

Visualización del mercado de acciones en 3D

Juan Esteban Pemberthy

Juan Sebastian Munoz

28 de abril de 2009

Índice general

Introducción	v
1. Objetivos	1
1.1. Objetivo General	1
1.2. Objetivos Específicos	1
2. Importancia del problema.	3
3. Preliminares	5
3.1. Conceptos de diseño de lenguajes de programación	5
3.1.1. Sintáxis concreta versus sintáxis abstracta	5
3.1.2. Gramáticas libres de contexto, Notación y ambigüedad. .	6
3.1.3. Operadores, precedencia y asociatividad	7
3.2. Lambda Calculo	7
3.2.1. Estrategias de Reducción	9
3.2.2. Sistema de Tipos	9
4. Parsing, Análisis y Evaluación.	11
4.1. Una visión general	11
4.2. Lexing y Parsing	11
4.3. Resolución de operadores	13
4.4. Sistema de tipos	13
4.5. Construcción de notación aritmética	13
4.6.	13
5. Conclusiones	15

Introducción

“Elegir acciones individuales sin una idea de lo que estás buscando es como atravesar una fábrica de dinamita con una cerilla encendida. Puedes sobrevivir, pero sigues siendo un idiota”

–Joel Greenblat¹

En la bolsa de valores de Colombia existen cuatro mercados principales de operación, el trabajo de ésta tesis se enfoca única y exclusivamente a uno de estos mercados; El mercado de Renta Variable, donde se negocian las acciones de compañías inscritas en el mercado publico de valores.

Al tratarse de un mercado público cualquier persona esta en la libertad de comprar y/o vender acciones según lo desee, generalmente estas personas llevan cierto control sobre sus inversiones, ya sea de un modo sistematizado ó escribiendo en una agenda de apuntes sus movimientos, con esta información realizan un análisis de datos que le permite determinar cuándo una acción le genera ganancia y/o perdida que apoyen una decisión, como comprar o vender.

El análisis de datos siempre ha estado ligado a una representación gráfica de los mismos, y el mercado de acciones mencionado anteriormente no es la excepción, generalmente para representar éstos datos, los que describen el estado y movimiento en el tiempo de una acción se han utilizado gráficos en segunda dimensión como barras, tortas, entre otras.

Esta tesis introduce un nuevo paradigma de visualización en este campo, al pasar de gráficos en dos dimensiones, a una nueva propuesta de contenidos para visualizar los datos en tres dimensiones, permitiéndole a una persona visualizar de una manera innovadora el estado y movimiento de las acciones sobre las que lleve algún tipo de registro.

Para lograr éste trabajo, se desarrollo una aplicación web, en la que se integró satisfactoriamente un modelo de datos con un conjunto de contenidos 3D, sobre los cuales se estableció un protocolo de comunicación que permite el apropiado despliegue de gráficos, buscando que sea más fácil para un usuario

¹ “The little book that beats the market” por Joel Greenblat.

identificar el movimiento de una ó varias acciones.

La propuesta acá descrita es presentada como proyecto de grado para optar por el título de Ingeniería de Sistemas de la Universidad EAFIT.

Capítulo 1

Objetivos

1.1. Objetivo General

Exploración del uso de operadores disfijos permisivos en un lenguaje fuertemente tipado.

1.2. Objetivos Específicos

- Entender que son los operadores disfijos.
- Implementar un lenguaje de programación prueba de concepto que simule operadores disfijos mediante el lenguaje de programación Haskell.
- Utilizar los operadores disfijos del lenguaje como mecanismo de extensibilidad para implementar un lenguaje con características objetuales. Dicha construcción se realizará en etapas.

Capítulo 2

Importancia del problema.

Muchos de los avances en el desarrollo de software han tenido su origen en el area de lenguajes de programación. Los desarrollos en lenguajes de programación han permitido:

1. Escribir software en forma portable gracias a la posibilidad de abstraer los detalles arquitectónicos de la plataforma para la cual se desarrolla el software.
2. Escribir software mas seguro, ocultando y restringiendo ciertas construcciones inseguras del lenguaje objeto.¹
3. Reutilización de código, mediante la incorporación de abstracciones para la generalización de soluciones.²

De la misma manera, desarrollos en lenguajes de programación han permitido el desarrollo de lenguajes mas expresivos. Entendiendo por lenguaje expresivo es aquel que brinda construcciones sintácticas y abstracciones que permiten al programador resolver problemas en forma mas “elegante”³.

