UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

TÉCNICAS PARA CONSTRUIR COMPILADORES EFICIENTES EN HASKELL.

T E S I S

QUE PARA OBTENER EL TÍTULO DE

LICENCIADA EN CIENCIAS DE LA C O M P U T A C I Ó N

P R E S E N T A :

LAURA ALICIA LEONIDES JIMÉNEZ

DIRECTOR DE TESIS: DR. FRANCISCO HERNÁNDEZ QUIROZ

2005

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"Técnicas para construir compiladores eficientes en Haskell"

realizado por Laura Alicia Leonides Jiménez con número de cuenta 9710112-5

quién cubrió los créditos de la carrera de Lic. en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Dr. Francisco Hernández Quiroz	
Propietario	Dra. Sofía Natalia Galicia Haro	
Propietario	Lic. en I. Elías Samra Hassán	
Suplente	Lic. en C.C. Karla Ramírez Pulido	
Suplente	Dr. Julio César Peralta Estrada	

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

A mi mamá, por haber estado siempre a mi lado, apoyando mis decisiones. A mi abue y a Citla, por la confianza y el cariño constante. A mi abuelo, por haberme cuidado y animado todo este tiempo.

A Aldo, por todo el amor, la ternura y la locura. Gracias por estar siempre conmigo.

A Rodrigo, Sergio, Gustavo, Paty, Mau y Clau, por haber hecho mi estadía en la Facultad, no sólo enriquecedora, sino también extremadamente divertida.

A Rulo, por el apoyo incondicional y las sonrisas compartidas.

A toda mi familia y amigos, sin cuyo apoyo esto no sería posible.

A mis profesores, por sus enseñanzas y consejos.

Índice general

Capítulo 1

Introducción

El desarrollo de programas a gran escala en Haskell presenta un reto interesante, ya que plantea problemas a resolver con un enfoque diferente al comúnmente utilizado, debido al uso de la programación funcional. Para resolver las dificultades encontradas durante el desarrollo de este proyecto, se consideraron características de los lenguajes funcionales puros como la evaluación perezosa, las funciones de orden superior y el uso de mónadas.

En el presente trabajo se exponen las herramientas necesarias para construir un compilador de IIFTEX a LATEX usando como lenguaje de programación a Haskell. Desarrollé el compilador durante mi servicio social, el cual realicé en la Facultad de Ciencias y el Departamento de Publicaciones del Instituto de Investigaciones Filosóficas de la UNAM. Decidí realizarlo en Haskell porque me pareció un ejercicio interesante desarrollar un compilador de este tipo utilizando un lenguaje funcional.

1.1. Breve descripción e historia de IIFTEX

El Departamento de Publicaciones del Instituto de Investigaciones Filosóficas tiene un largo historial de edición por computadora. Desde 1987 ha producido libros, folletos, revistas y otros materiales hemerográficos con el auxilio de programas de cómputo y conserva copias digitales de la mayor parte de este trabajo. Dichos archivos de texto contienen las convenciones de distintos editores: Word, Ventura Publisher y TEX (en diversos dialectos). Además, han iniciado la publicación electrónica de diversos materiales y quieren extenderla al formato HTML.

TEX es en esencia un conjunto de instrucciones primitivas que permiten al usuario ampliar, completar y modificar dichas instrucciones para adaptarlo a sus necesidades particulares. En el IIF crearon su propio conjunto de macros para satisfacer sus necesidades de edición. Este conjunto de macros no posee un nombre particular, pero en este documento lo nombraremos IIFTEX para poder diferenciarlo de otros paquetes que han surgido con el tiempo.

En 1987 TEX fue introducido al IIF por Adolfo García de la Sienra. Tiempo después, Miguel Navarro Saab introdujo Plain TEX con algunos paquetes adicionales para cubrir los requerimientos de los documentos generados por el Departamento de Publicaciones del IIF; Plain TEX fue usado en dicha institución durante dos años. En 1990 Antonio Zirión y Francisco Hernández Quiroz efectuaron un rediseño de los macros y crearon nuevos paquetes relacionados con diferentes colecciones del instituto.

Francisco Hernández continuó realizando modificaciones posteriores a los paquetes, hasta que en 1993 se concluyeron los paquetes con los que el Departamento de Publicaciones del IIF habría de trabajar durante los siguientes diez años. Este conjunto de paquetes es al que llamaré IIFTEX en este documento.

Debido a la manera en que fue desarrollado IIFTEX, no se cuenta con documentación de los paquetes, salvo algunos comentarios hechos por los desarrolladores dentro de las definiciones de los macros. Esta es la razón por la cual, para poder entender el funcionamiento del sistema, fue necesario mantener entrevistas tanto con los usuarios de IIFTEX, como con Francisco Hernández Quiroz, uno de sus desarrolladores.

El Departamento de Publicaciones del IIF cuenta con la mayoría de su acervo en IIFTEX y debido a su deseo de actualizarlo a formatos usados más ampliamente hoy en día, decidieron transformar dichos archivos a LATEX.

Los programas de cómputo existentes (tanto comerciales como gratuitos) no cubrían todas sus necesidades, tales como poder personalizar la forma en que serán convertidos los archivos, poder cambiar las instrucciones especiales para acentos por su equivalente en español (como en el caso de \'e → é) y considerar las diferentes clases de documentos que tienen definidos. Por este motivo, solicitaron al asesor de este proyecto su apoyo para elaborar nuevos programas que satisfagan las necesidades de conversión del IIF.

1.2. Plan de trabajo y objetivos del proyecto

El plan de trabajo que se siguió durante el desarrollo del compilador fue el siguiente:

1. A partir de la descripción informal de las necesidades del Departamento de Publicaciones del IIF, elaborar un documento con una descripción detallada de las tareas que debe realizar el software del presente proyecto.

2. Con base en la descripción resultante del objetivo 1, producir un programa que realice dichas tareas.

3. Sentar las bases para la posterior construcción de una interfaz conveniente para los usuarios del programa ya mencionado.

Los objetivos que se desean alcanzar con la elaboración del compilador en Haskell son:

1. Adquirir experiencia en la producción de software utilizando el lenguaje Haskell.

2. Contar con un ejemplo de lo que se puede alcanzar con dicho lenguaje.

3. Contribuir al acervo existente de software bajo la licencia GNU/Linux.

Capítulo 2

Haskell: un lenguaje funcional

2.1. Lenguajes funcionales

Lenguajes de programación como Fortran y C son llamados lenguajes de programación imperativos porque consisten en secuencias de acciones. El programador debe indicarle a la computadora cómo llevar a cabo una tarea paso a paso. Los lenguajes funcionales trabajan de manera distinta: en lugar de llevar a cabo acciones, evalúan expresiones.

La programación funcional está basada en la idea de la evaluación. Se definen las funciones que se van a utilizar y la implementación evalúa las expresiones que usan dichas funciones.

Como lo menciona Backus [2], los lenguajes funcionales están basados en un conjunto de formas funcionales, las cuales, junto con algunas definiciones simples son la base para formar nuevas funciones. En estos lenguajes, una vez que una variable ha sido asociada a un valor, el valor asociado no puede

ser modificado, es decir, no existe la noción de asignación destructiva como en los lenguajes imperativos.

A continuación muestro la descripción de un lenguaje funcional, presentada por Backus [2]. En este artículo fueron presentados por primera vez los fundamentos de los lenguajes funcionales con un enfoque algebraico. Algunas de las características del lenguaje descrito por Backus no se tienen en los lenguajes actuales, pero debido a la importancia histórica de esta descripción, es importante tomarla en cuenta.

En dicha publicación se muestra una descripción genérica de los lenguajes de programación funcionales. Sin embargo, los lenguajes modernos como Haskell rebasan dicha descripción, ya que incluyen características como la evaluación perezosa, la cual va más allá de la descripción de Backus.

