trozos. Si ocurrió un error durante la evaluación o el valor *valor* no pertenece a alguno de los intervalos definidos, entonces se retorna *Math.NaN* (Not a number).
public void *setNombre*(String nombre)	Establece el nombre de la función utilizado para imprimir la función.
public String *getNombre*()	Retorna el nombre de la función utilizado para imprimir la función. El valor por defecto es "f".
public int *getNumeroTrozos*()	Retorna el número de trozos que definen la función.
public ComponenteFuncion *componenteNumero*(int n)	Retorna la instancia de *ComponenteFuncion* especificada por el parámetro n, si no existe dicho objeto se retorna *null* y se carga una excepción a la lista de errores.
private void *revisarIntercepcion*()	Este método tiene la capacidad de lanzar excepciones, revisa si existe un par de intervalos que no sean disyuntos, en tal caso se carga una excepción a la lista de errores.
public boolean *hasError*()	Retorna *true* si la lista de errores no se encuentra vacía, en caso contrario se retorna *false*.
public String *getErrorInfo*()	Si la lista de errores no se encuentra vacía, se retorna una cadena (*String*) con la lista de errores, en caso contrario se retorna *null*.
public String *toString*()	Retorna una cadena que sirve para imprimir la función definida a trozos con la cual se esta trabajando.

5.3.1. Ejemplo general del uso de *FuncionTrozos*

La clase que se muestra a continuación es un ejemplo clásico del uso de la clase *FuncionTrozos* para implementar la función de definida a trozos de la figura 5.5. Esta es la forma correcta de utilizar *FuncionTrozos* en la mayoría de los casos. Es importante notar que la multiplicación es explícita, la especificación de los trozos y sus intervalos respectivos debe hacerse en un arreglo multidimensional de $n \times 2$, donde n es el número de trozos, la variable es adicionada en el constructor, el nombre es especificado como "f", la revisión de errores se hace en cada posible caso en que puedan ocurrir y los intervalos se pasan desordenados.

$$f(x) = \begin{cases} x^2 - 1; & \text{si } -1 \leq x < 0, \\ 0; & \text{si } x = 0, \\ x + 1; & \text{si } 0 < x \leq 1, \\ -2x + 4; & \text{si } 1 < x < 2, \\ x - 2; & \text{si } 2 \leq x \leq 3, \end{cases}$$

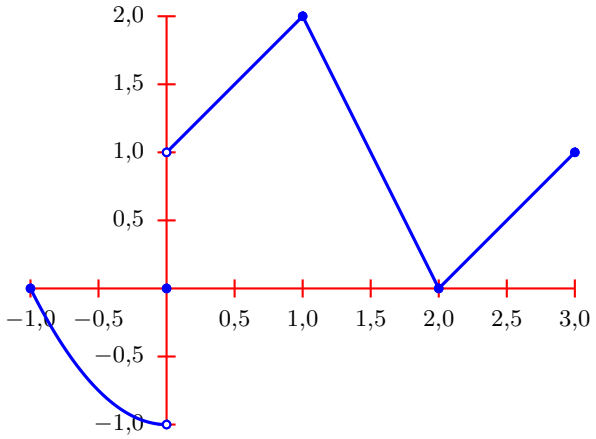


Figura 5.5: Ejemplo de una función definida a trozos

```
import java.util.*;%
import org.nfunk.jep.*;%
import FuncionTrozos.*;%
import Intervalo.*;

public class FuncionTrozosEjem {

    public FuncionTrozosEjem() {
    }

    public static void main(String[] args) {
        // definición de los trozos de la función con sus respectivos
        // intervalos de dominio
        String trozos[][] = {
            {"x-2","[2,3]"},
            {"x+1","(0,1]"},
            {"x^2-1","[-1,0)"},
            {"0","0"},
            {"-2*x+4","(1,2)"}
        };
        // crea una nueva instancia de FuncionTrozos con la
        // variable: "x", y nombre: "f"
        FuncionTrozos f = new FuncionTrozos(trozos,"x","f");
        // revisar si han ocurrido errores durante el análisis
        // de la función a trozos
        System.out.println(
                "Ejemplo del uso de FuncionTrozos con la función:\n" + f);
        if(f.hasError()){
            System.out.println("Error durante el análisis sintáctico");
            System.out.print(f.getErrorInfo());
        }
        //evalua la funcion a trozos en el valor dado
```

```
        double x = 1.5;
        double valor = f.evaluar(x);
        // revisar si han ocurrido errores durante la evaluación de la
        // función a trozos
        if(f.hasError()){
            System.out.println("Error durante la evaluación");
            System.out.print(f.getErrorInfo());
        }
        //imprime el resultado de la evaluación en la consola
        System.out.println(f.getNombre() + "(" + x + ")= " + valor);
    }
}
```