La importancia de la elegancia en los lenguajes de programación además de tener ventajas en mantenibilidad, y legibilidad tambien parece ser pieza importante en la reducción de errores. Existen indicios [11] para afirmar que la densidad de errores por número de líneas no se ve alterada por la elección del lenguaje de programación. De esta forma, lenguajes que requieren menos líneas de código para resolver diferentes problemas (lenguajes expresivos) pueden ayudar a disminuir la cantidad de errores.

Brindar expresividad enfrenta al diseñador del lenguaje de programación al dilema de Cardelli [4]: Decidir entre dar una amplia y expresiva notación o

¹Entendiendo lenguaje objeto como el lenguaje producido por el lenguaje de programación. Por ejemplo, **x86** es uno de los lenguajes objeto de **C**

²Un ejemplo de ésto es Polimorfismo. El lenguaje diseñado en esta tesis tiene características polimórficas. Más sobre esto en la sección 4.

³Elegante puede tener connotaciones subjetivas, sin embargo el concepto es formalizado en [5]

tener un “core” pequeño. Ambas propiedades son deseables en un lenguaje de programación. Los “cores” pequeños son mas fáciles de mantener y permiten que el lenguaje sea asimilado más fácil por los programadores. Existe un enfoque híbrido basado en lenguajes extensibles. Estos lenguajes parten de un “core” pequeño, pero permiten que el usuario defina su propia sintáxis.

Esfuerzos para construir lenguajes que puedan “crecer” [10] se han realizado anteriormente. Dentro de las técnicas utilizadas en este enfoque híbrido cabe mencionar las siguientes: “*Syntax Macros*” [2], “*Extensible Syntax*” [4], “*Conctypes*” [1].

Este trabajo complementa dichos esfuerzos mostrando como un subconjunto de operadores disfijos puede ser utilizado como mecanismo de extensibilidad en los lenguajes de programación.

Capítulo 3

Preliminares

3.1. Conceptos de diseño de lenguajes de programación

Esta sección presenta conceptos generales sobre construcción de lenguajes de programación. Dichas definiciones han sido basadas en [8]

3.1.1. Sintáxis concreta versus sintáxis abstracta

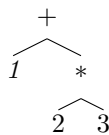
Definición 1. *Sintaxis concreta en lenguajes de programación se refiere a la representación del programa como cadenas de caracteres. Dicha representación es la que sirve de interfaz para el programador. La sintáxis concreta es el primer nivel en la definición de la sintaxis de un lenguaje.*

Ejemplo 1. *El siguiente es un ejemplo de sintaxis concreta para una expresión matemática.*

$$1 + 2 * 3$$

Definición 2. *Sintáxis Abstracta es el segundo nivel en la representación de sintáctica de un programa. Este nivel es alcanzado mediante dos procesos. Análisis léxico y parsing. El análisis léxico descompone la cadena de caracteres en “tokens” o lexemas como identificadores, literales y puntuación. El segundo paso transforma la lista de tokens en un árbol de sintaxis abstracta. Es en este paso es donde se resuelve la presedencia de operadores.*

Ejemplo 2. *El siguiente arbol representa la sintáxis abstracta del ejemplo presentado en 1*



Como se mencionó anteriormente la precedencia es resuelta en el paso anterior a la construcción del árbol sintáctico abstracto. De esta forma, la representación abstracta omite los parentesis.

3.1.2. Gramáticas libres de contexto, Notación y ambigüedad.

Una gramática libre de contexto es un formalismo desarrollado por Noam Chomsky para describir en forma recursiva la estructura de bloques de los lenguajes. Dicho formalismo es ampliamente utilizado para la descripción formal de los lenguajes de programación. Una de las razones es que dada la gramática libre de contexto, o un subconjunto de esta, es posible construir algoritmos que reconozcan el lenguaje expresado por dicha gramática en forma programática. Dos algoritmos ampliamente usados para este propósito son: *LL* y *LR*.

