Simulando Operadores Disfijos

Diego Echeverri Saldarriaga

May 3, 2010

Resumen

La extensibilidad en lenguajes de programación se refiere a la posibilidad de tener un lenguaje donde el programador puede añadir sus propias construcciones sintácticas. Dicha propiedad permite trasladar muchas construcciones del lenguaje de programación a librerías. Éste trabajo muestra algunos problemas en la implementación de un lenguaje con características objetuales a partir de un lenguaje funcional, perezoso que utiliza un subconjunto de operadores disfijos como mecanismo de extensibilidad.

Abstract

Extensibility in programming language design is about the possibility of letting the programmer add it's own syntactic constructions. This property allows to transfer some built-in functions to libraries. This thesis shows some problems implementing a programming language with object oriented features starting from a lazy, functional language that uses a subset of disfix operators as an extensibility mechanism.

Agradecimientos

Este trabajo de tesis no hubiese sido posible de no ser por los profesores Juan Francisco Cardona, Francisco Correa y Andrés Sicard. Fueron ellos los que despertaron mi curiosidad por el mundo de la programación declarativa, programación funcional y la teoría detrás del diseño e implementación de lenguajes de programación. El grupo de lógica y la lista de correo de dicho grupo me brindaron recursos y bibliografía para este trabajo e interesantes discusiones alrededor del tema.

Todos aquellos que me ayudaron en la travesía de programar en el paradigma funcional. Mis compañeros de trabajo (Joel Björnson, Ersoy Bayramoglu, Adam Granicz y Anton Tayanovskyy) fueron una invaluable fuente de soporte en el entendimiento de muchos temas que de otro modo hubieran permanecido esotéricos. De la misma manera, agradezco a todos aquellos que se tomaron el trabajo de leer los borradores iniciales y comentar al respecto.

Por último y no menos importante, mi familia. Fueron ellos los que me dieron soporte incondicional para encontrar mi vocación.

Índice general

I 6

1 Introducción 7

2 Marco Teórico 9
- 2.1 Conceptos de diseño de lenguajes de programación 9
  - 2.1.1 Sintaxis concreta versus sintaxis abstracta 9
  - 2.1.2 Gramáticas independientes del contexto, notación y ambigüedad. 10
  - 2.1.3 Operadores, precedencia y asociatividad 11
  - 2.1.4 Precedencia y ambigüedad 11
- 2.2 Lambda Cálculo 12
  - 2.2.1 Expresiones lambda 12
  - 2.2.2 Notación 13
  - 2.2.3 Variables Libres 13
  - 2.2.4 α-Conversión 13
  - 2.2.5 β-Reducción 14
  - 2.2.6 Lambda cálculo puro versus lambda cálculo aplicado. . . 14
- 2.3 Operadores disfijos 15

3 Descripción y Definición del Problema 16
- 3.1 Motivación 16
- 3.2 Definición 17

II 19

4 Parsing, Análisis y Evaluación. 20
- 4.1 Una visión general 20
- 4.2 Descripción del lenguaje utilizando BNF 21
  - 4.2.1 Resolución de operadores 22
- 4.3 Estrategia de reducción 26
- 4.4 Sistema de tipos 27
  - 4.4.1 Codificación de tipos 28
  - 4.4.2 Recursión 28

4.5 Simulación de operadores disfijos 29

5 Jugando con los operadores disfijos. 30
5.1 Notación aritmética 30
5.1.1 Operadores básicos 30
5.1.2 Agrupación de expresiones 31
5.2 Listas 31
5.2.1 Secuencias 32
5.3 Control de flujo 32
5.3.1 Operador ternario 32
5.3.2 Switch-Case-Default 33
5.4 Conversión entre tipos y notación húngara 34
5.5 “Literales” XML 35
5.6 Objetos 35

III 36

6 Trabajo Relacionado 37

7 Conclusiones 39
7.1 Conclusiones 39
7.2 Trabajo Futuro 40
7.2.1 Adiciones al lenguaje 40
7.2.2 Mejoras 41

A Compilación y ejecución de Nano. 43
A.1 Requisitos 43
A.2 Compilación y ejecución. 43
A.3 Ejemplos 44
A.3.1 Tipos de datos 44
A.3.2 Operadores 45
A.3.3 Recursión 46

Índice de cuadros

2.1 Posibles resultados de parsing de la expresión $1+2*3$ ante diferentes condiciones de precedencia y asociatividad 11

3.1 Ejemplos del subconjunto de operadores validos e invalidos en Nano. 17

4.1 Estructura general por capas del lenguaje. 20

Índice de figuras

2.1 Ejemplo de sintaxis abstracta 10
2.2 Ejemplo de ambigüedad. 11
2.3 Ambigüedad por multiple definición 12
2.4 Ambigüedad de un operador contenido en otro. 12

4.1 Resolución de operadores cerrados. 24
4.2 Resolución de operadores. 26
4.3 Ambigüedad de operadores. 26
4.4 Operador disfijo infijo simulado. 29

5.1 Reescritura paso a paso de la lista usando las definiciones de los operadores. 32

Parte I

Capítulo 1

Introducción

"A good Notation has a sublety and suggestiveness which at times makes it almost seem like a live teacher."
–Bertrand Russell[1]

Un operador disfijo es un operador de varias partes, en el cual, los argumentos pueden ser contenidos por las diferentes partes [Jon86] [Aas92]. Un ejemplo tomado del lenguaje de programación OBJ es el siguiente:

```
op if_then_else_fi : Bool Int Int -> Int .
```

Éste operador se comporta en forma similar a los bloques condicionales de otros lenguajes de programación, sin embargo posee una vital ventaja. Su definición está dada usando el mismo lenguaje de programación en oposición a estar definida como una construcción primitiva dentro del compilador. Ésta diferencia es de vital importancia para el programador, ya que le permite construir su propio lenguaje a través de un lenguaje básico.

La idea de construir un lenguaje de programación con soporte de operadores disfijos no es nueva. Lenguajes como Coq[dC07], OBJ [GWM$^+$96], y Maude [CDE$^+$99] ya proveen soporte para éstos. Éste trabajo en cambio, explora la idea de utilizar un conjunto reducido de operadores "primitivos" para soportar operadores disfijos más complejos. Para dicha exploración, se ha diseñado Nano[2]. Nano es un lenguaje prueba de concepto de características funcionales y de evaluación perezosa.

El capítulo 2 hace un recorrido sobre diferentes conceptos necesarios tratados en este trabajo, lo cual incluye algunas observaciones sobre la notación usada. Éste capítulo posee 3 secciones. 2.1 hace algunas distinciones importantes sobre diseño de lenguajes de programación. 2.2 introduce nociones básicas sobre lambda cálculo y 2.3 da algunas definiciones de operadores disfijos. Personas familiarizadas con lambda cálculo pueden obviar la segunda sección.

[1] Prólogo del capítulo *"Minilanguages: Finding a Notation that Sings"* en *"The art of Unix Programming"* por Eric Raymond [Ray03].

[2] Nano es el prefijo de SI para 10^{-9}. La idea es denotar la pequeñez del lenguaje

En el capítulo 3 comienza con una sección donde se explican las motivaciones de la construcción del lenguaje de programación de éste trabajo. La siguiente sección define el problema que intenta solucionar el lenguaje de programación.

En el capítulo 4 se describe en términos generales el diseño de Nano. Se inicia con una descripción de álto nivel de los diferentes pasos de traducción del lenguaje. Luego se menciona como se realiza el análisis sintáctico (resolución de operadores ambiguos, precedencia y operadores cerrados). Al final se explica el sistema de tipos y la estrategia de reducción utilizada.

El capítulo siguiente (5) muestra la expresividad que se puede lograr con Nano. Éste capítulo muestra como implementar, a nivel de librerías, muchas características que son agregadas *built-in* en otros lenguajes. Entre otras cosas se muestran operadores para expresar aritmética, listas y XML.

La última parte hace una síntesis del trabajo relacionado. El último capítulo establece las conclusiones y el posible trabajo futuro.

Capítulo 2

Marco Teórico

2.1 Conceptos de diseño de lenguajes de programación

Ésta sección presenta conceptos generales sobre construcción de lenguajes de programación. Dichas definiciones han sido basadas en [Pie02]

2.1.1 Sintaxis concreta versus sintaxis 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 sintaxis concreta es el primer nivel en la definición de la sintaxis de un lenguaje.*[1]

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

$$1 + 2 * 3$$

Definición 2. *Sintaxis Abstracta es el segundo nivel en la representación de un programa. Éste nivel es alcanzado mediante dos procesos: Análisis léxico y analisis sintáctico. 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 éste paso donde se resuelve la precedencia de operadores.*[2]

Ejemplo 2. *El siguiente árbol representa la sintaxis 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 ésta forma, la representación abstracta omite los paréntesis.*

[1] Páginas 53-54 de [Pie02]

[2] Íbid.

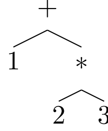


Figura 2.1: Ejemplo de sintaxis abstracta

2.1.2 Gramáticas independientes del 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 ésta, es posible construir algoritmos que reconozcan el lenguaje expresado por dicha gramática en forma programática. Dos algoritmos ampliamente usados para éste propósito son: $LL(K)$ y $LR(K)$.