El resultado obtenido al ejecutar éste programa es:

```
Ejemplo del uso de FuncionTrozos con la función:%
    f(x) = {
            x^2-1;          [-1,0)
            0;              {0.0}
            x+1;            (0,1]
            -2*x+4;         (1,2)
            x-2;            [2,3]
           }
    f(1.5)= 1.0
```

5.3.2. Ejemplos del uso de los métodos de *FuncionTrozos*

Uso de *public double evaluar(double valor)*

Al hacer uso de este método en la clase del ejemplo de la sección 5.3.1 se obtiene como resultado:

```
f(1.5)= 1.0
```

en caso de que el valor en el cual se quiera evaluar la función no pertenezca a ninguno de los intervalos, entonces se imprimirá *NaN*.

Uso de *public String getNombre()*

Si se hace uso de este método en el ejemplo de la sección 5.3.1 entonces se retornaría el nombre actual de la función, en este caso sería "f", pero si no se especifica el nombre con el constructor o no se utiliza el método *setNombre(String nombre)* entonces el nombre retornado sería el nombre por defecto "F".

Uso de *public int getNumeroTrozos()*

Retorna el número de expresiones utilizadas para realizar la definición de la función. Para el caso del ejemplo de la sección 5.3.1 el valor retornado seria 5.

Uso de *public void setNombre(String nombre)*

Al hacer uso de este método en la clase del ejemplo de la sección 5.3.1, se cambia el nombre que poseía la función y se le asigna *nombre*, si por ejemplo se utilizará este método especificando el nombre como "*g*", inmediatamente después del constructor, de la siguiente forma:

```
f.setNombre("g");
```

entonces se obtendría como resultado en la consola la siguiente expresión:

```
Ejemplo del uso de FuncionTrozos con la función:%
    g(x) = {
            x^2-1;          [-1,0)
            0;              {0.0}
            x+1;            (0,1]
            -2*x+4;         (1,2)
            x-2;            [2,3]
           }
    g(1.5)= 1.0
```

Uso de *public String toString()*

Este es un método heredado de la clase *Object*, en esta clase se sobre-escribió para ser utilizado para realizar la impresión este objeto. Por ejemplo si a la clase del ejemplo de la sección 5.3.1 se le adicionará la linea

```
System.out.println(f);
```

se obtendría en la consola impreso la siguiente expresión, donde los intervalos están ordenados.

```
f(x) = {
        x^2-1;          [-1,0)
        0;              {0.0}
        x+1;            (0,1]
        -2*x+4;         (1,2)
        x-2;            [2,3]
       }
```

Uso de *public ComponenteFuncion componenteNumero(int n)*

Retorna el objeto que encapsula la expresión n que sirve para definir la función después de haber ordenado las expresiones con respecto a los intervalos. En caso de que no exista la expresión número n entonces se retorna *null* y se carga una excepción a la lista de errores.

Uso de *public boolean hasError()*

Este método retorna *true* si ha ocurrido un error durante el análisis sintáctico o la evaluación de la función definida a trozos en caso contrario retorna *false*. Por ejemplo si a la clase del ejemplo de la sección 5.3.1, al arreglo en el que se definan la expresiones en vez de pasar la cadena `‘‘-2*x+4’’` se pasara la cadena `‘‘-2x+4’’`, entonces este método retornaría *true*.

Uso de *public String getErrorInfo(*

Retorna el informe de los errores ocurridos durante la más reciente acción (análisis sintáctico o evaluación). Por ejemplo en el apartado anterior se dio un ejemplo de un error, en este caso se imprimirá las siguientes líneas.

```
Error durante el análisis sintáctico    %
null                                    %
Error durante la evaluación             %
Syntax Error (implicit multiplication not enabled)
```

5.3.3. Ejemplo de una gráfica utilizando la clase *FuncionTrozos*

En la figura 5.6 se muestra un `applet` de `Java` en el cual se hace la gráfica de la función:

$$f(x) = \begin{cases} 2x+3; & \text{si } -2 \leq x < -1, \\ x^2; & \text{si } -1 \leq x \leq 0, \\ \sqrt{x}; & \text{si } 0 < x < 1, \\ \dfrac{1}{x^2}; & \text{si } 1 \leq x \leq 2, \end{cases}$$

haciendo uso de la clase *FuncionTrozos*.