Un lenguaje funcional está formado por lo siguiente:

- Un conjunto O de objetos.
- Un conjunto F de funciones f que van de objetos a objetos.
- Una operación, *aplicación.*
- Un conjunto F' de formas funcionales, éstas son usadas para combinar funciones existentes, u objetos, para formar nuevas funciones en F.
- Un conjunto D de definiciones de algunas funciones en F' asignándoles un nombre.

Un objeto x es un átomo, una secuencia $< x_1, \ldots, x_n >$ cuyos elementos x_i son objetos, o $\perp$ (indefinido). El conjunto A de átomos es el conjunto de

cadenas no vacías de letras mayúsculas, dígitos y símbolos no usados por la notación del sistema de programación funcional. El átomo ϕ es usado para denotar la secuencia vacía y es el único objeto que es un átomo y una secuencia. Los átomos T y F se usan para denotar verdadero y falso, respectivamente. Ninguna secuencia tiene a $\perp$ como elemento, es decir, si x es una secuencia con $\perp$ como elemento, entonces $x = \perp$.

La aplicación es la única operación con la que cuenta un sistema de programación funcional. Si f es una función y x es un objeto, entonces —usando la notación empleada por Backus— $f : x$ es una *aplicación* y denota al objeto que resulta de aplicar f a x.

Todas las funciones f de F van de objetos a objetos y preservan a indefinido ($\perp$). Cuando un lenguaje cumple con esta característica se dice que es un *lenguaje estricto.* Cada función de F es una *primitiva*, esto es, dada por el sistema, está *definida* o es una *forma funcional.*

Vale la pena distinguir dos casos en que $f : x = \perp$. Si la evaluación para $f : x$ termina y lleva al resultado $\perp$, se dice que f *no está definida en* x. En otro caso se dice que f *no termina en* x.

Algunas de las funciones primitivas presentadas por Backus en [2] son la identidad, el átomo, reversa, concatenación, selectores, distribución derecha e izquierda, tamaño, así como funciones de comparación y operadores lógicos.

Formas funcionales

Una forma funcional es una expresión que denota una función; dicha función depende de las funciones u objetos que son los *parámetros* de la expresión. A continuación se encuentran algunos ejemplos de formas funcionales:

Composición $(f \circ g) : x = f : (g : x)$, la cual puede ser aplicada a:
$(identity \circ tail) :< 1,2,3 >=< 2,3 >$

Construcción $[f_1, \ldots, f_n] : x =< f_1 : x, \ldots, f_n : x >$, la cual puede se aplicada a:
$[tail, reverse] :< 1,2,3 >=<< 2,3 >, < 3,2,1 >>$

Condicional $(p \rightarrow f; g) : x = (p : x) = T \rightarrow f : x; (p : x) = F \rightarrow g : x; \perp$, la cual puede ser aplicada a:
$(atom \rightarrow 0; 1) :< 1,2,3 >= 1$

Constante $\bar{x} : y = y = \perp \rightarrow \perp; x$, que puede ser aplicada a:
$\bar{1} : 10 = 1$

Inserción $/f : x = x =< x_1 > \rightarrow x_1;$
$x =< x_1, \ldots, x_n > \& \; n \geq 2 \rightarrow f :< x_1, /f :< x_2, \ldots, x_n >>; \perp$, que puede ser aplicada a:

$$/+ :< 1,2,3 >= + :< 1, + :< 2, /+ :< 3 >>>$$
$$= + :< 1, + :< 2,3 >>$$
$$= 6$$

Aplicar a todos $\alpha f : x = x = \phi \rightarrow \phi;$
$x =< x_1, \ldots, x_n > \rightarrow < f : x_1, \ldots, f : x_n >; \perp$, que puede aplicarse a:
$\alpha\bar{3} :< 1,3,7 >=< 3,3,3 >$

Definiciones

Una definición en un sistema de programación funcional es una expresión de la forma

$$\text{Def } l = r$$

donde el lado izquierdo l es un símbolo de función nuevo y el lado derecho r es una forma funcional. De esta forma se expresa que el símbolo l denota a la función dada por r. A continuación presento un ejemplo de definición para el factorial:

$$\text{Def } ! = eq0 \rightarrow \bar{1}; \times \circ [id, ! \circ sub1]$$

$$donde$$

$$\text{Def } eq0 = eq \circ [id, \bar{0}]$$

$$\text{Def } sub1 = - \circ [id, \bar{1}]$$

2.2. Principales características de Haskell

Como menciona Simon Peyton [19], Haskell es un lenguaje funcional puro de propósito general, que incorpora muchas de las innovaciones más recientes en el diseño de los lenguajes de programación. Haskell fue desarrollado para incorporar la sabiduría colectiva de la comunidad de los lenguajes funcionales en un lenguaje elegante, poderoso y general.

Haskell provee de funciones de orden superior, semántica no estricta, sistema de tipos fuerte, tipos de datos algebraicos, concordancia de patrones, listas por comprensión, sistema modular, polimorfismo ad-hoc y paramétrico, sistema monádico de entrada y salida y un rico conjunto de tipos primitivos de datos, incluyendo listas, arreglos, enteros de precisión arbitraria o fija y números de punto flotante.

La característica más importante de Haskell es su pureza. No permite ningún efecto secundario. Otra característica importante es que es perezoso, esto

significa que nada es evaluado hasta que *debe* ser evaluado, y una vez evaluado se conserva su resultado. De esta manera se pueden definir listas infinitas que serán de gran utilidad para resolver algunos problemas. Por ejemplo, una estrategia utilizada comúnmente para resolver problemas es crear una lista de todas las posibles soluciones y después filtrar la lista para eliminar los elementos inadecuados. La evaluación perezosa de Haskell hace esta tarea sumamente clara y sencilla. Si se necesitara únicamente una solución, se puede pedir sólo el primer elemento de la lista y la evaluación perezosa se encargará de que no se evalúe algo innecesario.

Además Haskell es fuertemente tipificado. Es imposible que por error se le asigne un valor real a un entero, esto es de gran utilidad para reducir los errores en tiempo de ejecución en los programas porque muchos errores se detectan en tiempo de compilación.

Como señala Sylvan [21], en Haskell los tipos son inferidos automáticamente. Esto quiere decir que rara vez debe declararse el tipo de una función, a no ser con propósitos de documentación de código. Para la inferencia de tipos, Haskell se fijará en cómo son usadas las variables y a partir de esto deducirá el tipo del que debe ser la variable, después hará una revisión de tipos para asegurarse de que no haya errores de tipos. Haskell siempre inferirá el tipo más general de una variable por medio de la unificación de tipos, lo cual brinda polimorfismo.

Haskell tiene otra propiedad que es muy valiosa para los programadores, a pesar de que no significa mucho en términos de estabilidad o desempeño: *elegancia*. Las cosas funcionan tal y como se espera que funcionen. Haskell provee al programador una manera elegante, concisa y segura de escribir programas.

2.3. Tipos en Haskell

Todo objeto en Haskell pertenece a un tipo, por esto es necesario que las funciones sean aplicadas únicamente a objetos del tipo apropiado. Para representar que un objeto o tiene tipo α, se utiliza la notación o :: α. En Haskell existen tipos monomórficos y polimórficos.

2.3.1. Tipos monomórficos

Como se menciona en [23], las reglas para la aplicación de funciones a objetos son las siguientes:

Aplicación. Si f :: $\alpha \to \beta$ y e :: α, entonces f puede ser aplicada a e y f e :: β.

Cancelación. Si f :: $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \alpha$ y e_1 :: α_1, ..., e_k :: α_k, donde $k \leq n$, entonces f puede ser aplicada a $e_1, \ldots, e_k$ y el resultado de f $e_1 \ldots e_k$ :: $\alpha_{k+1} \to \alpha_{k+2} \to \ldots \to \alpha_n \to \alpha$.

Condicional. Si c :: Bool y e_1, e_2 :: α, entonces if c then e_1 else e_2 :: α.

Función. Si x :: α y e :: β, entonces λx.e :: $\alpha \to \beta$.

Tupla. Si e_1 :: α_1, ..., e_k :: α_k, entonces (e_1, ..., e_k) :: (α_1, ..., α_k).