Formalmente, una gramática libre de contexto es una cuádrupla formada por 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 la convención Bakus-Naur o una de sus extensiones para definir la gramática. En éste trabajo usaremos la siguiente convención:

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

$\langle number \rangle \rightarrow \langle digit \rangle | \langle digit \rangle \langle number \rangle$

$\langle digit \rangle \rightarrow$ **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

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 independiente del contexto es ambigua si existe una cadena perteneciente al lenguaje generado por dicha gramática y para ésta, existe más de un árbol sintáctico.*[3]

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

[3] Página 7 de [Aas92]

$\langle E\rangle \rightarrow \langle E\rangle + \langle E\rangle$

$\langle E\rangle \rightarrow \langle E\rangle * \langle E\rangle$

$\langle E\rangle \rightarrow \langle digit\rangle$

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.*

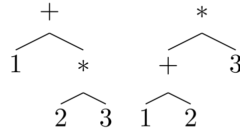


Figura 2.2: Ejemplo de ambigüedad.

2.1.3 Operadores, precedencia y asociatividad

La figura 2.2 del ejemplo 4 muestra dos árboles sintácticos para la misma expresión aritmética. Para resolver dicha ambigüedad utilizamos niveles de precedencia y asociatividad. El nivel de precedencia especifica que tan "fuerte" el operador liga los argumentos. La asociatividad en cambio dice como agrupar las operaciones de operadores con el mismo nivel de precedencia. Un operador puede ser asociativo por izquierda, derecha o no asociativo. La tabla 16 es tomada de [DN08] e ilustra las diferentes resoluciones basadas en asociatividad y precedencia.

Precedencia	Asociatividad	Resultado
$+ > *$		$(1 + 2) * 3$
$+ < *$		$1 + (2 * 3)$
$+ = *$	Ambos asociativos por derecha	$1 + (2 * 3)$
$+ = *$	Ambos asociativos por izquierda	$(1 + 2) * 3$
Otros:		Error de *"Parsing"*

Cuadro 2.1: Posibles resultados de parsing de la expresión 1+2∗3 ante diferentes condiciones de precedencia y asociatividad

2.1.4 Precedencia y ambigüedad

La utilización de precedencia como mecanismo de resolución de ambigüedad resulta insuficiente en algunas expresiones. Éste hecho es ilustrado en [Aas92] con los siguientes ejemplos:

Ejemplo 5. *Dado el lenguaje donde el operador* $\$$ *es prefijo, posfijo e infijo. La expresión* $1\$\$\$2$ *tiene las siguientes interpretaciones:*

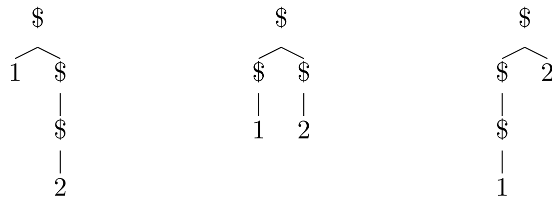


Figura 2.3: Ambigüedad por multiple definición

Ejemplo 6. *Dado el lenguaje con los operadores infijos* $\$\$\$$ *y* $\$\$$ *y el operador sufijo* $\$$*. La expresión* $1\$\$\$2$ *tiene las siguientes interpretaciones:*

Figura 2.4: Ambigüedad de un operador contenido en otro.

2.2 Lambda Cálculo

Es un formalismo matemático que describe como se definen y aplican funciones. Éste formalismo representa la noción de computabilidad en forma equivalente a la máquina de Turing [BB00]. En éste trabajo sólo utilizamos éste cálculo como base semántica del lenguaje diseñado [4].

2.2.1 Expresiones lambda

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

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

1. *Si* v *es una variable,* v *es una expresión lambda.*

[4] Sección 1.4 y 1.5 de [Sel]

2. *La operación $\lambda v.M$ donde v es una variable y M es una expresión es conocida como* ***Abstracción*** *de M sobre v. Toda abstracción es también una expresión lambda.*

3. *La operación (NM) donde N y M son expresiones lambda es conocida como* ***Aplicación****. Toda aplicación es a su vez una expresión lambda.*

2.2.2 Notación

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

1. Variables son representadas con minúsculas.

2. Expresiones lambda arbitrarias son representadas con mayúsculas.

3. Los paréntesis más externos serán omitidos. De esa forma $(N\,M)$ será escrito: $N\,M$

4. La abstracción se extiende hasta donde sea posible. $\lambda x.N\,M$ es equivalente a $\lambda x.(N\,M)$ y no a $(\lambda x.N)M$

5. La aplicación es asociativa por izquierda. $M\,N\,O$ es equivalente a $(M\,N)\,O$ y no a $M(N\,O)$

6. Se abreviará la expresión $\lambda x.\lambda y.M$ de la forma: $\lambda x\,y.M$

2.2.3 Variables Libres

La operación de abstracción liga[5] la variable asociada. Son variables libres aquellas que no están ligadas por ninguna abstracción.

Ejemplo 7. *En $\lambda x.x\,y$ la variable x en la expresión está ligada y la variable y está libre.*

2.2.4 α-Conversión

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

Ejemplo 8. $\lambda x.x\,y =_\alpha \lambda z.z\,y$

Dicho renombrado de variables en una expresión lambda es conocido como α-Conversión. Existen dos casos especiales para tener en cuenta al realizar una α-Conversión. En primer lugar, no es posible realizar una α-Conversión de una expresión por una variable que se encuentre libre en dicha expresión. Esto es llamado captura de nombres[6].

[5] El termino usual en la literatura es *"bind"*.
[6] En la literatura se refieren a *"name capture"*

Ejemplo 9. $\lambda x.x\,y \neq_\alpha \lambda y.y\,y$

El segundo caso se refiere al ámbito de las abstracciones. El ligado de una variable está dado por la abstracción más interna.

Ejemplo 10. $\lambda x.\lambda x.x \neq_\alpha \lambda y.\lambda x.y$

2.2.5 β-Reducción

β-Reducción es el proceso análogo a la aplicación de funciones. Éste está definido por medio de substitución.

Definición 5. *El resultado de aplicar una lambda abstracción a un argumento es una instancia del cuerpo de la lambda abstracción, en la cual, todas las ocurrencias libres del parámetro de la expresión lambda han sido sustituidas por el argumento.*[7]

Ejemplo 11.

$$(\lambda x.x\,y)E \to_\beta E\,y$$
$$(\lambda x.\,3)E \to_\beta 3$$

Una expresión puede ser reducida usando diferentes "caminos". Cada reducción es llamada *redex*. La siguiente es una expresión con dos posibles *redex*

Ejemplo 12.

$$(\lambda y.y((\lambda x.x)M))N \to_\beta N((\lambda x.x)M)$$
$$(\lambda y.y((\lambda x.x)M))N \to_\beta (\lambda y.yM)N$$

2.2.6 Lambda cálculo puro versus lambda cálculo aplicado.

Es posible codificar los tipos de datos básicos[8] (tuplas, listas, enteros, árboles, etc) usando expresiones lambda. Dichas codificaciones resultan convenientes como mecanismo de exploración del lambda cálculo como formalismo de computabilidad; sin embargo, ineficientes a la hora de implementar lenguajes de programación. Es por esto que suelen añadirse algunos tipos básicos al lambda cálculo "puro" explicado anteriormente. Dichos lambda cálculos extendidos suelen denominarse lambda cálculos aplicados. A continuación se ilustra la relación de ambos conceptos [9]

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

[7] Sección 2.2.2 de [Jon87]
[8] El capítulo 3 de [Sel] incluye dichas codificaciones
[9] Tomado de [LvL94] Página 370.

Si se agregan 2 nuevas reglas a la definición 4 de lambda expresión, el resultado es un lambda cálculo aplicado.

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 expresión lambda.

2.3 Operadores disfijos

Operadores disfijos, también conocidos como *"distfix"* y *"mixfix"* en inglés, son operadores de una o más partes que pueden traer los argumentos embebidos en si mismos. La notación estándar utilizada para describir operadores disfijos se basa en la utilización del carácter "_" para localizar los argumentos.

Ejemplo 13. *Operador condicional.*

$$if \ _ then \ _ else \ _$$

Los operadores disfijos pueden verse como una generalización de los operadores usuales en lenguajes de programación.

1. Operador Prefijo: $Op_$

2. Operador Sufijo: $_Op$

3. Operador Infijo: $_Op_$

Adicional a ésto, [Aas92] generaliza dicho concepto y habla de:

1. Disfijos posfijos: Existe un argumento a la izquierda, pero no existe ningún operador a la derecha. $_A_B$

2. Disfijos prefijo: Existe un argumento a la derecha, pero no existe ningún operador a la izquierda. $A_B_$

3. Disfijos infijo: Existen argumentos a la derecha e izquierda. $_A_B_$

4. Disfijos cerrado: No existen argumentos a la derecha o izquierda. A_B

Capítulo 3

Descripción y Definición del Problema

3.1 Motivación

Muchos de los avances en el desarrollo de software han tenido su origen en el área de lenguajes de programación. Los desarrollos en lenguajes de programación han permitido[1]:

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 más seguro, ocultando y restringiendo ciertas construcciones inseguras del lenguaje objeto.[2]
3. Reutilización de código, mediante la incorporación de abstracciones para la generalización de soluciones. [3]

De la misma manera, desarrollos en lenguajes de programación han permitido el desarrollo de lenguajes más expresivos. Se entenderá por lenguaje expresivo como aquel que brinda construcciones sintácticas y abstracciones que permiten al programador resolver problemas en forma más "elegante" [4]. Dicha expresividad parece disminuir el número de defectos en el software [Wig01].