A continuación se encuentra el código del archivo `GraficaTrozos.java` necesario para poder ejecutar este `applet`.

```
import javax.swing.*; %
import java.awt.*; %
import java.awt.geom.*; %
import FuncionTrozos.*;
```

```
public class GraficaTrozos extends javax.swing.JApplet {

    /** Creates a new instance of GraficaTrozos */
    public GraficaTrozos() {
    }

    Graphics2D g2;
    double a, b, x, y;
    double xMax = 2, xMin = -2, yMax = 2, yMin = -1;
    int particion = 100;
    double Dx = (xMax-xMin)/particion;
    FuncionTrozos funcion;

    public void init() {
        this.setSize(800,400);
         String trozos[][] = {
            {"2*x+3","[-2,-1)"},
            {"x^2","[-1,0]"},
            {"sqrt(x)","(0,1)"},
            {"1/x^2","[1,2]"},
        };
        funcion = new FuncionTrozos(trozos,"x");
    }

    public void paint(Graphics g) {
        g2 = (Graphics2D) g;
        g2.setRenderingHint(
              RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setColor(Color.RED);
        Line2D linea = new Line2D.Double(
                                  coordX(xMin),coordY(0),coordX(xMax),coordY(0));
        g2.draw(linea);
        linea.setLine(coordX(0),coordY(yMax),coordX(0),coordY(yMin));
        g2.draw(linea);
        g2.setColor(Color.BLUE);
        g2.setStroke(new BasicStroke(1.3f));
        a = xMin;
        b = funcion.evaluar(a);
        x = a + Dx;
        y = funcion.evaluar(x);
        for(int i = 0; i <= particion; i++){
            linea.setLine(coordX(a),coordY(b),coordX(x),coordY(y));
            g2.draw(linea);
            a=x;
            b=y;
            x+=Dx;
```

```
            y=funcion.evaluar(x);
        }
        this.showStatus(
                "Dimensión: (" + this.getWidth() +", "+ this.getHeight() + ")");
    }

    public static void main(String s[]) {
        JFrame lienzo = new JFrame("Grafica de una función definida a trozos");
        JApplet applet = new GraficaTrozos();
        lienzo.getContentPane().add(applet);
        applet.init();
    }

    public double coordX(double x) {
         return (this.getWidth()/(xMax-xMin))*(x-xMin);
    }

    public double coordY(double y) {
         return  (-(this.getHeight()/(yMax-yMin))*(y-yMax)+this.getWidth()) -
                                                              2*this.getHeight();
    }
}
```