Lista. Si e_1, ..., e_k :: α, entonces [e_1, ..., e_k][1] :: [α][2].

[1] Las listas en Haskell son representadas entre corchetes ([]), aunque en realidad eso es sólo otra forma de escribir $e_1 : \ldots : e_n : []$, donde : es asociativo por la derecha.

[2] A diferencia de las listas en Sheme, donde los elementos puedn ser de distintos tipos.

Las definiciones en Haskell consisten de cierto número de ecuaciones condicionales, cada una de las cuales puede contener múltiples cláusulas y tener definiciones locales. Las restricciones en este caso son las siguientes:

Guardia. Cada guardia debe ser una expresión de tipo `Bool`. Por ejemplo en:

```
sonIguales n m
    | n == m =True
    | otherwise =False
```

`n == m :: Bool` y `otherwise :: Bool`. Si éste no fuera el caso, ocurriría un error.

Lado derecho. Las expresiones en cada cláusula de una ecuación condicional deben tener el mismo tipo `t`. Como se puede observar en el ejemplo anterior, las expresiones del lado derecho son del mismo tipo, es decir, **True** y **False** son ambas de tipo `Bool`.

Patrones. Ningún patrón debe tener conflicto con los tipos de los argumentos de la función, en caso de que no coincidieran, ocurrirá un error en tiempo de compilación. Por ejemplo en:

```
miFunción :: Int → Int → Char
miFunción 1 1 ='a'
miFunción 2 0 ='b'
```

```
incorrecta 0 = 1
incorrecta 'B' = 7
```

`1`, `0` y `2` pertenecen al tipo `Int` y por lo tanto no hay un conflicto entre los tipos de éstos y los de los argumentos de la función.

Definiciones locales. Puede haber tipos para definir de manera local funciones y objetos. Las definiciones en una cláusula `where` deben adaptar-

se a las restricciones de tipos anteriores, dada la información adicional sobre las variables locales usadas en las definiciones. Por ejemplo en:

```
sumaCuadrado n m
    = sqN + sqM
        where
        sqN =n ×n
        sqM =m ×m
```

Si el tipo de `sqN` y de `sqM` no fuera `Int`, existiría un conflicto con el tipo de +.

2.3.2. Tipos polimórficos

Haskell cuenta también con variables de tipos, además de las variables convencionales. Las variables de tipos en Haskell se escriben exactamente igual que las variables comunes, pero su significado es distinto. Una variable de tipo no representa a una expresión, sino a un tipo. A lo largo de este documento representaré las variables de tipos con las primeras letras del alfabeto griego $(\alpha, \beta, \gamma \ldots)$, aunque se codifican con nombres que comienzan con minúscula: alfa, beta, gamma, etc.

Cuando una función u objeto tiene un tipo que consta de una o más variables de tipo, se dice que es de tipo polimórfico, como sucede en el caso de la función `foldl`:

```
foldl :: (α → β → α) → α → [β] → α
foldl f a []     = a
foldl f a (b:x) = foldl f (f a b) x
```

La cual actúa sobre los tipos α y β, sin importar qué tipos sean. Entonces la función `foldl` es de tipo polimórfico. Ésta puede ser aplicada a distintos valores, como:

```
f1 = foldl (+) 0 [1..7]
f2 = foldl (++) "" ["éste", " es", " un", " ejemplo"]
```

a pesar de que los tipos de (+) y (++) son diferentes:

$$(+) :: (\mathbf{Num}\ \alpha) \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$
$$(++) :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$$

así como los tipos de `0` y `""`:

$$0 :: (\mathbf{Num}\ \beta) \Rightarrow \beta$$
$$\texttt{""} :: \mathbf{[Char]}$$

Una de las principales ventajas de contar con el polimorfismo, es que permite reutilizar las definiciones, ya que no las liga a un tipo particular. Una misma función puede ser aplicada a objetos de distintos tipos sin tener que reescribir el cuerpo de la función.

2.3.3. Tipos de datos algebraicos

Los tipos de datos algebraicos pueden ser usados para modelar cosas que sería difícil describir con los tipos y constructores básicos, como cadenas (`String`), números enteros (`Int`) y listas, por ejemplo, en el caso de árboles:

```
data Tree α= NilTree | Fork { leftTree, rightTree :: Tree α } | Leaf
    { elem :: α }
```

En este caso se dice que `NilTree` y `Fork` son los constructores del tipo Tree α, y `leftTree` y `rightTree` son selectores.

Al definir un tipo de datos algebraico se señalan los constructores a utilizar para crear valores del tipo a definir. Para definir funciones sobre estos tipos es necesario utilizar concordancia de patrones, tomando los tipos de datos definidos anteriormente, podemos construir la función:

```
inOrder :: (Show α) ⇒ Tree α → String
inOrder NilTree  = ""
inOrder (Leaf a) = show a
inOrder (Fork l r) = (inOrder l) ++ (inOrder r)
```

Los tipos de datos pueden ser recursivos, se puede usar el nombre del tipo que se está definiendo como parte de cualquiera de los parámetros de sus constructores, como se haría para listas y árboles. El nombre del tipo puede estar seguido de una o más variables de tipos, las cuales pueden ser usadas en el lado derecho, haciendo que la definición sea polimórfica.

2.4. Clases en Haskell

Una clase en Haskell es una colección de tipos que a los que pueden aplicarse ciertas operaciones, llamadas métodos de la clase. Las clases en Haskell proporcionan un polimorfismo diferente al tratado en la Sección 2.3.2, a este polimorfismo se le conoce como polimorfismo ad-hoc o sobrecarga.

Se dice que un operador está sobrecargado cuando puede aplicarse a distintos tipos de datos, y además puede estar definido de distinta manera para cada uno de estos tipos. Esto difiere del polimorfismo antes visto (llamado polimorfismo paramétrico) en que en el polimorfismo paramétrico el operador

se comporta igual sin importar el tipo del objeto al que se le aplica y en el polimorfismo ad-hoc éste no es el caso.

Un ejemplo familiar de sobrecarga de un operador es el operador ==. Éste puede ser aplicado a números enteros, reales, cadenas, booleanos y a muchos otros tipos en Haskell. Esto es gracias a la existencia de la clase `Eq`, la cual contiene la definición:

class Eq where
$(==), (\neq) :: \alpha \rightarrow \alpha \rightarrow \textbf{Bool}$

`Eq` es el nombre de la clase, y (==) y (/=) son los métodos de la clase. Esta declaración debe leerse: "un tipo es una instancia de la clase `Eq` si hay una operación == y una operación /=, del tipo apropiado, definida en ella".

La restricción de que un tipo debe ser una instancia de la clase `Eq` es escrita `Eq` α. Entonces `Eq` no es una expresión de tipo, sino que expresa una restricción en un tipo, y es llamada contexto. Los contextos se colocan al principio de las expresiones de tipo. Por ejemplo:

$(==) :: (\textbf{Eq}\ \alpha) \Rightarrow \alpha \rightarrow \alpha \rightarrow \textbf{Bool}$

expresa que para todo tipo α que sea una instancia de la clase `Eq`, `==` tiene tipo $\alpha \rightarrow \alpha \rightarrow$ `Bool`.

2.5. Evaluación perezosa

Algunas veces en una llamada, una función nunca hace referencia a uno o más de sus parámetros formales. En este caso el tiempo invertido en evaluar los operandos correspondientes es desperdiciado. También puede ser que la

evaluación de tales operandos pueda resultar en un error o nunca terminar. Esperar hasta el último momento posible para evaluar una expresión, especialmente con el propósito de optimizar un algoritmo que puede no usar el valor de la expresión, es conocido como evaluación perezosa.

Por ejemplo, en:

```
f x y = if x = 0 then 0 else y
```

si se evalúa `f 0 1/0`, no se llegará a tener un error, ya que $\frac{1}{0}$ no será evaluado nunca.