[1] En el capítulo "Origen e historia de Unix" de [Ray03] se menciona que según Dennis Ritchie:

"It seems certain that much of the success of Unix follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages"

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

[3] Un ejemplo de ésto es polimorfismo. El lenguaje diseñado en esta tésis tiene características polimórficas.

[4] Elegante puede tener connotaciones subjetivas, sin embargo el concepto es formalizado en [Cha99]

3.2 Definición

Brindar expresividad enfrenta al diseñador del lenguaje de programación al dilema de Cardelli [CMA94]: Decidir entre dar una amplia y expresiva notación o tener un *"core"* pequeño. Ambas propiedades son deseables en un lenguaje de programación. Los *"cores"* pequeños son más 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" [Ste99] se han realizado anteriormente. Dentro de las técnicas utilizadas en este enfoque híbrido cabe mencionar las siguientes: *"Syntax Macros"* [BS], *"Extensible Syntax"* [CMA94], *"Conctypes"* [Aas92] y operadores disfijos [Jon86].

De todas estos enfoques: ¿Cuál es el más minimal? Este trabajo le apuesta a un conjunto reducido de operadores. Éstos pueden ser combinados para generar operadores disfijos más complejos. El subconjunto de operadores escogidos para este trabajo consta de operadores prefijos, posfijos, infijos y cerrados según se explicó en 2.3. Además se agregan las siguientes limitaciones:

1. Todo operador infijo está limitado a 2 argumentos.
2. Los operadores no cerrados cuentan con una única parte.
3. Los operadores cerrados cuentan con exactamente 2 partes.

Tipo de Operador	Validos	Invalido por 1	Invalido por 2 ó 3
Prefijo	`++2`, `+ 2 3`		`! 3 : 2`
Posfijo	`2--`, `2 3 -`		`3 : 2 !`
Infijo	`1+2`,	`3 <&> 2 2`	`true ? 3 : 4`
Cerrado	`(3)`, `(* 3 2 4 *)`		`{1 * 2}`,

Cuadro 3.1: Ejemplos del subconjunto de operadores validos e invalidos en Nano.

Otra característica de Nano es la eliminación de la distinción entre operadores e identificadores. Por ejemplo el operador `+suma+` puede ser definido en Nano, pero resulta inválido en muchos lenguajes que soportan definición de operadores. Así mismo, un operador que solo utilice caracteres alfanuméricos (por ejemplo `foo`) se comportará de la misma forma que el resto de operadores. [5].

Teniendo en cuenta las características de Nano, ésta tesis intenta responder las siguientes preguntas:

- ¿Es posible construir éste lenguaje?
- ¿Es Nano un lenguaje "extensible"?
- ¿Qué ventajas en expresividad nos brinda Nano?

[5]Ésta característica trae algunas consecuencias que serán exploradas en el capítulo 4 y utilizadas en 5

- ¿Es Nano suficientemente extensible como para servir de base para construir un lenguaje con características objetuales?

Parte II

Capítulo 4

Parsing, Análisis y Evaluación.

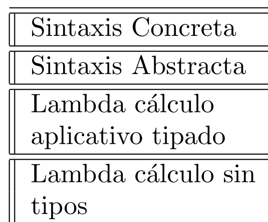
Esta capítulo explica los diferentes pasos en la implementación de Nano.

4.1 Una visión general

El cuadro 4.1 muestra en términos generales las capas sobre las que está implementado el lenguaje. En el nivel inferior, se tiene, el cálculo lambda sin tipos. Es en este nivel donde se implementa la estrategia de reducción del lenguaje. Sobre este, se implementa el cálculo lambda tipado que permite definir tipos sobre las expresiones de nivel inferior. La construcción de listas, pares y booleanos usan la codificación estándar pero tienen tipos en el lambda cálculo aplicativo de segundo nivel.

Finalmente hay sintaxis concreta sobre el lambda cálculo aplicativo. Ésta sintaxis es resuelta por medio de los siguientes pasos:

1. Construcción de Tokens: En este paso se detectan palabras reservadas, identificadores y literales. El mecanismo utiliza exclusivamente Parsec [LM01]. Específicamente el modulo `ParsecToken` es usado para éste propósito.



Sintaxis Concreta
Sintaxis Abstracta
Lambda cálculo aplicativo tipado
Lambda cálculo sin tipos

Cuadro 4.1: Estructura general por capas del lenguaje.

2. Construcción de Declaraciones: En este paso se definen operadores y sus definiciones. Dichas definiciones son reconocidas como listas de Tokens de Expresión (⟨*ExprTokens*⟩). Los ⟨*ExprTokens*⟩ no son más que un subconjunto de los lexemas del lenguaje (operadores y literales).
3. Construcción de los Árboles de Expresión: Para cada lista de tokens que conforma una expresión se construye un árbol sintáctico resolviendo los operadores ambiguos y precedencia (Ver: 4.2.1).

4.2 Descripción del lenguaje utilizando BNF

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 2.

⟨Program⟩→⟨Declarations⟩⟨Main⟩

⟨Declarations⟩→⟨Declarations⟩⟨Declaration⟩|ϵ

⟨Declaration⟩→⟨Infix⟩|⟨Prefix⟩|⟨Suffix⟩|⟨Closed⟩

⟨Infix⟩→⟨Init⟩⟨InfixKeyword⟩⟨Precedence⟩⟨Id⟩⟨Id⟩⟨Id⟩=⟨Definition⟩

⟨Precedence⟩→ϵ⟨Number⟩

⟨InfixKeyword⟩→**infixr**|**infixl**

⟨Init⟩→**let**|**let rec**

⟨Prefix⟩→⟨Init⟩⟨Id⟩⟨Ids⟩=⟨Definition⟩

⟨Suffix⟩→⟨Init⟩**suffix**⟨Ids⟩⟨Id⟩=⟨Definition⟩

⟨Closed⟩→⟨Init⟩**closed**⟨Id⟩⟨Ids⟩⟨Id⟩=⟨Definition⟩

⟨Ids⟩→⟨Id⟩⟨Ids⟩|ϵ

⟨Definition⟩→⟨ExpresionTokens⟩

⟨ExprTokens⟩→⟨ExprToken⟩⟨ExprTokens⟩|ϵ

⟨ExprToken⟩→⟨Literal⟩|⟨Id⟩

⟨Literal⟩→⟨Number⟩|⟨String⟩

⟨Number⟩→⟨Digit⟩|⟨Digit⟩⟨Number⟩

⟨Digit⟩→**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

⟨String⟩→**"**⟨Chars⟩**"**

⟨Chars⟩→⟨Char⟩⟨Chars⟩|ϵ

⟨Char⟩→Cualquier carácter

⟨Id⟩→⟨Char⟩|⟨Char⟩⟨Id⟩

⟨Main⟩→**main =** |⟨Definition⟩

Basados en [DN08] realizamos el *"parsing"* de las definiciones inicialmente como listas de $\langle ExprToken \rangle$. Hasta este punto se utilizan mecanismos convencionales de *"parsing"*. Específicamente se usa Parsec [LM01] para análisis léxico y *"parsing"*.

4.2.1 Resolución de operadores

Dadas las expresiones definidas como una lista de $\langle ExprToken \rangle$ es necesario construir un árbol que tenga en cuenta la precedencia de éstos. Existe un procedimiento bien conocido ([ASU86]) para convertir una gramática con precedencias en una gramática libre de contexto. Él método se basa en añadir símbolos no terminales a la gramática para resolver la ambigüedad.

Existen otros algoritmos para este propósito. Entre estos se encuentra el *"Shunting-yard algorithm"* de Dijkstra. En [Nor99] se puede encontrar una recopilación de algoritmos. El lenguaje diseñado cuenta con un sistema más expresivo de operadores lo cual impide el uso de los anteriores métodos "clásicos".

1. Operadores Prefijos: Éstos tienen la forma $Op_$ con la adición de soportar n[1] argumentos después del operador.
2. Operadores Sufijos: Tienen la forma $_Op$ con soporte para n argumentos antes del operador.
3. Operadores Cerrados: Tienen la forma $Op1_Op2$ con soporte para n argumentos en el interior de la primera y segunda parte del operador.
4. Operadores Infijos: Tienen la forma $_Op_$. A diferencia de los anteriores, estas funciones siempre tienen aridad 2.[2]

Resolución de operadores ambiguos.

Los operadores, a diferencia de los identificadores, pueden ser usados sin requerir espacio en blanco. Ésto genera un problema de ambigüedad.

Ejemplo 14. *La cadena* `"+++"` *puede ser interpretada como:* `["+++"]` `["+","++"]` `["++" "+"]` `["+", "+", "+"]`.

En la sección 2.1.4 se muestran algunas expresiones donde la precedencia de los operadores no sirve para resolver la ambigüedad. El primer caso menciona el problema de tener operadores homónimos. En Nano la unicidad de operadores es una condición (al igual que en [Aas92]) y por tanto dicho problema no ocurre.

El segundo caso se refiere a operadores compuestos de sub-partes de otros operadores. [Aas92] menciona que esto puede ser resuelto en *lexing*[3] asumiendo el token mas largo. En Nano se toma la decisión de lidiar con este problema de

[1] El número de argumentos soportados es constante. Funciones polivariádicas introducen mayor ambigüedad. Ésto trae consecuencias para la introducción de polimorfismo Ad-Hoc (Ver: 7.2.1)

[2] La utilidad de operadores infijos de aridad mayor de 2 permanece elusiva para el autor.