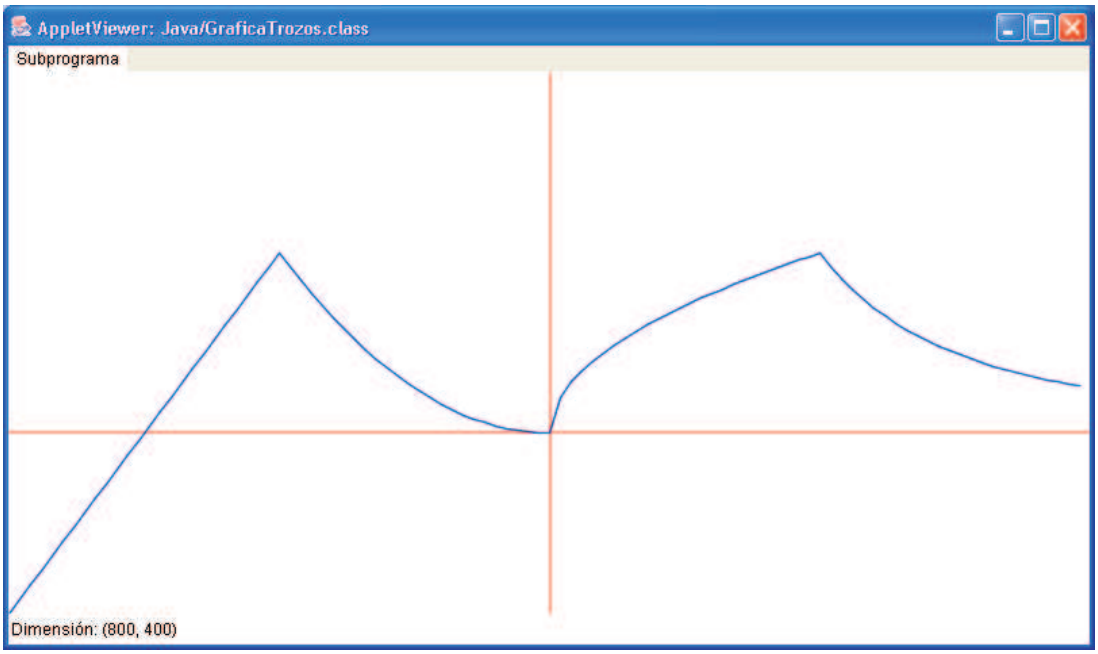


Figura 5.6: Gráfica de una función definida a trozos

Conclusiones

- El paquete `JEP` es una herramienta poderosa que permite la evaluación de expresiones matemáticas que incluyan las funciones y constantes matemáticas más comúnmente utilizadas, de una forma optima, confiable y eficiente.
- El paquete `JEP` es una herramienta popular en la comunidad científica, pues es utilizada por varias instituciones prestigiosas, entre las que se incluye la NASA.
- El paquete `JEP` a pesar de ser una herramienta de programación excelente, tiene algunas deficiencias y contradice en algo la filosofía de *Java*, por ejemplo si ocurre algún error durante la evaluación, el valor retornado por *getValue* es 0 lo que puede generar errores en los resultados, y el manejo de excepciones se realiza por medio de condicionales en vez de atrapar las excepciones lanzadas.
- Como el paquete `JEP` es de libre uso, entonces es de esperar que con este trabajo se haga una divulgación entre los estudiantes y profesores de la universidad con el fin de motivar la investigación científica y el trabajo en aplicaciones matemáticas.
- Que existe gran cantidad de código libre del cual se puede hacer uso para resolver problemas en matemáticas y que es necesario conocer y utilizar, pues esto se refleja en ahorro de tiempo y esfuerzo para resolver un problema en especifico.
- Que es necesario revaluar los cursos de programación que se dictan en la carrera de matemáticas con el fin de enseñar a los estudiantes a programar en lenguajes más modernos y sofisticados, que están al alcance de cualquier persona ya que son de libre uso, tal es el caso del lenguaje de programación `Java`.
- Que es necesario tomar en serio por parte de los directivos de la universidad la creación del departamento de ciencias de la computación; pues con respecto a latinoamérica y el resto mundo estamos bastante atrasados en este tema tan fundamental hoy en día y el cual nos podría a ayudar a salir del subdesarrollo en el cual nos encontramos inmersos.

- Que la redacción de trabajos escritos es importante para los estudiantes que estamos interesados en realizar investigación científica seria, con el fin de poder publicar los resultados obtenidos de una forma clara y concisa.
- Que es importante la realización de un trabajo de grado al culminar la carrera, pues ésta es nuestra primera experiencia en el desarrollo de investigaciones que tengan un objetivo especifico.
- Que lo conocimientos adquiridos al realizar éste trabajo son importantes pues logre hacer la fusión y aplicación de las asignaturas de la linea de informática a otras asignaturas de matemática pura y con otras areas del conocimiento que no son estudiadas en la carrera, por ejemplo la teoría de los compiladores.
- Que es necesario saber manejar el programa LaTeX por parte de los matemáticos y científicos en general, pues esta herramienta ya es un estándar en la divulgación de texto científicos.

Apéndice A

Glosario de los principales términos de Java

Abstracción: La abstracción con respecto a la (POO), es una filosofía en la cual se ignoran los detalles de la implementación de un objeto, y en su lugar se utiliza libremente un objeto como un todo.

Java API: La interfaz de programación de aplicaciones Java, es un conjunto de clases y paquetes de Java.

Atributo: es el conjunto de datos definidos en una clase. También se conocen como campos, variables de instancia, variables miembro ó variables ejemplar.

Clase: Es una construcción lógica de tipos definidos por el programador, cada uno de ellos contiene datos, además de un conjunto de interfaces que manipulan los datos. Los componentes de datos de una clase se denominan atributos (variables de instancia, miembro ó ejemplar), y las interfaces se denominan métodos ó métodos miembro.

Clase abstracta: Cuando una clase es declarada como `abstrac`, indica que en la clase hay métodos declarados pero que que no se encuentran implementados en ésta, entonces dichos métodos deben ser implementados en las subclases de esta clase.