La evaluación perezosa es bastante útil cuando una expresión es muy cara o imposible de evaluar y puede no ser necesitada para nada. También es usada para definir estructuras infinitas de manera recursiva. Dado que cada nivel de recursión es evaluado sólo cuando se necesita, los datos sólo son generados mientras son consumidos y la evaluación de la estructura de datos puede terminar cuando el consumo es completado.

La noción de *evaluación perezosa* puede ser extendida para que el valor de una expresión sea usado de manera intercambiable con la expresión misma. Esta extensión de la *evaluación perezosa* y el compartir el valor de la expresión evaluada es usado para implementar la semántica del *llamado por necesidad*. El *llamado por necesidad* esencialmente significa que las expresiones sólo son evaluadas una vez y sólo si la evaluación es en realidad necesaria. Todas las futuras instancias de la expresión son intercambiadas directamente por el valor calculado.

La evaluación perezosa necesita que los datos usados en el cálculo estén disponibles en el momento de la evaluación. En los lenguajes de programación funcionales, como Haskell, esta condición es garantizada por la manera en

que son evaluadas todas las expresiones. En lenguajes funcionales no puros, como Scheme, se puede hacer uso de la instrucción `delay` para retrasar la evaluación de una expresión.

Muchas discusiones sobre la evaluación perezosa se concentran en aspectos de rendimiento de la evaluación perezosa. Sin embargo, lo más importante de la evaluación perezosa es que provee nuevos medios para resolver tareas de programación. Por ejemplo, partes de un programa pueden comunicarse fácilmente con ayuda de estructuras de datos potencialmente infinitas.

La evaluación perezosa acelera el tiempo de creación e inicialización, pero esta ganancia tiene un costo: con ciertas implementaciones la rapidez con que son evaluadas algunas expresiones puede disminuir. También ahorra mucha memoria al no evaluar ninguna expresión hasta que ésta sea necesitada sin ningún costo real en el programa. Es común concentrarse en el rendimiento al hablar de la evaluación perezosa, pero el ahorro de memoria también es muy importante. Para mayor información sobre el tema puede consultarse Efficient Compilation of Lazy Evaluation [10].

2.6. Mónadas

En Haskell se sabe qué se va a obtener al evaluar una expresión, pero no cómo, no se conocen los estados intermedios del cálculo. Una secuencia de operaciones está formada por un conjunto de operaciones que deben efectuarse una después de otra, en el orden especificado. Las mónadas nos permiten establecer el orden en que se evaluará una secuencia de operaciones.

El concepto de mónada, el cual surge de la teoría de categorías, permite al programador construir cálculos usando bloques de construcción secuencial,

que pueden a su vez ser secuencias de cálculos. Una mónada es una manera de estructurar los cálculos en términos de los valores y las secuencias de cálculos usando dichos valores. En Haskell, las mónadas tienen un papel muy importante en el sistema de entrada y salida.

Una mónada es una triada $(m, unit, >>=)$ que consiste de un constructor de tipos m y dos operaciones. Los tipos de estas operaciones son:

$$unit \;::\; \alpha \;\rightarrow m\;\alpha$$

$$(>>=) \;::\; m\;\alpha \;\rightarrow (\alpha \;\rightarrow m\;\beta) \;\rightarrow m\;\beta$$

unit sirve para encapsular un valor dentro de una mónada y $(>==)$ nos permite aplicar una función de tipo $\alpha \;\rightarrow m\;\beta$ a una función de tipo $m\;\alpha$.

En una expresión de la forma

$$e >>= \lambda a.n$$

e y n son expresiones, y a es una variable. La forma $\lambda a.n$ es una expresión lambda, en la cual el alcance de la variable a es la expresión n, lo que esta expresión denota es: evalúa la expresión e, liga a al resultado, y evalúa n. Del tipo de $(>>=)$, se puede ver que la expresión e debe ser de tipo $m\;a$, la variable a tiene tipo a, la expresión n es de tipo $m\;b$, la expresión lambda $\lambda a.n$ tiene tipo $a \;\rightarrow m\;b$, y el tipo del valor que regresa es $m\;b$.

La expresión

$$e >>= \lambda a.n$$

es análoga a

$$\texttt{let}\; a \;=\; e\; \texttt{in}\; n$$

2.6.1. Las leyes monádicas

Como menciona Wadler [26], las operaciones de una mónada satisfacen tres leyes:

Neutro izquierdo. Calcular el valor de a, ligar b al resultado, y calcular n. El resultado es el mismo que n con el valor a sustituido por la variable b.

$$unit\ a >>= \lambda b.n = n[a/b]$$

Neutro derecho. Calcular m, ligar el resultado a a, y regresar a. El resultado es el mismo que m.

$$m >>= \lambda a.unit\ a = m$$

Asociatividad. La operación $(>>=)$ es asociativa.

$$m >>= (\lambda a.n >>= \lambda b.o) = (m >>= \lambda a.n) >>= \lambda b.o$$

El alcance de la variable a incluye a o en el lado izquierdo, pero excluye a o en el derecho, por eso esta ley es válida sólo cuando a no ocurre libre en o.

2.6.2. Mónadas en Haskell

En Haskell una mónada es representada como un constructor de tipos `m`, una función que construye valores del tipo $\alpha \rightarrow$ `m` α, y una función que combina valores del tipo `m` α para producir cálculos más colmplejos. La función que construye valores del tipo `m` α es llamada `return` y la función que los combina es conocida como `bind`, pero es escrita como $>>=$.

La definición de la clase `Monad` en Haskell es la siguiente:

```
class Monad m where
    (>>=)  :: m α → (α → m β) → m β
    (>>)   :: m α → m β → m β
    return :: α → m β
    fail   :: String → m α

    m >>k = m >>= λ_. k
    fail s = error s
```

2.6.3. Notación *do*

La notación *do* es una abreviatura para construir cálculos monádicos. Ésta permite escribir cálculos monádicos usando un estilo pseudo-imperativo con variables. El resultado del cálculo monádico puede ser "asignado" a una variable usando el operador ← y al usar esa variable en un cálculo monádico posterior, se lleva a cabo el ligado automáticamente. El tipo de la expresión a la derecha de la flecha es el tipo monádico `m` α. La expresión a la izquierda de la flecha es un patrón que concuerda con el valor dentro de la mónada. El operador ← no es una asignación en el sentido en que se usa en los lenguajes imperativos. No es destructiva, por lo que no se permite modificar el valor "asignado" a una variable. A continuación se muestran ejemplos de un uso correcto y uno incorrecto de dicho operador:

```
correcto x =          | incorrecto x =
    do a ← x          |     do y ← x
       b ← x + x      |        y ← x + x
       ⋮              |        ⋮
```

La notación *do* se parece a un lenguaje de programación imperativo, en el cual un cálculo es construido a partir de una secuencia explícita de cálculos más simples. Esta notación es simplemente una manera más sencilla de utilizar mónadas en Haskell. No hay nada que pueda ser hecho usando la notación *do*, que no pueda ser hecho usando los operadores monádicos estándar. Pero la notación *do* es más clara y más convincente en algunos casos.

La traducción de una notación *do* a operadores monádicos estándar se ejemplifica a continuación[3]:

```
putStr "x: " >>                    do putStr "x: "
getLine >>=λl.          ⇔             l ←getLine
    return (words l)                  return (words l)
```

2.6.4. Enmascarado por mónadas

Haskell permite la creación de mónadas de un solo sentido. Las mónadas de un solo sentido permiten que los valores entren a la mónada a través de la función `return` y permiten que los cálculos sean realizados dentro de la mónada usando las funciones >>= y >>, pero no permiten que los valores se extraigan de la mónada.

La mónada `IO`[4] es un ejemplo de una mónada de un solo sentido. No se puede escapar de la mónada `IO`, es imposible escribir una función que realice un cálculo en la mónada `IO` y el tipo de su resultado no incluya el constructor

[3]Para mayor información sobre el tema puede consultarse la Sección 3.14 del Reporte de Haskell [19].