[3] *Lexing* es el proceso de análisis léxico. És en esta etapa donde se transforma la cadena de caracteres en una secuencia de "Lexemas" o *Tokens*. Dicho proceso se explica en [ASU86]

ambigüedad. La estrategia es basada en una especie de *lexing* no determinista. A continuación se muestra una primera aproximación. El siguiente algoritmo genera todas las listas de sub-cadenas que concatenadas son iguales al operador a resolver.

1. Si el operador está conformado de un solo carácter c entonces, todas las subcadenas son $\{["c"]\}$
2. Si el operador está conformado de más de un carácter $\{c : resto\}$ donde c es el primer carácter y $resto$ es el resto de la cadena, entonces todas las subcadenas pueden ser computadas como $C(c, subcadenas(resto)) \cup A(c, subcadenas(resto))$, donde $C(x, y)$ es una función que concatena x a todas las primeras cadenas de los elementos de y. En cambio $A(x, y)$ agrega a x como nueva primera cadena a cada uno de los elementos de y

La función de número de cadenas(f), esta dada por:

1. $f(1) = 1$ (caso trivial).
2. $f(n) = 2 * f(n-1)$ (dado por: $|C| = |A| = |subcadenas(resto)|$ y $(C \cap A) = \{\}$). Ésto equivale a $f(n) = 2^{n-1}$.

A partir de ésta estrategia es posible eliminar las listas de subcadenas que contienen “Tokens” inválidos u operadores que no han sido declarados. Ésta estrategia tiene dos inconvenientes:

- La estrategia no reduce el orden de posibles subcadenas en el peor caso.
- La estrategia no elimina la ambigüedad.

El peor caso solo puede darse cuando el ambiente está “contaminado” con todas las posibles partes del operador. Se sostiene que la legibilidad del código es tal vez un problema más grave que el parsing del mismo. La sección 7.2.2 menciona algunas alternativas para mejorar el algoritmo actual. El manejo de ambigüedad es discutido en la sección 4.2.1.

Resolución de operadores cerrados.

En el trabajo de [DN08] asocian la mayor precedencia a la resolución de operadores cerrados. Dado el operador `Op1_Op2` nos referiremos a `Op1` como la parte “abierta” del operador y `Op2` como la parte “cerrada”.

1. Buscar la parte abierta más interna Op_1. Si antes de encontrar la primera parte abierta se encuentra una parte cerrada, se lanza un error. Si no se encuentra una parte abierta se finaliza.
2. Buscar la primera parte cerrada Op_2. Si la parte cerrada, no cierra Op_1, lanzar error.
3. Resolver la expresión demarcada por la partes encontradas en Op_1 y Op_2.[4]

[4]Dado que se trata de la expresión más interna, se garantiza que no habrá operadores cerrados sin resolver. La resolución de expresiones libres de operadores cerrados se explica en la siguiente sección.

4. Reemplazar en la cadena la subcadena demarcada por Op_1 y Op_2 por la aplicación del operador cerrado a la expresión resuelta.
5. Llamar esta función en forma recursiva sobre la cadena generada en 4.

Ejemplo 15. *Resolución de la expresión* $\{(2) + 3\}$.

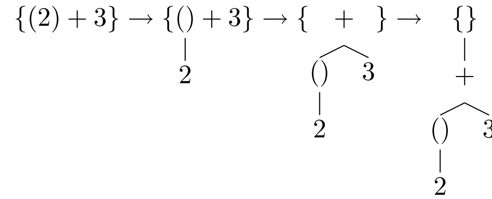


Figura 4.1: Resolución de operadores cerrados.

Construcción de los árboles de expresión.

En primer lugar, definimos la siguiente relación de orden parcial entre cada una de los $\langle ExprToken \rangle$. Dicha relación es denotada por $<_p$ y es definida de la siguiente forma:

1. Si x es un operador y tiene aridad 0[5] ó x es un literal, entonces $x <_p y$ para todo y diferente de operadores de aridad 0 y literales.
2. Si x y y son operadores, entonces $x <_p y$ si x tiene menor precedencia que y.
3. Si x y y son operadores, entonces $x <_p y$ si x y y son asociativos por derecha.
4. Si x y y son operadores, entonces $x >_p y$ si x y y son asociativos por izquierda.

Dada esta relación y la lista de $\langle ExprToken \rangle$, podemos definir el algoritmo de resolución de precedencia de la siguiente forma:

1. Si todos elementos a resolver son: literales, operadores de aridad 0 ó subexpresiones ya evaluadas, retornar la lista.
2. Buscar un operador x tal que no exista un y que cumpla $x >_p y$.
3. Si x es infijo, entonces llamar ésta función usando la parte derecha de x y parte izquierda de x. Construir un árbol con cada una de las subexpresiones resultantes donde la raíz sea el operador x. Si alguno de los lados está vacío, lanzar un error.

[5]Nano no limita la aridad de los operadores no infijos. Ésto implica que es posible definir operadores con aridad cero. Dichos operadores hacen las veces de variables. Por ejemplo: `let five = 5`

4. Si x es prefijo, retornar los elementos del lado izquierdo concatenados con un árbol con raíz x y como hijos, el resultado de aplicar esta función en forma recursiva al lado derecho de x. Si el número de argumentos provistos por la evaluación del lado derecho no corresponde a la aridad de x, lanzar un error.
5. Si x es sufijo, resolver en forma análoga al caso donde x es prefijo.

Ejemplo 16. *Dada la expresión* $2!+2+3*4*\sqrt{9}$. *Con las siguientes precedencias:*

Operador	Precedencia	Asociatividad
$\sqrt{}$	2	
!	2	
+	3	derecha
$*$	4	izquierda

Los pasos de resolución de los operadores están dados por la figura 4.2.

Detección y Eliminación de Ambigüedad

En la sección 5.1.1 se muestra como existe la posibilidad de que un operador pueda ser interpretado de diferentes formas debido al *lexing* no determinista que usa Nano. Pese a que se hace un filtro inicial donde se eliminan los árboles que usan símbolos que no han sido definidos, esto no garantiza la no ambigüedad. Tómese por ejemplo el siguiente caso:

Ejemplo 17. *Se tienen dos operadores:* `s` *y* `ss`. *Ambos son operadores prefijos y tienen tipos Bool $\rightarrow$ String y a $\rightarrow$ a respectivamente. Es posible construir dos árboles sintácticos (Figura: 4.3).*

En el caso de Nano, el no determinismo, está expresado mediante computaciones sobre listas. Si después de la resolución de operadores existen uno (o más) árboles sintácticos en ésta lista, Nano procede a realizar chequeo de tipos sobre cada uno de los árboles sintácticos[6]. Dicho chequeo se realiza en forma perezosa sobre toda la lista. Después de ésto existen 3 escenarios:

- **La lista de posibles árboles sintácticos está vacía:** Nano imprime información para ayudar al programador a corregir dicha expresión.
- **La lista de posibles árboles sintácticos tiene un único elemento:** Éste árbol es utilizado en la declaración o evaluación de la expresión.
- **La lista de posibles árboles sintácticos tiene uno o más elementos:** Se evalúan los primeros dos árboles y se notifica al programador de las dos (primeras) posibles interpretaciones.

En el caso del ejemplo 17, sólo el segundo árbol posee tipos que unifican. Por esto, Nano considerará la expresión válida.

[6]El chequeo de tipos de Nano se explica en la sección 4.4

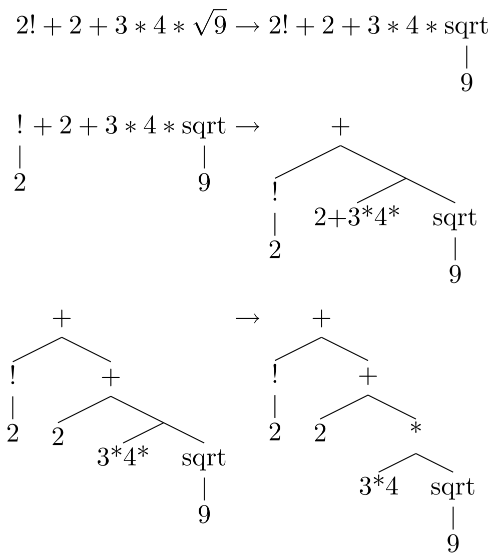


Figura 4.2: Resolución de operadores.

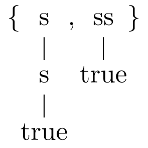


Figura 4.3: Ambigüedad de operadores.

4.3 Estrategia de reducción

En los lenguajes de programación convencionales, la evaluación de funciones utiliza orden aplicativo, es decir, los argumentos son evaluados y luego substi-

tuidos en la expresión[7]. Dicho orden tiene consecuencias en la construcción de funciones que requieren la no evaluación de los argumentos.

```
let foo x y = if (true) x y
```

El comportamiento de $(foo\ 3\ \bot)$[8] en un lenguaje de orden aplicativo implicaría la no terminación del programa. Dado que la idea es construir azúcar sintáctica alrededor de las primitivas del lenguaje, un orden de evaluación aplicativo resulta bastante restrictivo.[9] Por esta razón se opta por diseñar el lenguaje con una estrategia de evaluación perezosa.[10]

La estrategia de reducción utilizada corresponde a *"Weak Head Normal Form"* (WHNF). Ésta estrategia posee además la ventaja de evitar el problema de captura de nombres descrito en la sección 2.2.4. La definición de dicha estrategia de reducción esta explicada en [Jon87][11] de la siguiente manera:

Definición 6. *Una lambda expresión de la forma* $F\,E_1\,E_2 \cdots E_n$ *con* $n > 0$ *está en WHNF sí y solo sí* F *es una variable o constante, o* $F\,E_1\,E_2 \cdots E_m$ *no es una redex*[12] *para ningún* $m \leq n$

4.4 Sistema de tipos

El lenguaje de programación es fuertemente tipado y está construido a partir de la implementación encontrada en [Gra07]. El sistema de tipos de este lenguaje es expresado por la siguiente gramática.