Formalmente, una gramática libre de contexto consta de un conjunto de símbolos terminales (V). Un conjunto de símbolos no terminales Σ , un conjunto de reglas generadoras $V \rightarrow (V \cup \Sigma)^*$. Y un $S \in V$ llamado símbolo inicial. Usualmente se utiliza convención Bakus-Naur o una de sus derivadas para definir la gramática. En este trabajo usaremos la siguiente convención:

Ejemplo 3. *Simple gramática para la definición de números.*

$$\begin{aligned}\langle number \rangle &\rightarrow \langle digit \rangle | \langle digit \rangle \langle number \rangle \\ \langle digit \rangle &\rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9\end{aligned}$$

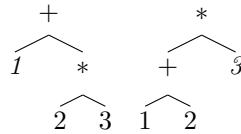
1. Símbolo inicial es el primero en la lista de “producciones”
2. Símbolos no terminales son aquellos en negrilla.
3. Símbolos no terminales están encerrados en $\langle \rangle$

Definición 3. *Una gramática libre de contexto es ambigua si existe una cadena perteneciente al lenguaje generado por dicha gramática y para esta, existe mas de un árbol sintáctico.*

Ejemplo 4. *El siguiente es un ejemplo de expresiones aritméticas ambiguas. (Por brevedad usaremos la misma definición de $\langle digit \rangle$ anteriormente mencionada)*

$$\begin{aligned}\langle E \rangle &\rightarrow \langle E \rangle + \langle E \rangle \\ \langle E \rangle &\rightarrow \langle E \rangle * \langle E \rangle \\ \langle E \rangle &\rightarrow \langle digit \rangle\end{aligned}$$

Dada la anterior gramática y la expresión $1+2*3$ es posible construir dos árboles sintácticos.



Esto es suficiente para concluir que la gramática es ambigua.

3.1.3. Operadores, precedencia y asociatividad

El ejemplo 4 muestra dos árboles sintácticos para la misma expresión aritmética. Solo uno de estos árboles cumple con el orden de evaluación esperado. El orden de evaluación esperado no es mas que el concepto de precedencia.

3.2. Lambda Calculo

Lambda cálculo es un formalismo matemático detras de la definición y aplicación de funciones. Dicho formalismo además representa la noción de computabilidad en forma equivalente a la maquina de Turing [3].

Lambda expresiones

El elemento constitutivo del lambda cálculo son “lambda expresiones”.

Definición 4. Una lambda expresión puede ser definida en forma recursiva de la siguiente manera:

1. Si v es una variable, v es una lambda expresión.
2. Si v es una variable y M es una lambda expresión, $\lambda v.M$ es una lambda expresión. A esto operación nos referiremos como abstracción de M sobre v .
3. Si N y M son lambda expresiones entonces $N M$ también es una expresión. Esto es llamado aplicación (M se aplica a N)

Variables Libres

La operación de abstracción liga¹ la variable asociada. Son variables libres aquellas que no están ligadas por ninguna abstracción.

Ejemplo 5. En $\lambda x.x$ y la variable x en la expresión está ligada y la variable y está libre.

¹El termino usual en la literatura es “bind”.

α -Conversión

Dos expresiones son alfa-equivalentes (denotado por $=_\alpha$) si es posible transformar una en otra, renombrando las variables.

Ejemplo 6. $\lambda x.x y =_\alpha \lambda z.z y$

Renombrar las variables ligadas de una lambda expresión es conocido como α -Conversión. Existen dos casos especiales para tener en cuenta al realizar α -Conversión. En primer lugar, no es posible realizar α -Conversión de una expresión por una variable que se encuentre libre en dicha expresión. Esto es llamado captura de nombres².

Ejemplo 7. $\lambda x.x y \neq_\alpha \lambda y.y y$

El segundo caso se refiere al ámbito de las abstracciones. $\lambda x.\lambda x y$

Notación

Por practicidad seguiremos ciertas convenciones notacionales descritas en [9].

1. Los paréntesis más externos serán omitidos. De esa forma $(N M)$ será escrito: $N M$
2. La abstracción se extiende hasta donde sea posible. $\lambda x.N M$ es equivalente a $\lambda x.(N M)$ y no a $(\lambda x.N)M$
3. Aplicación es asociativa por izquierda. $M N O$ es equivalente a $(M N) O$ y no a $M(N O)$
4. Podemos abreviar la expresión $\lambda x.\lambda y.M$ de la forma: $\lambda x y.M$

Lambda cálculo puro versus lambda cálculo aplicado.