[4]IO es la mónada de entrada y salida, es llamada así por sus siglas en inglés Input-Output

de tipos `IO`. Esto significa que cualquier función que regrese una expresión cuyo tipo no contenga el constructor de tipos `IO` no usa la mónada `IO`. Otras mónadas como `List` y `Maybe` sí permiten que los valores salgan de la mónada. De esta manera es posible escribir funciones que usen estas mónadas internamente, pero que regresen valores no monádicos.

Una mónada de un solo sentido crea de manera eficiente un dominio computacional aislado, en el cual las reglas de un lenguaje funcional puro pueden relajarse. Los cálculos funcionales pueden trasladarse al dominio, pero los peligrosos efectos laterales y las funciones no transparentes no pueden escapar de él.

Utilizando las mónadas de un solo sentido, pueden efectuarse acciones, como en el caso del ejemplo presentado por Wadler [27]:

```
putc 'x'
```

si alguna vez se evalúa, efectúa la acción de imprimir `x` y devuelve el valor `IO()`, una acción de entrada y salida. Ahora se puede comparar el comportamiento de la función `putc` en StandardML y en Haskell, para mayor claridad a la función `putc` en StandardML será llamada `putcML`.

En StandardML, el evaluar la expresión

```
putcML #"h"; putcML #"a";
putcML #"h"; putcML #"a"
```

como efecto lateral crea una acción, la cual es un objeto que al regresar imprime "haha".

imprime "haha" como efecto lateral, pero

```
let val x = (putcML #"h"; putcML #"a")
in x; x end
```

imprime sólo "ha".

En Haskell, la expresión:

```
putc 'h' >>putc 'a' >>
putc 'h' >>putc 'a'
```

y la expresión

```
let x = (putc 'h' >>putc 'a')
in x >>x
```

son equivalentes. Esto sucede porque en StandardML 'h' sólo se evalúa una vez dentro del `let`, mientras que en Haskell la semántica es la de llamada por nombre, con las ventajas de las llamadas por valor, pero el valor de 'h' es una acción pasada, que al momento de evaluarse imprime 'h', por eso se hace las dos veces. Esto sucede porque en Haskell los valores e interacciones pueden abstraerse de la misma manera, ya que se conserva la transparencia referencial.

2.6.5. La mónada `IO`

El sistema de entrada y salida en Haskell es puramente funcional, y aún así puede llevar a cabo las mismas operaciones que pueden realizarse en un lenguaje de programación convencional. La importancia de este hecho puede notarse en el ejemplo presentado en la Sección 2.6.4, donde se hace la comparación entre StandardML y Haskell. Para lograr esto, Haskell usa la

mónada IO, que es una instancia de la clase `Monad`, para integrar operaciones de entrada y salida en un contexto puramente funcional.

La mónada de entrada y salida usada por Haskell media entre los valores naturales de un lenguaje funcional y las acciones que caracterizan a las operaciones de entrada y salida. Estas acciones deben ser ordenadas de una manera bien definida para que la ejecución del programa tenga sentido. La mónada de entrada y salida en Haskell le brinda al usuario una manera de especificar el encadenamiento secuencial de las acciones, y toda implementación está obligada a preservar ese orden.

Las funciones que proporciona Haskell para llevar a cabo la entrada y salida son:

Funciones de salida. Estas funciones escriben en la salida estándar.

putChar ::**Char** → **IO**()
putStr :: **String** → **IO**()
putStrLn ::**String** → **IO**()
print :: **Show** a ⇒ a → **IO**()

Funciones de entrada. Estas funciones leen de la entrada estándar.

getChar ::**IO Char**
getLine :: **IO String**
getContents ::**IO String**
interact :: (**String** → **String**) → **IO**()
readIO ::**Read** a ⇒ **String** → **IO** a
readLn ::**Read** a ⇒ **IO** a

Archivos. Estas funciones operan en archivos de caracteres. Los archivos son nombrados usando cadenas, las cuales están representadas por el tipo `FilePath`.

```
type FilePath =String
writeFile :: FilePath → String → IO()
appendFile ::FilePath → String → IO()
readFile :: FilePath → IO String
```

Operaciones de secuenciamento de entrada y salida

El constructor de tipos `IO` es una instancia de la clase `Monad`. Las dos funciones monádicas de ligadura (métodos en la clase `Monad`) son usadas para componer operaciones de entrada y salida. La función >> es usada donde el resultado de la primera operación no es de interés, como cuando es (). La operación >>= pasa el resultado de la primera operación como un argumento a la segunda operación.

```
(>>=) :: (Monad m) ⇒ m a → (a → m b) → m b
(>>) :: (Monad m) ⇒ m a → m b → m b
```

Como `IO` es una instancia de la clase `Monad`, tenemos que:

```
(>>=) :: IO a (a → IO b) → IO b
(>>) :: IO a → IO b → IO b
```

Manejo de excepciones con la mónada `IO`

La mónada de entrada y salida incluye un sistema de manejo de errores muy simple. Cualquier operación de entrada y salida puede generar una excepción en lugar de regresar un resultado.

Las excepciones en la mónada `IO` son representadas por valores del tipo `IOError`. Éste es un tipo abstracto: sus constructores son invisibles al usuario. La biblioteca `IO` define funciones que construyen y examinan los valores

`IOError`. La única función del preludio[5] de Haskell que crea un valor `IOError` es `userError`. Estos valores incluyen una cadena que describe al error.

userError :: **String** → **IOError**

Las excepciones son generadas y atrapadas por las siguientes funciones:

ioError :: **IOError** → **IO** a
catch :: **IO** a → (**IOError** → **IO** a) → **IO** a

La función `ioError` genera una excepción; la función `catch` establece un manejador que recibe cualquier excepción protegida por `catch` generada en la acción. Estos manejadores no son selectivos: todas las excepciones son atrapadas. La propagación de excepciones debe ser dada explícitamente en el código del manejador, volviendo a lanzar cualquier excepción no deseada.

2.7. Ventajas

Haskell está basado en un modelo simple que nos permite tanto efectuar la evaluación a mano, alentando tanto la comprensión, como a razonar sobre cómo los programas se comportan. Las definiciones genéricas son introducidas casi sin gastos, en contraste con los lenguajes orientados a objetos como Java. El polimorfismo, junto con las funciones de orden superior, brinda el poder de reusabilidad en los programas escritos en Haskell. Haskell también permite, a través de sus clases de tipos, sobrecargar nombres para representar cosas similares en tipos diferentes.

[5]El preludio de Haskell es un módulo donde se encuentran definidas todas las funciones estándar del lenguaje, en el Capítulo 8 del reporte de Haskell [19] se encuentra la especificación del preludio.

Haskell es un lenguaje elegante, poderoso y de propósito general. Es puro, perezoso y con fuerte sistema de tipos. Gracias a que no permite efectos colaterales, tiene menos errores y son detectados fácilmente. Además, tal como todo lenguaje que intente incrementar la productividad, cuenta con programación modular. La modularidad ayuda a organizar un proyecto de programación a gran escala, facilitando su diseño, desarrollo y mantenimiento.

Además, como lo señala Sebastian Sylvan [21], los programas en Haskell tienen menos errores porque es:

Puro. No hay efectos laterales.

Fuertemente tipificado. No puede haber uso ambiguo de tipos.

Conciso. Los programas son más cortos e intuitivos.

De alto nivel. Los programas en Haskell se leen casi exactamente como la descripción del algoritmo, lo cual hace más fácil verificar que la función hace lo que el algoritmo indica. Al codificar en un nivel de abstracción mayor, dejando los detalles al compilador, hay menor probabilidad de cometer errores.

Tiene manejo automático de memoria. No hay que preocuparse por apuntadores inválidos, el recolector de basura se encarga de ello.

Modular. Haskell ofrece más y mejor "pegamento" para armar el programa partiendo de módulos existentes.