$\langle \text{Tipo} \rangle \rightarrow$ "Variable"

$\langle \text{Tipo} \rangle \rightarrow$ "Entero"

$\langle \text{Tipo} \rangle \rightarrow$ "Booleano"

$\langle \text{Tipo} \rangle \rightarrow$ "String"

$\langle \text{Tipo} \rangle \rightarrow$ "Lista" $\langle \text{Tipo} \rangle$

$\langle \text{Tipo} \rangle \rightarrow$ "Producto" $\langle \text{Tipo} \rangle \langle \text{Tipo} \rangle$

$\langle \text{Tipo} \rangle \rightarrow$ "Funcion" $\langle \text{Tipo} \rangle \langle \text{Tipo} \rangle$

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

---

[7] Capítulo 1 de [AS96]

[8] $\bot$ Es el símbolo de no terminación o fallo. Una implementación de $\bot$ en Haskell podría ser: `foo = foo`

[9] Un ejemplo de la vida real es la implementación del operador `_?_:_` al estilo de **C**.

[10] Se consideraron alternativas como evaluación estricta con anotaciones de evaluación perezosa en los argumentos.

[11] Sección 11.3.1

[12] En éste caso, *redex* (Sección 2.2.5) considera también las operaciones primitivas del sistema. Las operaciones aritméticas requieren contar con todos sus argumentos para ser reducidas lo cual fuerza evaluación estricta (Sección 11.4 de [Jon87]).

4.4.1 Codificación de tipos

A excepción de los enteros y *strings*, los demás datos utilizan representaciones puras en el lambda cálculo sin tipos que es usado por el intérprete. Dichas representaciones son "nombradas" y tipadas en el lambda cálculo aplicativo. A continuación se muestran las definiciones en el lambda cálculo sin tipos y los tipos asignados.

Booleanos:

$$False : Bool = \lambda x\, y.y;$$
$$True : Bool = \lambda x\, y.x$$
$$Not : Bool \rightarrow Bool = \lambda x\, y\, z.x\, z\, y$$
$$If : Bool \rightarrow a \rightarrow a = \lambda x\, y\, z.x\, y\, z$$

Pares:

$$BuildPar : a \rightarrow b \rightarrow (a \cdot b) = \lambda x\, y\, p.p\, x\, y$$
$$First : (a \cdot b) \rightarrow a = \lambda p.p\, True$$
$$Second : (a \cdot b) \rightarrow b = \lambda p.p\, False$$

Listas:

Para la construcción de listas se utilizan pares. La idea básica es definir una lista como un par que contiene en la primera posición el primer elemento de la lista y en segundo, una lista con el resto de la lista (cola). Además se define la lista vacía.

$$Cons : a \rightarrow [a] \rightarrow [a] = (head, rest)$$
$$Empty : [a] = \lambda x.True$$
$$IsNull : [a] \rightarrow Bool = \lambda l.(\lambda x\, y.False)$$

4.4.2 Recursión

Es posible[13] definir funciones recursivas utilizando el combinador Y. La gramática de la sección 4.2 muestra como cada *"let"* puede ser definido con una palabra clave adicional *"rec"*. Éstos *"let"* recursivos son traducidos de la siguiente manera[14].

$$let\, rec\, v = E \rightarrow let\, v = Y(\lambda v.E)$$

[13] La sección 2.4.1 de [Jon87]

[14] Íbid sección 3.2.2

4.5 Simulación de operadores disfijos

Dado el lenguaje de programación diseñado, es posible "simular" el concepto general de operadores disfijos. Ésta aproximación se basa en el encadenamiento de operadores soportados y construcción de valores intermedios. A continuación se ilustra la regla de construcción de dichos operadores:

1. Si el operador es un operador disfijo infijo de la forma $_Op_1_Op_2 \cdots Op_n_$ entonces dicho operador puede ser simulado reemplazando cada parte $Op_i, 1 \leq i \leq n$ con operadores infijos. Donde, $Op_x <_p Op_y, x < y$.
2. Si el operador es un operador disfijo prefijo de la forma $Op_0_Op_1 \cdots Op_n_$ este puede ser simulado como un operador prefijo Op_0 aplicado al un operador disfijo infijo.
3. Si el operador es un operador disfijo sufijo de la forma $_Op_1 \cdots Op_n_Op_{n+1}$, este puede ser simulado como un operador sufijo Op_{n+1} aplicado al un operador disfijo infijo.
4. Si el operador, es un operador disfijo cerrado, este puede ser simulado aplicando un operador cerrado $Op_0_Op_{n+1}$ a un operador disfijo infijo.

La estructura general de un operador disfijo infijo simulado a partir de operadores infijos.

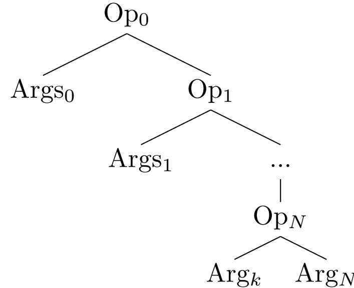


Figura 4.4: Operador disfijo infijo simulado.

Teniendo este orden esperado de evaluación, se utilizan los operadores Op_i con i mayor que 0 para colectar los argumentos en una estructura de datos conveniente. El operador Op_0 implementa la lógica del operador a simular. El mecanismo de colección de argumentos está limitado a listas y tuplas. Desventajas y trabajo futuro de usar dichas estructuras se discute en 7.2.1.

Como nota adicional, resulta importante garantizar que los operadores utilizados en el interior de los argumentos tengan mayor precedencia que los operadores utilizados para simular el operador disfijo. De otro modo, la construcción del árbol sintáctico fallará si se omiten paréntesis.

Capítulo 5

Jugando con los operadores disfijos.

Los operadores diseñados pueden ser utilizado como mecanismo de extensibilidad del lenguaje. Muchas de las construcciones sintáticas *"built-in"* en los lenguajes convencionales pueden ser derivadas de dichos operadores. Ésta sección comienza a crecer un pequeño lenguaje usando el mecanísmo de simulación de operadores disfijos.

5.1 Notación aritmética

5.1.1 Operadores básicos

Es posible construir los operadores básicos de suma y multiplicación mediante el siguiente código:

```
let infixr 1 x + y = add x y
let infixr 2 x * y = times x y
let infixr 3 x == y = equals x y
```

Asignando una mayor precedencia a la multiplicación que a la suma se obtiene el comportamiento usual en otros lenguajes de programación. Las expresiones `1 + 2 * 3` y `3 * 2 + 1` serán evaluadas correctamente a 7.

Nano no provee de literales para números negativos, sin embargo éstos pueden ser fácilmente implementados con la ayuda de un operador prefijo.[1]

```
let 10 - x = sub  x
```

Con esta adición es posible escribir `-2 + 2` y `2 + -2`. En la sección se describe la condición de unicidad de cada operador en Nano. Dicha limitación impide

[1]Se utiliza una alta precedencia (10) para evitar que dicho operador "capture" expresiones mas grandes que la que está inmediatamente a la derecha.

utilizar el operador "-" para la resta binaria sin eliminar el operador unario de negación.

5.1.2 Agrupación de expresiones

El lector cuidadoso habrá notado que Nano no posee sintaxis para la agrupación de expresiones. Los lenguajes de programación convencionales proveen paréntesis con el fin de hacer explícito el orden de aplicación. Nano, en cambio, permite definir mecanismos de agrupación explotando 2 de sus características: polimorfismo y operadores cerrados.

```
let closed ( x ) = x
```

El anterior operador tiene como tipo $a \rightarrow a$ y corresponde a la implementación de la función identidad. En este caso, se aprovecha que los operadores cerrados ligan a sus argumentos con la mayor precedencia (Sección 4.2.1). La expresión `3 * (2 + 1)` será evaluada, naturalmente, en el valor 9.

5.2 Listas

La estructura de datos por excelencia de los lenguajes funcionales es la lista. Debido a esto es usual que los lenguajes de programación brinden "azúcar sintáctica" para escribir dichas estructuras. A continuación se muestra como es posible obtener una sintaxis similar a la de Haskell.

Mientras que en Haskell "[]" es un caso sintáctico especial. En Nano, "[]" es un operador como cualquier otro[2].

```
let [] = empty
```

Ahora es posible definir el operador `:` para poder construir listas.

```
let infixr 2 x : y = cons x y
```

Hasta este momento es posible escribir `1:2:3:[]`. Sin embargo, esto no es suficiente. Haskell también provee la conveniente sintaxis `[1,2,3]`. La versatilidad de Nano permite definir listas de esa forma mediante las siguientes declaraciones.

```
let 10 [ x = x
let infixr x , y = x : y
let suffix x ] = x : empty
```

El primer operador es la simple función de identidad. Su utilidad no es sinó estética[3]. El segundo operador es un alias para cons. La tercera declaración concatena la lista vacía al elemento mas a la derecha. La figura 5.2 muestra la equivalencia entre ambas representaciones.