Clase de envoltura: Es una clase que nos permite manipular variables de los tipo primitivos como objetos de la clase *Object*.

Comparador: Es un objeto extendido de la interfaz *Comparator* que sirve para comparar objetos, en éste se sobre-escribe el método *int compare(Object obj1, Object obj2)*, de tal manera que se retorne cero si los objetos son iguales, un valor positivo si *obj1* es mayor que *obj2*, y en cualquier otro caso, un valor negativo.

Constructor: Tiene el mismo nombre que la clase en la que reside, y sintácticamente es similar a un método. Su función es la de inicializar las variables en el momento de la creación de una instancia de la clase.

Encapsulado: Es el mecanismo que permite unir el código junto con los datos que manipula, y mantiene a ambos a salvo de las interfaces exteriores y de uso indebido.

Error: Es una condición excepcional no esperada por el programa en condiciones normales, ésta es generalmente catastrófica y se encuentra generalmente ligada con problemas en el hardware. Un ejemplo de un error es aquel que ocurre cuando se produce el desbordamiento de una pila (overfull).

Estructura de datos: Es la implementación física de un tipo de dato abstracto que obedece a las leyes de la informática.

Excepción: Es un objeto que describe una condición anormal que surge en una parte del código en tiempo de ejecución.

Fábrica: o factoría, es una interfaz que sirve para crear familias de objetos relacionados o dependientes sin especificar su clase concreta, dejando a las subclases decidir el tipo específico al que pertenecen.

Final: Es un modificador al cual puede aplicarse a métodos, atributos y clases; en el caso de los métodos el modificador indica que el método no puede ser sobre-escrito, en los atributos éste indica que el atributo es declarado como una constante (es el análogo en C/C++ a *const*), en la clases tiene tarea de evitar que una clase sea heredada.

Flujos: Los flujos son secuencias ordenadas de datos que tienen una fuente (flujo de entrada) ó un destino (flujo de salida).

Herencia: Es el proceso por el cual un objeto adquiere la propiedades de otro, es decir: es cuando se da la creación de nuevas clases a partir de clases ya existentes absorbiendo sus atributos y métodos, y adornándolos con capacidades que las nuevas clases requieren.

Interfaz: Son sintácticamente semejantes a las clases, pero carecen de atributos y sus métodos se declaran pero sin ningún cuerpo, su finalidad es que cuando una clase implemente esta interfaz se deben definir obligatoriamente los métodos declarados en la interfaz con sus respectivos cuerpos.

Método: Es el código que opera y manipula los datos de una clase, definiendo cómo se pueden utilizar estos datos.

Objeto: Es una instancia física que adquiere la estructura y comportamientos definidos por una clase.

Paquete: Es una agrupación mediante subdirectorios del disco en categorías de clases relacionadas entre si.

Polimorfismo: Es una característica que permite que una interfaz sea utilizada por una clase general de acciones, esto significa que es posible diseñar una interfaz genérica para un grupo general de acciones relacionadas entre si.

Private: Es un especificador de acceso el cual se puede aplicar a métodos y atributos; éste indica que se puede acceder al elemento declarado como de este tipo, solo dentro de la clase a la que pertenece.

Programación orientada a objetos (POO): Es una metodología de programación, la cual ayuda a organizar programas complejos mediante el uso de la herencia, el encapsulamiento, y el polimorfismo.

Protected: Es un especificador de acceso el cual se puede aplicar a métodos y atributos; éste indica que se puede acceder al elemento declarado como de este tipo solo desde las subclases de la clase a la que pertenece y dentro de las clases del paquete al que pertenece.

Public: Es un especificador de acceso el cual se puede aplicar a clases métodos y atributos; éste indica que se puede acceder al elemento declarado como de éste tipo desde cualquier parte del programa.

Serialización: La serialización es el proceso de leer y escribir el estado de un objeto en un flujo de bytes.

Static: Es un especificador de acceso el cual se puede aplicar a métodos y atributos; en el caso del uso de un método este se invoca sin necesidad a que se refiera a un objeto especifico y en el caso de un atributo estos son esencialmente, variables globales, es decir, no se hace una copia de este atributo para cada instancia sino que todas las instancias comparten el mismo atributo.

Subclase: Es una clase que hereda, es decir la subclase de una clase es aquella clase que se extiende de dicha clase.

Superclase: Es una clase que es heredada, es decir la superclase de una clase, es aquella clase de la cual se extiende dicha clase.