Capítulo 3

Compiladores en Haskell

Los lenguajes funcionales, como Haskell, son muy efectivos en el desarrollo de compiladores. Especialmente durante las últimas etapas de un compilador, ya que la mayoría de los compiladores pueden ser escritos como una serie de transformaciones de representaciones comenzando con el código fuente y terminando con el código destino; cada una de estas representaciones puede ser vista como árboles de estructuras o listas de árboles de estructuras. Precisamente los lenguajes funcionales facilitan el uso de las listas, y es más sencillo convertir una lista a otra, así como acceder a cada elemento de las listas. Gracias a que Haskell cuenta con polimorfismo, es más fácil implementar compiladores de una manera simple; se escriben rutinas pequeñas que pueden aplicarse a distintos tipos de listas.

Actualmente se cuenta con diferentes herramientas para generar analizadores léxicos y sintácticos en Haskell. Al utilizar generadores de analizadores léxicos y sintácticos obtenemos funcionalidad y facilidad de uso, además éstos pueden adaptarse fácilmente a las necesidades de los usuarios. Algunas

de las herramientas existentes para la generación de analizadores léxicos y sintácticos en Haskell son Alex y Happy, respectivamente, también se cuenta con *The Compiler Toolkit*[1].

3.1. Descripción de un compilador

Como señala Aho [1]:

> Un compilador es un programa que toma como entrada un programa escrito en un lenguaje, llamado lenguaje fuente, y lo transforma en un programa escrito en otro lenguaje, llamado lenguaje objeto. Durante este proceso de traducción de un lenguaje a otro, el compilador debe informar al usuario sobre los errores que se detecten en la traducción.

En la compilación hay dos etapas: análisis y síntesis. La parte del análisis divide al programa fuente en sus elementos componentes y crea una representación intermedia del programa fuente. La parte de síntesis construye el programa objeto deseado.

En la compilación, el análisis consta de tres fases:

Análisis léxico, (llamado también análisis lineal) en el que la cadena de caracteres que constituye el programa se lee de izquierda a derecha y se agrupa en componentes léxicos, que son secuencias de caracteres que

[1]The Compiler Toolkit puede adquirirse gratuitamente en http://www.cse.unsw.edu.au/~chak/haskell/ctk/

tienen características comunes, por ejemplo, si son números, si son caracteres especiales o si son instrucciones. Quien diseñe el compilador será el encargado de decidir cuáles secuencias de caracteres se encontrarán en el mismo grupo.

Análisis sintáctico, (llamado también análisis jerárquico) en el que los componentes léxicos se agrupan jerárquicamente en colecciones anidadas con un significado colectivo.

Análisis semántico, en el que se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan de un modo significativo.

La división entre análisis léxico y sintáctico es arbitraria, ésta se hace basándose en las estructuras a analizar. Para el análisis léxico se utilizan expresiones regulares y para el análisis sintáctico se necesitan gramáticas independientes del contexto.

El análisis léxico realizado por autómatas regulares no es suficientemente poderoso para analizar expresiones o proposiciones. Por ejemplo, no se puede distinguir entre `(())` y `(((` y no se pueden emparejar de manera apropiada las palabras `begin` y `end` en proposiciones sin imponer alguna clase de estructura jerárquica a la entrada.

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones. Un componente importante del análisis semántico es la verificación de tipos. En lenguajes con tipos, el

compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente.

Usualmente después del análisis léxico, sintáctico y semántico, el compilador pasa a las fases de generación y optimización de código, las cuales no se tratarán en este trabajo, ya que, por la naturaleza del compilador a realizar, estas etapas no serán necesarias, por lo menos en la manera en que siempre son aplicadas. El compilador a realizar sí generará código destino, pero al no ser código ejecutable por una máquina, todas las técnicas desarrolladas para la generación de código no se pueden aplicar a este compilador.

Una de las principales funciones de un compilador es registrar los diferentes identificadores utilizados en el programa fuente y reunir información sobre cada uno de los atributos de los identificadores. Tales atributos pueden proporcionar información como el espacio que debe reservárseles en memoria, su tipo, su ámbito y, en el caso de procedimientos, se puede saber cuántos argumentos tiene y de qué tipo son, además de la forma en que se pasan los argumentos (por valor, por nombre, etc.) y el tipo del valor devuelto.

Todos los identificadores y sus atributos deben ser almacenados en una estructura que permita recuperarlos fácilmente, dicha estructura es conocida como *tabla de símbolos*. Una *tabla de símbolos* tiene un registro por cada identificador, con campos para sus atributos.

En cualquiera de las fases de un compilador es posible encontrar errores en el programa compilado, y cada vez que un error es encontrado, la fase correspondiente debe darle un tratamiento adecuado a dicho error, para poder continuar con la compilación y detectar, en caso de que existieran, otros errores en el programa fuente. Es posible también que el compilador se detenga con el primer error que encuentra; sin embargo, esto es poco eficiente y

no resulta útil para quien está llevando a cabo la compilación. El sistema de detección de errores debe poder recuperarse y brindar información amplia y útil sobre el error detectado. En la construcción del compilador a considerar en este trabajo, se optó por un sistema de recuperación de errores simple, porque es muy poco probable encontrar un error en los programas que se compilarán, debido a que todos los archivos que serán procesados han sido anteriormente analizados por compiladores que reconocen a IIFTEX.

3.2. Análisis léxico

El analizador léxico es la primera fase de un compilador. El analizador léxico se encarga de leer los caracteres de entrada y entregar como salida una secuencia de componentes léxicos que utilizará el analizador sintáctico para llevar a cabo su análisis en la siguiente fase del compilador. La interacción entre el analizador léxico y el sintáctico suele realizarse convirtiendo al analizador léxico en una subrutina del analizador sintáctico. El analizador sintáctico le pedirá al analizador léxico el *siguiente componente léxico*, y entonces el analizador léxico leerá los caracteres de entrada hasta que puede formar un nuevo componente léxico que le entregará al analizador sintáctico.

Al ser la parte del compilador que lee el texto fuente, el analizador léxico puede encargarse de otras tareas, tales como eliminar del programa fuente los comentarios y espacios en blanco (si éstos no son relevantes para el análisis sintáctico). Otra función del analizador léxico es relacionar cada componente léxico con el número de línea en el que éste se encuentra. Esto podrá ser usado si ocurre un error, entonces al reportarlo al usuario, se le dirá exactamente en qué línea del programa ocurrió el error.

COMPONENTE LÉXICO	LEXEMAS DE EJEMPLO	DESCRIPCIÓN INFORMAL DEL PATRÓN	PATRÓN
inicioAmbiente	{	Llave izquierda	'{'
Math	$	Signo de pesos	'$'
instrucción	\vfill	Diagonal invertida seguida de una cadena	'\' [A-Za-z] [A-Za-z0-9]*

Cuadro 3.1: Ejemplos de componentes léxicos, lexemas y patrones en TEX

3.2.1. Componentes léxicos, patrones y lexemas

Hay un conjuntó de cadenas en la entrada para el cual se produce como salida el mismo componente léxico. Este conjunto de cadenas se describe mediante una regla llamada *patrón* asociada al componente léxico. Se dice que el patrón *concuerda* con cada cadena del conjunto. Un lexema es una secuencia de caracteres en el programa fuente con la que concuerda el patrón para un componente léxico. En el Cuadro 3.1 se muestran algunos ejemplos de componentes léxicos patrones y lexemas en TEX.

3.2.2. Atributos de los componentes léxicos

Cuando un patrón concuerda con más de un lexema, el analizador léxico debe proporcionar información adicional sobre el lexema concreto que concordó con el patrón. Por ejemplo, el patrón **instrucción** concuerda con las cadenas `\vfill` y `\textit`, por eso es importante que el generador de código conozca con exactitud la cadena que concordó con el patrón **instrucción**.

El analizador léxico reúne información sobre los componentes léxicos y sus atributos. Los componentes léxicos influyen en las decisiones del análisis sin-

OPERACIÓN	DEFINICIÓN
unión de L y M, que se escribe $L \cup M$	$L \cup M = \{s \mid s \epsilon L \ o \ s \epsilon M\}$
concatenación de L y M, que se escribe LM	$LM = \{st \mid s \epsilon L \ y \ t \epsilon M\}$
cerradura de Kleene de L, que se escribe L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
cerradura positiva de L, que se escribe L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$
$L^?$	$L^? = L \bigcup \varepsilon$

Cuadro 3.2: Operaciones sobre lenguajes

táctico, y los atributos, en la traducción de los componentes léxicos. Algunas veces no es necesario ningún atributo para llevar a cabo la traducción; por ejemplo, en el caso del componente léxico **inicioAmbiente**, no se necesita información sobre sus atributos, ya que el único lexema que puede concordar con el patrón es {.