[2]En Nano, `empty` es una primitiva para la lista vacía de tipo `[a]`
[3]La sección 7.2.1 ilustra un posible uso futuro para dicho operador.

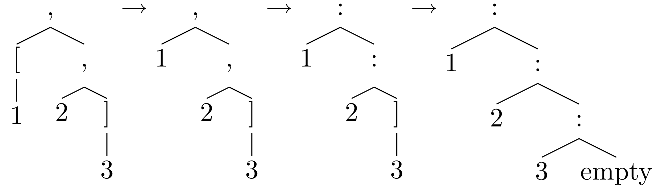


Figura 5.1: Reescritura paso a paso de la lista usando las definiciones de los operadores.

5.2.1 Secuencias

Muchos lenguajes proveen sintaxis para describir secuencias. Mediante recursión es posible definir dicho operador:

```
let rec infixr x ... y = if (x > y)
                             empty
                             (cons x ((x+1)...y))
```

5.3 Control de flujo

Los lenguajes convencionales proveen sintaxis para la construcción de ciclos (`for`, `foreach`, `while`, `do-while`) y decisiones (`if-then`, `if-then-else`, `_?_:_`). Nano, por ser un lenguaje funcional, utiliza recursión en vez de ciclos. Para proveer computaciones condicionales Nano provee la función `if`. Ésta función de tipo: $Bool \rightarrow a \rightarrow a$ es suficiente para proveer otros mecanismos de control de flujo[4] mas complejos.

5.3.1 Operador ternario

Lenguajes como **C** proveen del operador ternario `_?_:_`. Si el primer argumento es verdadero, el operador retorna el segundo argumento. Sino, retorna el tercer argumento. Es fácil simular este operador mediante las siguientes declaraciones:

```
let infixl 4 x ? y = if x (cons y empty) (empty)
let infixl 4 x : y = if (isNull x) y (hd x)
```

Él operador `?` se encarga de construir una lista vacía, o una lista con el segundo argumento[5]. Luego, el operador `:` dependiendo de si es una lista vacía o una lista de un elemento retorna el valor apropiado.

[4]Él uso del término "Control de flujo" es tal vez inapropiado en este contexto. El control de flujo trabaja sobre el orden de ejecución de sentencias. Nano solo posee expresiones y declaraciones. Se apela al término debido a que se construyen expresiones que son similares **sintácticamente** a sus contra-partes imperativas.

[5]Se simula el tipo "Maybe" de Haskell por medio de una lista

Resulta importante notar que éste truco funciona porque la evaluación de Nano es perezosa (ver sección 4.3). La construcción de la lista que hace el operador : no requiere la evaluación del segundo argumento. Para probar esto basta realizar la siguiente prueba:

```
Nano> let rec bottom = bottom
Type: a0
Nano> true ? 1 : bottom
Type: Int
Val: Const (IData 1)
Nano> false ? bottom : 1
Type: Int
Val: Const (IData 1)
```

La estrategia de reducción es la que permite que ambas expresiones terminen.

5.3.2 Switch-Case-Default

Si se quisiera implementar una función simple que asignara a los valores de $1-3$ su sucesor y en otro caso el valor cero. Se podría utilizar el operador ternario de la siguiente manera:

```
let v x = (equals x 1) ? 2 :
            ((equals x 2) ? 3 :
               ((equals x 3) ? 4 :
                 0))
```

El problema de dicha aproximación es que rápidamente el numero de paréntesis anidados destruyen la legibilidad del código. El `switch-case-default` es un mecanísmo de computación condicional que proveen algunos lenguajes. En Nano se puede simular mediante las siguientes declaraciones:

```
let 10 switch x = buildPair empty x
Type: a0 -> ([a5] * a0)

let infixl x case y = if (isNull (fst x))
                         (if (equals y (snd x))
                             (buildPair true x)
                             (buildPair false x))
                         (buildPair false x)
Type: ([a9] * Int) -> Int -> (Bool * ([a9] * Int))

let infixl x -> y = if (fst x)
                       (buildPair (cons y empty) (snd (snd x)))
                       (snd x)
Type: (Bool * ([a22] * a27)) -> a22 -> ([a22] * a27)
```

```
let infixl x default y = if (isNull (fst x)) y (hd (fst x))
Type: ([a2] * a14) -> a2 -> a2
```

Se construyen tuplas intermedias para acumular valores importantes. En el caso de `switch` se guarda el valor con el que se realizan las comparaciones. El operador `case` agrega una bandera que indica si el valor de este `case` será utilizado. El operador `->` utiliza la bandera para decidir si debe reemplazar el valor actual. Al final, el operador `default` devuelve el valor acumulado o retorna el valor por defecto.

Éste mecanismo nos permite escribir expresiones como la siguiente:

```
Nano> switch 0 \
      case 0 -> "cero"\
      case 1 -> "uno"\
      default   "otros"
Type: String
Val: Const (SData "cero")
```

5.4 Conversión entre tipos y notación húngara

Una inspiración para la construcción de Nano fué el idioma húngaro. En éste idioma las palabras cambian su función de acuerdo a los sufijos y prefijos utilizados. Esta técnica de sufijos y prefijos fue usada por Charles Simonyi para definir la "notación húngara" [Sim99]. Uno de los usos de dicha notación es anotar los tipos de las variables. Por ejemplo, se podrían utilizar el prefijo "i" para enteros y "s" para cadenas de caracteres.

Ejemplo 18.

```
let sThree = "3"
let iThree = 3
```

Dado que Nano trata todos los identificadores como operadores. No es necesario dejar espacio en blanco en la aplicación de los mismos. Esto permite construir prefijos para hacer conversión de tipos. Por ejemplo, es posible definir los siguientes prefijos para conversión de tipos:

```
let s x = if x "true" "false"
let i x = if x 1 0
```

De esta manera es posible escribir `strue`, `sfalse`, `itrue` y `ifalse`[6]

[6] La ausencia de polimorfismo *Ad-hoc* nos impide generalizar esta conversión a otros tipos. La posibilidad de agregar *type classes* se discute en 7.2.1

5.5 “Literales” XML

XML es un lenguaje ampliamente utilizado para el intercambio de información. Dada su amplia utilización algunos lenguajes ([MSD10]) tienen soporte sintáctico para dichos literales. En Nano es posible reproducir, hasta cierto punto, la funcionalidad de éstos literales mediante la utilización de operadores cerrados. Por ejemplo, es posible implementar un pequeño DSL para escribir HTML.

```
let infixr x ; y = buildPair x y
let closed <html> x </html> = buildPair "html" x
let closed <head> x </head> = buildPair "body" x
let closed <body> x </body> = buildPair "body" x
let closed <div> x </div> = buildPair "div" x
```

Ésto permite escribir cosas como:

```
<html>
    <head>""</head>;
    <body>
        <div>
          "div #1"
        </div>;
        <div>
          "div #2"
        </div>
    </body>
</html>
```

5.6 Objetos

Pese a la gran flexibilidad que permite Nano, la construcción de un lenguaje con características objetuales permanece elusivo. Inicialmente se pretendía construír un lenguaje con orientación a objetos basada en prototipos ([UCCH91]).

La sintáxis podría ser creada usando la misma estrategia que se utilizó en la sección 5.3.2 donde el parsing de cada objeto sería un diccionario de métodos y atributos. Dicha aproximación falla en dos puntos clave.

Él primero es que en Nano, las funciones no son realmente “ciudadanas de primera clase”. Ésto se debe a que el algoritmo de resolución de precedencia requiere que las funciones tengan todos sus argumentos “aplicados”. Ésto impide que las funciones se pueden asignar y mucho menos guardar en tablas. Igualmente, a pesar de que el sustrato de Nano es el lambda-calculo, Nano no provee mecanísmos sintácticos para escribir abstracciones.

En segundo lugar, construir tablas para almacenar métodos en un lenguaje con chequeo estático de tipos resulta difícil. Especialmente cuando el sistema de tipos es tan poco expresivo como el de Nano [7].

[7] La sección 7.2.1 hace un pequeño comentario sobre una posible solución a éste problema

Parte III

Capítulo 6

Trabajo Relacionado

Uno de los primeros lenguajes que exhibe dicha característica es Hope [BMS80]. El concepto es resucitado por Simon Peyton Jones en [Jon86]. En éste artículo, el autor muestra como es posible agregar operadores disfijos a casi cualquier lenguaje de programación mediante una pequeña extensión. Dicha aproximación utiliza mecanismos comunes para generación de parsers (específicamente **Yacc**). Uno de los problemas que no son abordados es la resolución de precedencia de dichos operadores. La idea del autor es dar a estos operadores la precedencia más baja, de tal modo, que estos operadores no produzcan ningún conflicto con los demás elementos constitutivos del lenguaje.

El problema sobre precedencia de operadores es tratado en la tesis doctoral [Aas92] y el artículo [Aas91] de dicha tesis. En ese trabajo, se construyen gramáticas que tienen en cuenta la precedencia para la construcción de los árboles sintácticos. La precedencia es especificada mediante niveles de precedencia, representados por números naturales. El autor provee un algoritmo (Algoritmo M) que traduce dichas gramáticas de precedencia en gramáticas libres de contexto que incorporan las reglas de precedencia. Éste algoritmo es utilizado por el autor para implementar un lenguaje experimental con soporte de operadores disfijos.