Tipo de dato abstracto (TDA): Es un tipo de datos definido de forma única mediante un tipo y un conjunto dado de operaciones definidas sobre el tipo.

Tipos de datos: Un tipo de datos es una colección de valores. Se denominan tipos de datos **escalares** a aquellos en los que el conjunto de valores está ordenado y cada valor es atómico: Carácter, Entero, Real y Booleano. Se denominan tipos de datos **ordinales** a aquellos en los que cada valor tiene un predecesor (excepto el primero),

y un único sucesor (excepto el último): Carácter, Entero y Booleano. Otro tipo de datos son los denominados **compuestos** (Conjuntos, Arreglos) dado que son divisibles en componentes que pueden ser accedidos individualmente; las operaciones asociadas a los tipos compuestos o estructurados son las de almacenar y recuperar sus componentes individuales.

Tipos primitivos de `Java`: Son los tipos de datos más elementales que se definen en `Java`, los cuales son: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, **boolean**.

Nombre	No. bits	Rango
byte	8	-128 a 127
short	16	-32.768 a 32.767
int	32	-2.147.483.648 a 2.147.483.647
long	64	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	32	-3.4e+038 a 3.4e+038
double	64	-1.7e+308 a 1.7e+308

Tabla A.1: Los tipos de datos primitivos en `Java`

Apéndice B

Gramática que especifica la sintaxis de la herramienta JEP

Esta gramática se especifica en el archivo `Parser.jjt`, con el cual se generan las clases que permiten realizar el análisis sintáctico recursivo mediante el uso de `JavaCC`. En la construcción se tuvo en cuenta que los operadores fueran analizados de acuerdo a la prioridad de precedencia (de menor a mayor prioridad).

$$
\begin{aligned}
\texttt{Start} &::= \texttt{(Expression}\langle EOF\rangle|\langle EOF\rangle\texttt{)}\\
\texttt{Expression} &::= \texttt{OrExpression}\\
\texttt{OrExpression} &::= \texttt{AndExpression((``||''AndExpression))}^{*}\\
\texttt{AndExpression} &::= \texttt{EqualExpression((``\&\&''EqualExpression))}^{*}\\
\texttt{EqualExpression} &::= \texttt{RelationalExpression((``!=''RelationalExpression)}\\
&\quad \texttt{|(``==''RelationalExpression))}^{*}\\
\texttt{RelationalExpression} &::= \texttt{AdditiveExpression((``<''AdditiveExpression)}\\
&\quad \texttt{|(``>''AdditiveExpression)}\\
&\quad \texttt{|(``<=''AdditiveExpression)}\\
&\quad \texttt{|(``>=''AdditiveExpression))}^{*}\\
\texttt{AdditiveExpression} &::= \texttt{MultiplicativeExpression(}\\
&\quad \texttt{(``+''MultiplicativeExpression)}\\
&\quad \texttt{|(``-''multiplicativeExpression))}^{*}\\
\texttt{MultiplicativeExpression} &::= \texttt{UnaryExpression((PowerExpression)}\\
&\quad \texttt{|(``*''UnaryExpression)|(``/''UnaryExpression)}\\
&\quad \texttt{|(``\%''UnaryExpression))}^{*}
\end{aligned}
$$

```
                UnaryExpression ::= (‘‘+’’UnaryExpression)|(‘‘−’’UnaryExpression)
                                    |(‘‘!’’UnaryExpression)|PowerExpression
                PowerExpression ::= UnaryExpressionNotPlusMinus(
                                    (‘‘∧’’UnaryExpression))?
UnaryExpressionNotPlusMinus ::= AnyConstant|(Function|Variable)
                                    |‘‘(’’Expression‘‘)’’
                    Variable ::= (Identifier)
                    Function ::= (Identifier‘‘(’’ArgumentList‘‘)’’)
                ArgumentList ::= (Expression(‘‘,’’Expression)*)?
                  Identifier ::= ⟨IDENTIFIER⟩
                AnyConstant ::= (⟨STRING_LITERAL⟩|RealConstant|Array)
                       Array ::= ‘‘[’’RealConstant(‘‘,’’RealConstant)*‘‘]’’
                RealConstant ::= (⟨INTEGER_LITERAL⟩|⟨FLOATING_POINT_LITERAL⟩)
```