3.2.3. Patrones y expresiones regulares

Para definir los patrones que serán reconocidos por el analizador léxico se utilizan expresiones regulares. Como señala Aho [1], una expresión regular se construye a partir de expresiones regulares más simples utilizando un conjunto de reglas que las definen. Cada expresión regular r representa un lenguaje $L(r)$. Entonces también conviene tener presentes las diferentes operaciones que se pueden llevar a cabo sobre los lenguajes, lo cual podemos observar en el Cuadro 3.2, para presentarlo de una manera más sencilla, en el Cuadro 3.2 se utilizará L para referirse a $L(r)$ y M para referirse a $M(r)$, siendo r una expresión regular.

Las reglas de definición especifican cómo se forma $L(r)$ combinando de varias maneras los lenguajes representados por las subexpresiones de r. Las

siguientes reglas definen las expresiones regulares de un alfabeto Σ:

1. ε es una expresión regular que denota $\{\varepsilon\}$, el conjunto que contiene la cadena vacía.
2. Si a es un símbolo de Σ, entonces a es una expresión regular designada por $\{a\}$.
3. Si r y s son expresiones regulares representadas por los lenguajes $L(r)$ y $L(s)$, entonces,
   (a) $(r)|(s)$ es una expresión regular que denota $L(r) \cup L(s)$.
   (b) $(r)(s)$ es una expresión regular que denota $L(r)L(s)$.
   (c) $(r)^*$ es una expresión regular que denota $(L(r))^*$.
   (d) $(r)^+$ es una expresión regular que denota $(L(r))^+$.
   (e) (r) es una expresión regular que denota $L(r)$.

3.2.4. Errores léxicos

En la fase de análisis léxico son muy pocos los errores que se pueden detectar porque el analizador léxico tiene una visión muy limitada del programa fuente ya que las estructuras que puede reconocer no son tan complejas como las estructuras que pueden detectarse durante el análisis sintáctico. Si el analizador léxico no puede continuar debido a que ninguno de los patrones concuerda con un prefijo del resto de la entrada, el analizador debe recuperarse del error y seguir revisando la entrada.

El modo de recuperación de errores más sencillo de implementar es el conocido como recuperación en modo de pánico, éste consiste en desechar caracteres

sucesivos de la entrada hasta que el analizador puede reconocer un componente léxico. Sin embargo esta técnica de recuperación puede confundir en ocasiones al analizador sintáctico. Otras posibles acciones de recuperación de errores son borrar de manera especulativa un carácter extraño, insertar un carácter que falta para poder emparejar la entrada con algún patrón existente, reemplazar –también especulativamente– un carácter incorrecto por uno correcto o intercambiar dos caracteres adyacentes.

3.3. Alex: un generador de analizadores léxicos en Haskell

Alex es una herramienta que facilita la construcción de analizadores léxicos en Haskell. Alex recibe una descripción de los componentes léxicos a analizar, por medio de expresiones regulares. Después de habérsele proporcionado a Alex la expresión regular, éste genera un módulo de Haskell que contiene el código para analizar texto.

3.3.1. Sintaxis de un archivo de Alex

Las expresiones regulares deben dársele a Alex en un archivo. El archivo comienza y termina con fragmentos opcionales de código. Estos fragmentos de código son copiados en el archivo generado por Alex.

En la parte inicial del archivo, el fragmento de código es usado normalmente para declarar el nombre que se le dará al módulo generado e indicar los módulos a importar. Aquí no deben declararse funciones o tipos, ya que Alex

puede necesitar indicar otros módulos a importar, y esto lo hace incluyéndolos directamente debajo del código opcional. Si en esta parte se incluyeran declaraciones de funciones o tipos ocurriría un error al compilar el código generado, ya se violaría la forma en que deben estar estructurados los módulos en Haskell [19]. El principio de un archivo de Alex puede verse de la siguiente manera:

```
{
module Lexico (alexScanTokens) where
import Definiciones
import Tipos
}
```

Después puede especificarse, de manera opcional, la "envoltura" a utilizar. Las "envolturas" de Alex proporcionan más funcionalidad al analizador léxico generado, ya que contienen funciones específicas para controlar otros aspectos del analizador, tales como llevar la cuenta de los números de línea, regresar toda una lista de componentes léxicos y no sólo el primero que encuentra, así como permitir el manejo de mónadas. Al especificar la "envoltura" que se utilizará, es necesario que esté precedida por el símbolo %[2]. A pesar de que más adelante será tratado con más detalle el tema de las "envolturas", a continuación se encuentra cómo especificar que la envoltura a utilizar será la envoltura `monad`.

```
% wrapper "monad"
```

A continuación la especificación del analizador puede contener definiciones de macros. Existen dos tipos distintos de macros: (a) las macros de un conjunto de caracteres, que empiezan con $; y (b) las macros de expresiones regulares, que empiezan con @. Éstas también serán explicadas más adelante. La

[2]Como se verá más adelante, éste símbolo tiene un significado especial para Alex

presentación del ejemplo de la definición de macros será dado más adelante, cuando se cuenten con las bases suficientes para comprender el manejo de macros en Alex.

En seguida se encuentran las reglas, éstas son de la forma *id* : −DEFINICIÓN DE LA REGLA, donde *id* es el nombre de la regla, el cual sólo es usado para propósitos de documentación. Las reglas le indican a Alex cuáles componentes léxicos serán aceptados y qué debe hacerse cuando se encuentre cada uno de ellos.

Cada regla define un componente léxico en la especificación léxica. Cuando parte de la entrada concuerde con la expresión regular de alguna regla, el analizador léxico de Alex regresará el valor de la expresión del lado derecho, al cual se le conoce como acción. La acción puede ser cualquier expresión de Haskell, la única restricción es que todas las acciones deberán ser del mismo tipo. También puede ser que la acción no esté definida, en cuyo caso el componente léxico será ignorado, esto puede ser útil en lenguajes en los que los espacios en blanco no sean importantes para fases posteriores de la compilación. Lo anterior puede ser visto en el siguiente ejemplo de reglas en Alex:

```
tokens :−

  "−−".*                          ;
  let                             { λs . Let }
  in                              { λs . In }
  [\=\+\-\*\/\(\)]                { λs . Sym (head s) }
```

También es posible asignarles códigos de inicio a las reglas. Esto es para añadir estados a la especificación léxica, de tal forma que sólo ciertas reglas puedan ser usadas para un estado dado. Un código de inicio es simplemente

un identificador, o el código de inicio especial '0'. A cada regla puede dársele una lista de códigos de inicio en los cuales es aplicable. Cuando se le solicita al analizador léxico que revise el siguiente componente léxico de la entrada, el código de inicio a usar también es especificado. Sólo las reglas que usan los códigos de inicio son habilitadas.

3.3.2. Expresiones regulares

Las expresiones regulares son los patrones que Alex utiliza para emparejar componentes léxicos en el flujo de entrada.

A continuación se especifica la sintaxis de las expresiones regulares en Alex.:

```
regexp := rexp2 "|" rexp2
        | rexp2
rexp2 := rexp1 rexp1
        | rexp1
rexp1 := rexp0 optional
optional := *
        | +
        | ?
        | repeat
        | ε
rexp0 := set | rmac | string
        | "(" regexp ")"
        | "(" ")"
repeat := "{" digito "}"
        | "{" digito "," "}"
        | "{" digito, digito "}"
```

La sintaxis de las expresiones regulares es bastante estándar, la única diferencia es que a las expresiones regulares en Alex se les permite la secuencia () para denotar la expresión regular que puede concordar con la cadena vacía (ϵ).