El trabajo es además importante ya que muestra algunas limitaciones de la utilización de precedencia para la resolución de ambigüedad. Ésto se menciona en 2.1.4.

El primero se refiere a la unicidad de los operadores y el segundo se refiere a la construcción de operadores usando partes de otros operadores. El autor asume la primera como condición de su trabajo. La segunda es resuelta, asumiendo la ocurrencia más larga en el paso de *"Lexing"*. Bajo éstas premisas El autor además demuestra unicidad de los árboles con precedencia correcta y correctitud del Algoritmo M.

Otro trabajo más reciente sobre incorporación de operadores disfijos en lenguajes de programación se puede encontrar en [DN08]. En éste, los autores construyen una gramática parametrizada por una relación de precedencia. A diferencia de [Aas92] ésta relación no está limitada a niveles de precedencia,

sino que acepta un grafo dirigido y acíclico. Una idea interesante de éste trabajo es la resolución de los operadores disfijos cerrados con la mayor precedencia posible. La meta de dicho trabajo es reemplazar el parsing de operadores disfijos de Agda([wika]).

Actualmente, el soporte de operadores disfijos se da principalmente en asistentes de demostraciones como [dC07] y [wika]. Existen algunos otros lenguajes que poseen capacidad de definir dichos operadores, tales como [GWM+96] y [CDE+99].

Capítulo 7

Conclusiones

7.1 Conclusiones

Este trabajo muestra como la no distinción entre identificadores y operadores, junto con la adición de operadores cerrados son una poderosa herramienta notacional en los lenguajes de programación. Éste mecanismo es capaz por si solo de simular muchas características de operadores disfijos. Dicha simulación de operadores disfijos resulta útil como mecanismo de extensibilidad en lenguajes de programación.

La adición de operadores cerrados simplifica el diseño de lenguajes de programación en al menos un aspecto. Los mecanismos para agrupar expresiones, pasan de ser construcciones sintácticas del lenguaje (Sección 5.1.2), a ser elementos que pueden ser implementados desde el mismo lenguaje de programación. Ésto permite mayor generalidad y flexibilidad notacional.

Nano muestra como es posible construir sintaxis para definir el flujo del programa mediante nada más que recursión y ejecución condicional (Sección 5.3). Para esto, una estrategia de reducción perezosa resulta fundamental (Sección 5.3.1). Es esta, la que permite la construcción de valores intermedios garantizando un buen comportamiento respecto a la terminación de los programas.

La elección de una gramática ambigua para la descripción del lenguaje no es, necesariamente, una mala decisión en el diseño de lenguajes de programación. Es posible utilizar los procesos posteriores al *parsing* para eliminar la ambigüedad. Dentro de éstos procesos Nano prueba que el chequeo de ámbito de variables y unificación de tipos son métodos eficaces para eliminar algunos casos de ambigüedad.

7.2 Trabajo Futuro

7.2.1 Adiciones al lenguaje

Tipos algebraicos y *"Pattern Matching"*

El lenguaje de programación diseñado carece por ejemplo de *"Pattern Matching"*. Una de las razones es la ausencia de tipos aditivos o *"Variants"*[1] Pese a ser ésta una característica ampliamente utilizada en el diseño de lenguajes de programación funcionales, existen ciertos problemas para hacer de estos "ciudadanos de primera clase" en el lenguaje de programación. Una propuesta interesante puede encontrarse en [Tul00]. El siguiente es un ejemplo de la utilización de combinadores como reemplazo de *"Pattern Matching"* del artículo mencionado[2]:

```
zipTree :: (Tree a, Tree b) => Maybe (Tree (a, b))
zipTree = Leaf# .* Leaf# .-> Leaf
       .| Tree# .* Tree# .: (zipTree.*zipTree .-> Tree) . zipTuple
```

La sintaxis concreta de esta aproximación podría verse beneficiada si se contara operadores permisivos.

Adicionalmente, el proceso de construcción de valores intermedios para la simulación de operadores disfijos bien podría beneficiarse de tener tipos aditivos, o en general, de tipos algebraicos. Ésto evitaría colisiones entre los tipos usados por el programador y los tipos previstos para la construcción de operadores disfijos. Ésto resulta útil para mejorar mensajes de error de algunas construcciones sintácticas.

Por ejemplo, la construcción de listas de la sección 5.2. Permite escribir cosas como `[[[[3`. La razón es que el operador `[` no es sinó la función de identidad. Si nano tuviese tipos algebraicos, sería posible cambiar la definición de este operador para que construyera un tipo temporal. En pseudo-nano algo como:

```
data InitList a = InitList a
let suffix x ]  = InitList (cons x empty)
let infixr x , (InitList y) = InitList (cons x y)
let [ (InitList x) = x
```

De ésta manera el operador `[` pasa de ser un operador estético a un mecanismo que garantiza consistencia sintáctica.

Polimorfismo Ad-Hoc

Polimorfismo *Ad-hoc* es el comportamiento de algunas funciones que dan la apariencia de trabajar sobre un conjunto de tipos, que bien pueden no exhibir la misma estructura y que el comportamiento para cada tipo puede variar independiente del tipo [CW85]. El mecanismo de polimorfismo Ad-Hoc de Haskell

[1] Formalización de dichos tipos puede encontrarse en secciones 11.9 y 11.10 de [Pie02]

[2] Segundo ejemplo de la sección 3.0

son las llamadas *"type classes"*. Mediante ellas es posible la construcción de funciones que pueden recibir diferentes números de parámetros (Funciones polivariádicas [Wikb]). Dado que nuestro algoritmo utiliza información sobre la aridad de las funciones, no está claro si dichas características son compatibles.

Otros cambios al sistema de tipos

Sistemas de tipos mas poderosos podrían servir para construir operadores mas estrictos. Una extensión que puede servir para esto es "Phantom Types" [CH03]. Este mecanismo puede ser utilizado para la construccion de estructuras de datos indexadas por tipo. Dicho mecanismo tal vez sería de utilidad en la construcción de objetos por medio de tablas.

Ámbito, *Lets* anidados y Abstracciones.

Pese a ser un lenguaje funcional, ésta primera versión no cuenta con la posibilidad de realizar declaraciones anidadas o generar abstracciones en forma *"inline"*. En los lenguajes funcionales actuales el ligado de una variable tiene una sintaxis concreta definida (`\var -> E` en Haskell, `fun var -> E` en dialectos de ML) La posibilidad de brindar primitivas de ligado al usuario que puedan ser combinadas con los operadores diseñados en este trabajo abre otras posibilidades interesantes en la reducción de construcciones inherentes en el lenguaje.

7.2.2 Mejoras

Reporte de errores

El reporte de errores al usuario se ve complicado por la naturaleza no determinista del mecanismo para resolver operadores.

Ejemplo 19. *Se tiene definido el operador de pre-incremento y de adición de la siguiente forma:*

```
let infixr x + y = add x y
let ++ x = x + 1
```

El usuario puede escribir la siguiente expresión incorrecta.

```
6 ++ 7
```

¿Hay un operador de más o de menos? La generación de errores comprensibles para el programador es algo importante que este trabajo no abordó. Concatenar los errores de cada una de las posibles resoluciones multiplica el número de posibles errores que pudo cometer el programador. ¿Que heurísticas podrían ser aplicadas para ordenar dichos errores por relevancia?

Resolución de operadores

El capítulo 4 muestra como la complejidad algorítmica del "peor caso" tiene un comportamiento $O(2^n)$. Pese a que se argumenta que esto no debería ser un problema, existe la pregunta abierta sobre el manejo de dicho caso. Introducción de *"time-out"* en la resolución, utilización de nuevas y mejores heurísticas, análisis del caso típico bien podrían mejorar el rendimiento del algoritmo de éste trabajo.

Apéndice A

Compilación y ejecución de Nano.

Nano es el lenguaje construido para este trabajo. Éste apéndice muestra como compilar y ejecutar el Intérprete.

A.1 Requisitos

1. **GHC**[1]: Éste es uno de los principales y mas completos compiladores para Haskell. Al momento de escribir este documento solo se ha hecho probado con la versión 6.10.1 del mismo. Otros compiladores como **Hugs**[2] y otras versiones de **GHC** pueden tambien ser suficientes dado que el compilador no hace uso extensivo de las extensiones provistas por GHC.
2. **Parsec3**: Ésta es la librería de "Parser combinadores" utilizada en la implementación de algunas partes de nano[3].
3. **Make**(Opcional)

A.2 Compilación y ejecución.

El código puede ser obtenido de `http://github.com/diegoeche/nano/tree/master`. Una vez descomprimido, es solo cuestión de.

```
#make all
#./Main
```

Si `make` no está disponible en su sistema:

[1]Puede ser descargado de `http://www.haskell.org/ghc/`
[2]Puede ser descargado de `http://haskell.org/hugs/`
[3]Puede ser instalada usando cabal-install `http://www.haskell.org/cabal/download.html`

```
#ghc -Wall --make *.hs
#./Main
```

Algunas advertencias pueden ocurrir en la construcción del interprete. Para salir del interprete puede utilizar el comando `:quit`.