Apéndice C

Teoría de los números de máquina

[1]Cada número real x puede ser representado en un sistema numérico de base $B \in \mathbb{Z}^+$, en la forma

$$\begin{aligned} x &= \pm \left(\sum_{i=0}^{\infty} a_i B^{-i} \right) \cdot B^L \\ &= \pm 0.a_1 a_2 \ldots \cdot B^L \text{ con } a_i \in \{0, 1, 2, \ldots, B-1\} \text{ para } i = 1(1) \ldots, \ L \in \mathbb{Z}. \end{aligned}$$

Esta representación se llama *de punto flotante*, más exactamente la llamamos "representación de punto flotante de x para la base B". L se llama *exponente* y la cadena $a_1 a_2 \ldots$ se llama *mantisa.*

Si $x \neq 0$, a través de cambios en el exponente L puede lograrse siempre que la primera cifra de la mantisa no sea cero; así por ejemplo, $0{,}0001 \cdot B^L = 0{,}1 \cdot B^{L-3}$. De esta forma se obtiene una representación de punto flotante *normalizada.*

Ejemplo Una representación en forma de punto flotante normalizada del número $x = 46{,}5$ para la base 10 es:

$$x = \left(\sum_{i=0}^{\infty} a_i 10^{-i} \right) \cdot 10^L = 0.a_1 a_2 a_3 \ldots \cdot 10^L = 0{,}4650\overline{0} \ldots \cdot 10^2,$$

que se acostumbra simplificar escribiendo solamente

$$x = 0{,}465 \cdot 10^2.$$

Pero ésta no es la única representación de punto flotante normalizada de x para la base 10. En efecto, también podemos escribir

$$x = 0{,}4649\overline{9} \ldots \cdot 10^2.$$

[1]Tomado de la serie *Notas de clase*, "Análisis Numérico", *véase*[Man04].

La forma de punto flotante es la usada para representar los números en un computador, pero con dos decisivas restricciones debidas a la limitada capacidad de memoria disponible: el rango para el exponente y la longitud de la mantisa.

C.1 Definición (números de máquina). El conjunto $\mathcal{M}$ de los números, en forma de punto flotante normalizada que pueden ser representados en un computador se llama conjunto de números de máquina. $\mathcal{M}$ depende de la base B, de la longitud de la mantisa M, y del rango para el exponente $\{-k, -k+1, \ldots, K-1, K\}$, siendo $k, K \in \mathbb{Z}^+$. Explícitamente,

$$\begin{aligned}\mathcal{M} &= \mathcal{M}(B, M, -k, K) \\ &= \{0\} \cup \big\{0.a_1a_2\ldots a_M \cdot B^L : a_1, a_2, \ldots, a_M \in \{0, 1, \ldots, B-1\}, \\ &\quad a_1 \neq 0,\ L \in \{-k, -k+1, \ldots, K-1, K\}\big\}.\end{aligned}$$

El conjunto $\mathcal{M}$ de números de máquina es un conjunto finito, contenido como subconjunto propio en el conjunto $\mathbb{Q}$ de los números racionales. El computador sólo puede calcular con números de $\mathcal{M}$ y sólo puede arrojar como resultados, números de $\mathcal{M}$. Ésta es, de partida, una gran limitación y una enorme fuente de errores. Cualquier máquina, incluso aún no inventada por el hombre, tendrá estas limitaciones.

Ejemplo Construir explícitamente la máquina más sencilla $\mathcal{M}(2, 2, -2, 2)$.

Teniendo en cuenta la definición anterior,

$$\mathcal{M} = \{0\} \cup \{\pm 0{,}1a \cdot 2^L : a \in \{0, 1\},\ L \in \{-2, -1, 0, 1, 2\}\}.$$

El menor número positivo de $\mathcal{M}(2, 2, -2, 2)$ es

$$x_{\text{mín}} := 0{,}10_2 \cdot 2^{-2} = \left(1 \cdot 2^{-1} + 0 \cdot 2^{-2}\right) \cdot 2^{-2} = \frac{1}{8}$$

y el mayor número positivo de la máquina es

$$x_{\text{máx}} := 0{,}11_2 \cdot 2^2 = \left(1 \cdot 2^{-1} + 1 \cdot 2^{-2}\right) \cdot 2^2 = 3.$$