Existen diferentes codificaciones bien conocidas usando lambda expresiones para diferentes tipos de datos usuales en lenguajes de programación³. Dichas codificaciones resultan convenientes como mecanismo de exploración del lambda cálculo como formalismo de computabilidad; sin embargo, ineficiente a la hora de implementar lenguajes de programación. Debido a esto suelen añadirse construcciones al lambda cálculo "puro," explicado anteriormente. Dichos lambda cálculos extendidos suelen denominarse lambda cálculos aplicados. A continuación se ilustra la relación de ambos conceptos⁴

$$\begin{aligned} \text{programming language} &= \text{applied lambda calculus} \\ &= \text{pure lambda system} + \text{basic data types} \end{aligned}$$

Ahora agregaremos 2 nuevas reglas a la definición 4 de lambda expresión. Siguiendo la terminología usada lo resultante es un lambda cálculo aplicado.

²En la literatura se refieren a "name capture"

³El capítulo 3 de [9] incluye dichas codificaciones para enteros, tuplas, listas arboles

⁴Tomado de [7] Página 370.

1. Si k es un entero o k es una cadena de caracteres entonces k es una constante.
2. Si k es una constante entonces k es una lambda expresión.

3.2.1. Estrategias de Reducción

3.2.2. Sistema de Tipos

Capítulo 4

Parsing, Análisis y Evaluación.

Esta sección explica los diferentes pasos en la implementación de nuestro lenguaje para soportar operadores permisivos.

4.1. Una visión general

Inicialmente

$$Texto \rightarrow_1 Tokens \rightarrow_2 (Declaraciones, Main)$$

4.2. Lexing y Parsing

El primer paso de parsing detecta las declaraciones de funciones y la función principal main. Esta es la gramática del lenguaje siguiendo la notación del capítulo 3.

$\langle Program \rangle \rightarrow \langle Declarations \rangle \langle Main \rangle$
 $\langle Declarations \rangle \rightarrow \langle Declarations \rangle \langle Declaration \rangle | \epsilon$
 $\langle Declaration \rangle \rightarrow \langle Infix \rangle | \langle Prefix \rangle | \langle Suffix \rangle | \langle Closed \rangle$
 $\langle Infix \rangle \rightarrow \langle Init \rangle \langle InfixKeyword \rangle \langle Precedence \rangle \langle Id \rangle \langle Id \rangle \langle Id \rangle = \langle Definition \rangle$
 $\langle Precedence \rangle \rightarrow \epsilon \langle Number \rangle$
 $\langle InfixKeyword \rangle \rightarrow \mathbf{infixr} | \mathbf{infixl}$
 $\langle Init \rangle \rightarrow \mathbf{let} | \mathbf{let\ rec}$
 $\langle Prefix \rangle \rightarrow \langle Init \rangle \langle Id \rangle \langle Ids \rangle = \langle Definition \rangle$
 $\langle Suffix \rangle \rightarrow \langle Init \rangle \mathbf{suffix} \langle Ids \rangle \langle Id \rangle = \langle Definition \rangle$
 $\langle Closed \rangle \rightarrow \langle Init \rangle \mathbf{closedId} \langle Ids \rangle \langle Id \rangle = \langle Definition \rangle$