A.3 Ejemplos

Si todo salió bien, la siguiente consola debe aparecer en el sistema:

```
Nano>
```

A.3.1 Tipos de datos

Datos primitivos

Nano soporta operaciones comunes con enteros. Cada evaluación de expresión muestra el tipo del valor evaluado, así como tambien la estructura interna en el lambda cálculo sin tipos que utiliza para reducción.

```
Nano> 65
Type: Int
Val: Const (IData 65)
Nano> add 23 12
Type: Int
Val: Const (IData 35)
Nano> times 2 4
Type: Int
Val: Const (IData 8)
Nano> sub 2 2
Type: Int
Val: Const (IData 0)
```

Existe también soporte de para cadenas de caracteres, pero en el momento hay ausencia de primitivas para manipularlas.

```
Nano> "Hello World!"
Type: String
Val: Const (SData "Hello World!")
```

Otro tipo de dato en Nano son Booleanos. Éstos estan implementados usando Booleanos de Church. La falta de *"Pretty Printer"* revela la verdadera estructura usada por el intérprete.

```
Nano> false
Type: Bool
Val: Lam "x" (Lam "y" (Var "y"))
```

```
Nano> true
Type: Bool
Val: Lam "x" (Lam "y" (Var "x"))
Nano> equals 3 3
Type: Bool
Val: Lam "x" (Lam "y" (Var "x"))
Nano> equals 2 3
Type: Bool
Val: Lam "x" (Lam "y" (Var "y"))
```

Listas y pares

Existen dos tipos adicionales que están parametrizados por otros tipos. El primero de estos son los pares. El constructor de éstas es la función `buildPair` de tipo `a -> b -> (a * b)`.

```
Nano> buildPair 3 3
Type: (Int * Int)
Val: Lam "x" (App (App (Var "x") (Const (IData 3))) (Const (IData 3)))
Nano> buildPair "Three" 3
Type: (String * Int)
Val: Lam "x" (App (App (Var "x") (Const (SData "Three"))) (Const (IData 3)))
Nano> fst buildPair 3 2
Type: Int
Val: Const (IData 3)
```

El lenguaje también posee listas. Éstas estan definidas internamente como pares que contienen un elemento en el primer lugar y el resto de la lista en el segundo lugar. El constructor de éstas es la función `cons` de tipo `a -> [a] -> [a]`. La función `empty` retorna una lista vacía de tipo `empty`

```
Nano> cons 1 cons 2 empty
Type: [Int]
Val: Lam "x" (App (App (Var "x") (Const (IData 1)))
    (Lam "x" (App (App (Var "x") (Const (IData 2)))
    (Lam "x" (Lam "x" (Lam "y" (Var "x")))))))
```

A.3.2 Operadores

El verdadero proposito de Nano es la implementación de operadores permisivos. Los operadores permisivos en Nano pueden ser prefijos, sufijos, infijos y cerrados. cada operador puede usar casi cualquier carácter para su definición.[4] Es posible asociar a los operadores infijos precedencia. Ésta es representada por un entero despues de la palabra clave `infixr` o `infixl`, las cuales indican asociatividad (derecha e izquierda respectivamente).

[4]Omitimos ciertos tipos para ahorrar algo de espacio.

```
Nano> let incrP x = add 1 x
Type: Int -> Int
Nano> incrP 3
Val: Const (IData 4)
Nano> let suffix x incrS = add 1 x
Type: Int -> Int
Nano> 3 incrS
Val: Const (IData 4)
Nano> let infixr 1 x + y = add x y
Type: Int -> Int -> Int
Nano> let infixr 2 x * y = times x y
Type: Int -> Int -> Int
Nano> 1 + 2 * 3
Val: Const (IData 7)
```

Igualmente pueden definirse operadores cerrados. Un ejemplo importante son los paréntesis. Es importante notar que el tipo de dicha función es polimórfico.

```
Nano> let closed ( x ) = x
Type: a0 -> a0
Nano> (1 + 2) * 3
Val: Const (IData 9)
```

A.3.3 Recursión

Las funciones recursivas deben usar la palabra clave `rec`. El siguiente es un caso de factorial.

```
Nano> let rec suffix n ! = if (equals n 0)
                              1
                             (n * (sub n 1)!)
Type: Int -> Int
Nano> 25!
Val: Const (IData 15511210043330985984000000)
```

Este último ejemplo ilustra la estrategia de reducción perezosa del lenguaje. Se implementa una lista infinita con todos los números de Fibonacci. Inicialmente implementamos una función `zipAdd` que suma los elementos de 2 listas. Definimos los números de Fibonacci en forma similar a como lo haríamos en Haskell. La evaluación perezosa permite la que la función `fibNumbers` termine. El carácter \ nos permite escapar el salto de línea.

```
Nano> let rec zipAdd l1 l2 = if (or (isNull l1) (isNull l2)) \
                                empty \
                               (cons (add (hd l1) (hd l2)) \
                                   (zipAdd (rest l1) (rest l2)))
Type: [Int] -> [Int] -> [Int]
```

```
Nano> let rec fibNumbers = cons 1 cons 1 (zipAdd (fibNumbers) \
                                                 (rest fibNumbers))\
Type: [Int]
Nano> let rec infixr l .[ n = if (neg (equals n 0)) \
                                 ((rest l).[sub n 1)\
                                 (hd l)
Type: [a3] -> Int -> a3
Nano> let suffix n ] = n
Type: a0 -> a0
Nano> (cons 3 cons 2 empty).[0]
Val: Const (IData 3)
Nano> (fibNumbers).[11]
Type: Int
Val: Const (IData 144)
```

Bibliografía

[Aas91] Anika Aasa, *Precedences in specications and implementations of programming languages*, Proceedings of the Third international Symposium on Programming Language Implementation and Logic Programming, vol. 528, 1991.

[Aas92] ———, *User defined syntax*, Ph.D. thesis, Chalmers University of Technology, 1992.

[AS96] Harold Abelson and Gerald J. Sussman, *Structure and interpretation of computer programs - 2nd edition (mit electrical engineering and computer science)*, The MIT Press, July 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BB00] 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.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella, *Hope: An experimental applicative language*, LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming (New York, NY, USA), ACM, 1980, pp. 136–143.

[BS] Arthur Baars and Doaitse Swierstra, *Syntax macros*, Unfinished Draft.

[CDE+99] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada, *Maude: Specification and programming in rewriting logic*, Tech. report, Computer Science Laboratory SRI International, 1999.

[CH03] James Cheney and Ralf Hinze, *Phantom types*, 2003.

[Cha99] G. J. Chaitin, *People and ideas in theoretical computer science*, ch. Elegant LISP Programs, Springer-Verlag, 1999, Disponible en línea: `http://www.cs.auckland.ac.nz/CDMTCS//chaitin/lisp.html`.

[CMA94] Luca Cardelli, Florian Matthes, and Martin Abadi, *Extensible syntax with lexical scoping*, Tech. report, Systems Research Center, 1994.

[CW85] Luca Cardelli and Peter Wegner, *On understanding types, data abstraction, and polymorphism*, ACM Comput. Surv. **17** (1985), no. 4, 471–522.

[dC07] Equipo de Coq, *The Coq Proof Assistant Reference Manual, Version 8.1.*, 2007.

[DN08] Nils Anders Danielsson and Ulf Norell, *Parsing mixfix operators*, Preliminary Draft in Citeseer, 2008.

[Gra07] Martin Grabmüller, *Algorithm w step by step*, Draft paper, September 2007.

[GWM+96] Joseph A. Goguen, Timothy Winkler, Jose Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud, *Introducing OBJ*, Tech. report, Menlo Park, 1996.

[Jon86] Simon Peyton Jones, *Parsing distfix operators*, Communications ACM **29** (1986), no. 2, 118–122.

[Jon87] ______, *The implementation of functional programming languages*, Prentice Hall, 1987.

[LM01] Daan Leijen and Erik Meijer, *Parsec: Direct style monadic parser combinators for the real world*, Tech. Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[LvL94] Jan Leeuwen and Jan van Leeuwen (eds.), *Handbook of theoretical computer science: Formal models and semantics*, MIT Press, 1994.

[MSD10] MSDN, *Xml literals overview*, Disponible en línea en: `http://msdn.microsoft.com/en-us/library/bb384629.aspx`, 2010.

[Nor99] Theodore Norvell, *Parsing expressions by recursive descent*, Disponible en línea en `http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm`, 1999.

[Pie02] Benjamin C. Pierce, *Types and programming languages*, The MIT Press, 2002.

[Ray03] Eric S. Raymond, *The art of unix programming*, Pearson Education, 2003.

[Sel] 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.

[Sim99] Charles Simonyi, *Hungarian notation*, Disponible en línea en: `http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx`, November 1999.

[Ste99] Guy L. Steele, Jr., *Growing a language*, Higher Order Symbol. Comput. **12** (1999), no. 3, 221–236.

[Tul00] Mark Tullsen, *First class patterns*, In 2nd International Workshop on Practial Aspects of Declarative Languages, volume 1753 of LNCS, Springer-Verlag, 2000, pp. 1–15.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle, *Organizing programs without classes*, Lisp and Symbolic Computation (1991), 37–56.

[Wig01] Ulf Wiger, *Four-fold increase in productivity and quality*, Tech. report, Ericsson, 2001.

[wika] *Wiki de agda*, Disponible en línea en: `http://wiki.portal.chalmers.se/agda/`.

[Wikb] Haskell Wiki, *Polyvariadic functions*, Disponible en línea `http://haskell.org/haskellwiki/Polyvariadic_functions`.