El conjunto

$$\mathcal{M}^+ := \{\frac{1}{8}, \frac{3}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 3\}$$

contiene todos los números positivos de $\mathcal{M}$ escritos en forma ascendente. Como se observa, su distribución en la recta real no es uniforme, cerca al origen hay mayor densidad y los números mayores que 3 no pueden ser dominados por esta "mini$_2$-máquina". Si en algún proceso de cálculo se sobrepasa este valor máximo $x_{\text{máx}}$, se produce un error conocido con el nombre de **overflow** y el proceso se detiene. Los números reales $x : \frac{1}{8} < x < 3$ que no pertenecen a $\mathcal{M}_+$ se aproximan (o se "arrastran") al siguiente número (a la derecha) de máquina más cercano, evitando así detenciones debidas a nuevos errores. Por ejemplo, $2{,}6 \to 3$. De igual manera, los números negativos entre -3 y $-\frac{1}{8}$ que no están en la máquina se aproximan al anterior número (a la izquierda), por ejemplo $-0{,}7 \to -0{,}75 = -\frac{3}{4}$. Los números reales que se encuentran en el intervalo $(-\frac{1}{8}, \frac{1}{8})$ son reemplazados por cero, por ejemplo $0{,}12 \to 0$, pero la máquina no detiene el proceso de cálculo cuando hace estos "arrastres". En este caso hablamos de **underflow**.

Apéndice D

Instalación del paquete JEP

1. Descargar el paquete `JEP`. Puede descargar la ultima version de la página de internet `http://www.singularsys.com/download/jep-2.24.zip` ó utilizar la versión 2.24 que se encuentra guardada en el CD adjunto.

2. Descomprimir el archivo `jep-2.24.zip`.

3. Mover el archivo `jep-x.xx.jar` al directorio optativo que usted haya elegido para realizar su programa.

4. Adicionar a la variable `CLASSPATH` la ubicación del archivo `jar` en su PC, por ejemplo `C:\java\paquetes\jep-x.xx.jar`

Bibliografía

[AJ95] Sandy Anderson and Priyantha Jayanetti. *org.netlib.math.complex*. Se puede obtener en la página web http://www.nr.com, 1995.

[ASU90] Alfred V. Aho, Ravi Sethi, and Jeffrey Ullman. *Compiladores Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, 1990.

[BG02] Sara Baase and Allen Van Gelder. *Algoritmos computaciones: introducción al analisis y diseño*. Pearson Educatión, tercera edition, 2002.

[CB92] Ruel V. Churchill and James Ward Brown. *Variable Compleja y sus aplicaciones*. McGraw-Hill, quinta edition, 1992.

[DD98] Harvey M. Deitel and Paul J. Deitel. *Cómo programar en Java*. Prentice Hall, 1998.

[Fun02] Nathan Funk. *JEP - Java Mathematical Expression Parser*. Es un evaluador de expresiones matemáticas del cual se puede encontrar el código y la documentación completa de la página web http://www.singularsys.com/jep/index.html, 2002.

[Hen72] Peter Henrici. *Elementos de Análisis Numérico*. Trillas S.A., 1972.

[JHM02] J.D. Ullman J.E. Hopcroft and R. Motwani. *Introducción a la teoría de autómatas, lenguajes y computación*. Pearson Education S.A., segunda edition, 2002.

[Kor03] Rodrigo De Castro Korgi. *El universo LaTeX*. Facultad de Ciencias, Universidad Nacional de Colombia, segunda edition, 2003.

[Kor04] Rodrigo De Castro Korgi. *Teoría de la Computación. Lenguajes , Autómatas y Gramáticas*. Colección Notas de clase, Facultad de Ciencias, Universidad Nacional de Colombia, 2004.

[Lei98] Louis Leithold. *El Cálculo*. Oxford University Press, séptima edition, 1998.

[Lem96] Karen A. Lemone. *Fundamentos de compiladores: Cómo traducir al lenguaje de computadora*. Compañía Editorial Continental S.A.(CECSA), 1996.

[Man04] Ignacio Mantilla. *Análisis Numérico*. Colección Notas de clase, Facultad de Ciencias, Universidad Nacional de Colombia, 2004.

[Sch01] Herbert Schildt. *JAVA 2 Manual de referencia*. McGraw-Hill, cuarta edition, 2001.

[TF87] George B. Thomas and Ross L. Finney. *Cálculo con Geometría Analítica*, volume 1. Addison-Wesley Iberoamericana, sexta edition, 1987.

[VPH97] Laurence Vanhelsuvé, Ivan Phillips, and Goan-Tag Hsu. *La Biblia de Java*. Ediciones Amaya Multimedia, S.A, 1997.