$\langle \text{Ids} \rangle \rightarrow \langle \text{Id} \rangle \langle \text{Ids} \rangle | \epsilon$
 $\langle \text{Definition} \rangle \rightarrow \langle \text{ExpresionTokens} \rangle$
 $\langle \text{ExprTokens} \rangle \rightarrow \langle \text{ExprToken} \rangle \langle \text{ExprTokens} \rangle | \epsilon$
 $\langle \text{ExprToken} \rangle \rightarrow \langle \text{Literal} \rangle | \langle \text{Id} \rangle$
 $\langle \text{Literal} \rangle \rightarrow \langle \text{Number} \rangle | \langle \text{String} \rangle$
 $\langle \text{Number} \rangle \rightarrow \langle \text{Digit} \rangle | \langle \text{Digit} \rangle \langle \text{Number} \rangle$
 $\langle \text{Digit} \rangle \rightarrow \mathbf{0} | \mathbf{1} | \mathbf{2} | \mathbf{3} | \mathbf{4} | \mathbf{5} | \mathbf{6} | \mathbf{7} | \mathbf{8} | \mathbf{9}$
 $\langle \text{String} \rangle \rightarrow \mathbf{''} \langle \text{Chars} \rangle \mathbf{''}$
 $\langle \text{Chars} \rangle \rightarrow \langle \text{Char} \rangle \langle \text{Chars} \rangle | \epsilon$
 $\langle \text{Char} \rangle \rightarrow \text{Cualquier caracter}$
 $\langle \text{Id} \rangle \rightarrow \langle \text{Char} \rangle | \langle \text{Char} \rangle \langle \text{Id} \rangle$
 $\langle \text{Main} \rangle \rightarrow \mathbf{main} = | \langle \text{Definition} \rangle$

Ahora bien. Para definir una expresión añadiremos notación adicional: Op_p^a Indica que el operador tiene precedencia p asociatividad a . Usaremos un entero para denotar p y $\{l, r\}$ son los posibles valores de a . l denota asociatividad por izquierda y r asociatividad por derecha.

$$Expr := Open Expr Closed$$

$$Expr := Expr_1 Infix_n^r Expr_2 | \forall_{Op_m^r \in Expr_1 \text{ or } Expr_2} n \leq m$$

En español... Una expresión puede estar formada por un operador infijo operando 2 subexpresiones. siempre y cuando el operador tenga la menor precedencia. En la defición falta agregar el manejo de asociatividad (en el caso de $Infix_n^r$ se debe garantizar que no exista ningún operador con la misma precedencia y misma asociatividad en $Expr_2$).

$$Expr := Expr_1 Infix_n^l Expr_2 | \forall_{Op_m^l \in Expr_1 \text{ or } Expr_2} n \leq m$$

$$Expr := PrefixOp Expr$$

$$Expr := Expr SuffixOp$$

$$Expr := Literal$$

$$Literal := string | natural$$

4.3. Resolución de operadores

Data: Lista de ExprTokens
Result: Arbol de evaluación
 initialization;
if *understand* **then**
 | go to next section;
end

Algoritmo 1: Resolución de precedencia

4.4. Sistema de tipos

El lenguaje de programación es fuertemente tipado y está construido a partir de la implementación encontrada en [6]. El sistema de tipos esta conformado por los siguientes constructores en Haskell.

```
data Type    = TVar String
              | TInt
              | TBool
              | TString
              | TList Type
              | TProd Type Type
              | TFun Type Type
```

El sistema de tipos cuenta con variables de tipos, enteros, booleanos, cadenas de caracteres, listas genéricas, funciones y producto de tipos.

4.5. Construcción de notación aritmética

4.6.

Capítulo 5

Conclusiones

Bibliografía

- [1] Anika Aasa. *User Defined Syntax*. PhD thesis, Chalmers University of Technology, 1992.
- [2] Arthur Baars and Doaitse Swierstra. Syntax macros. Unfinished Draft.
- [3] Henk Barendregt and Erik Barendsen. Introduction to lambda calculus. Disponible en línea: <http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/geuvers.pdf>, March 2000.
- [4] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. Technical report, Systems Research Center, 1994.
- [5] G. J. Chaitin. *People and Ideas in Theoretical Computer Science*, chapter Elegant LISP Programs. Springer-Verlag, 1999. Disponible en línea: <http://www.cs.auckland.ac.nz/CDMTCS//chaitin/lisp.html>.
- [6] Martin Grabmüller. Algorithm w step by step. Draft paper, September 2007.
- [7] Jan Leeuwen and Jan van Leeuwen, editors. *Handbook of Theoretical Computer Science: Formal models and semantics*. MIT Press, 1994.
- [8] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [9] Peter Selinger. Lecture notes on the lambda calculus. Disponible en línea <http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf>. Notas de un curso de lambda cálculo de la universidad de Ottawa de 2001.
- [10] Guy L. Steele, Jr. Growing a language. *Higher Order Symbol. Comput.*, 12(3):221–236, 1999.
- [11] Ulf Wiger. Four-fold increase in productivity and quality. Technical report, Ericsson, 2001.