Los espacios son ignorados en una expresión regular. Los espacios en blanco literales pueden ser incluidos poniéndoles comillas " ", o poniendo una \ antes de cada espacio. A continuación se presentan algunos ejemplos de expresiones regulares en Alex:

"begin"	*Coincide con la cadena begin*
[0−9]*	*Coincide con cero o más dígitos*
[a−z]+	*Coincide con una o más letras minúsculas*
[A−Z]{3}	*Coincide con tres letras mayúsculas*

A continuación presento las construcciones que se pueden utilizar en Alex para especificar una expresión regular:

conjunto Concuerda con cualquiera de los caracteres en el conjunto. Más adelante se detallará la sintaxis de los conjuntos en Alex.

@foo Expande la definición del macro de la expresión regular correspondiente

"..." Concuerda con la secuencia de caracteres de la cadena, en ese orden.

r* Concuerda con cero o más ocurrencias de `r`.

r+ Concuerda con una o más ocurrencias de `r`.

r? Concuerda con cero o una ocurrencia de `r`.

r{n} Concuerda con n ocurrencias de `r`.

r{n,} Concuerda con n o más ocurrencias de `r`.

r{n,m} Concuerda con entre n y m ocurrencias de `r`.

3.3.3. Sintaxis de los conjuntos de caracteres

Los conjuntos de caracteres son elementos fundamentales en una expresión regular. Un conjunto de caracteres es un patrón que concuerda con un solo carácter de los contenidos en el conjunto.

Las distintas construcciones de conjuntos de caracteres aceptadas por Alex son:

carácter El conjunto de caracteres más simple es aquél que sólo contiene un carácter. Sin embargo, existen caracteres especiales (como "[" y ".") que deben estar precedidos por el carácter de escape (\). Algunos caracteres no imprimibles tienen también secuencias especiales para poder ser representados. Tal es el caso de \a, \b, \f, \n, \r, \t y \v. Otros caracteres pueden ser representados usando su valor numérico. Los espacios en blanco son ignorados, para representarlos es necesario que estén precedidos por "\".

carácter-carácter Un rango de caracteres puede ser expresado separando dichos caracteres con un −, todos los caracteres con códigos en el rango dado serán incluidos en el conjunto.

- Concuerda con todos los caracteres, excepto con el de línea nueva (\n).

conjunto0 # conjunto1 Concuerda con todos caracteres en el *conjunto0* que no estén en el *conjunto1*.

[conjuntos] Es la unión de los *conjuntos*.

[^conjuntos] Es el complemento de la unión de los *conjuntos*. Es equivalente a `.#[conjuntos]`.

~conjunto Es el complemento de *conjunto*. Es equivalente a `. # conjunto`.

Un macro de conjuntos es escrito como $ seguido de un identificador. Hay algunos macros de conjuntos de caracteres predefinidas en Alex:

$white Concuerda con todos los espacios en blanco, incluyendo una línea nueva.

$printable Concuerda con todos los caracteres imprimibles (caracteres del 32 al 126 en ASCII)

Los macros de conjuntos de caracteres pueden ser definidos al principio del archivo junto con los macros de expresiones regulares.

3.3.4. Interfaz de un analizador léxico generado por Alex

Alex provee una interfaz básica para el analizador léxico generado, la cual puede ser usada para analizar componentes léxicos dado un tipo abstracto de entrada con operaciones sobre el mismo. También se tiene la opción de incluir una "envoltura", la cual provee de abstracción a alto nivel.

Interfaz básica

Si un archivo de Alex es compilado sin la declaración de una "envoltura", entonces se tendrá acceso a la capa de más bajo nivel del analizador léxico. En este caso deben proveerse definiciones para lo siguiente, ya sea en el mismo módulo o en otro módulo que sea importado.

```
type AlexInput = ...
alexGetChar :: AlexInput → Maybe (Char,AlexInput)
alexInputPrevChar :: AlexInput → Char
```

El analizador léxico generado es independiente del tipo de entrada, razón por la cual el usuario debe proveer la definición del tipo de entrada. El tipo de entrada debe llevar un registro del carácter previo en la entrada, esto es usado para implementar los patrones con un contexto izquierdo. La única función que Alex proveerá será `alexScan`.

Al llamar a `alexScan`, éste analizará un solo componente léxico del flujo de entrada y regresará la entrada restante, el tamaño del componente léxico y la acción. La acción es simplemente el valor de la expresión dentro de $\{\ldots\}$ en el lado derecho de la regla correspondiente del archivo de Alex.

Una vez que se tiene la acción, depende del usuario decidir qué debe hacerse con ella. El tipo de la acción podría ser una función que toma la representación como cadena de un componente léxico y regresa un valor, o podría ser una continuación que toma la nueva entrada y llama a `alexScan` nuevamente, construyendo de esta manera una lista de componentes léxicos.

3.3.5. "Envolturas"

Para obtener mayor funcionalidad, puede hacerse uso de alguna de las envolturas que Alex proporciona. Para poder usar alguna éstas, debe incluirse la siguiente declaración en el archivo de Alex:

```
%wrapper "nombre"
```

donde `nombre` es el nombre de una de las envolturas. A continuación se describe cada una de las envolturas.

La envoltura "basic"

La envoltura `basic` es una buena manera de obtener una función del tipo `String → [Token]` partiendo de la especificación de un analizador léxico. Provee definiciones para `AlexInput`, `alexGetChar`, y `alexInputPrevChar` que son adecuadas para analizar una entrada. También provee la función `alexScanTokens`, la cual toma una cadena como entrada y regresa la lista de componentes léxicos que ésta contiene. Esta envoltura no provee ningún tipo de soporte para usar los códigos de inicio.

El usuario debe proporcionar la definición del tipo `Token`. Todas las acciones en la especificación léxica deberán tener el tipo:

```
{...} :: String → Token
```

para algún tipo `Token`.

La envoltura "posn"

La envoltura `posn` provee más funcionalidad que la envoltura basic: lleva el registro del número de línea y columna de los componentes léxicos de entrada. La envoltura posn provee las definiciones para `data AlexPosn`, `type AlexInput`, `AlexSkip` y `alexScanTokens`, además de `alexGetChar` y `alexInputPrevChar`:

Los tipos de las acciones deberán ser:

```
{ ...} :: AlexPosn → String → Token
```

La envoltura "monad"

La envoltura `monad` es la más flexible de las envolturas que incluye Alex. Incluye una mónada de estados, que lleva registro de la entrada actual y la posición del texto, además del estado inicial. El objetivo de esta envoltura es proporcionar las bases para que el usuario pueda construir sus propias mónadas.

Las funciones que se incluyen en el analizador léxico cuando se elige la envoltura monad son las siguientes:

```
data AlexState =AlexState {
        alex_pos :: !AlexPosn,  posición en la entrada actual
        alex_inp :: String,     entrada actual
        alex_chr :: !Char,      carácter anterior a la entrada
        alex_scd :: !Int        código de inicio actual
    }
```

```
newtype Alex α=Alex { unAlex ::AlexState
                                → Either String (AlexState, α) }

runAlex          :: String → Alex α → Either String α

alexGetInput     :: Alex AlexInput
alexSetInput     :: AlexInput → Alex ()

alexError        :: String → Alex α

alexGetStartCode :: Alex Int
alexSetStartCode :: Int → Alex ()
```

Para invocar al analizador que usa la envoltura monad, debe usarse la función:

```
alexMonadScan  :: Alex result
```

Las acciones deberán tener el tipo:

```
type AlexAction result = AlexInput → Int → Alex result
{...} ::AlexAction result
```

La envoltura monad también proporciona algunos combinadores muy útiles para construir acciones:

```
skip :: AlexAction result
skip input len = alexMonadScan

andBegin :: AlexAction result → Int → AlexAction result
(action `andBegin` code) input len = do alexSetStartCode code; action
    input len

begin :: Int → AlexAction result
```

```
begin code = skip 'andBegin' code

token :: (String → Int → token) → AlexAction token
token t input len = return (t input len)
```

La envoltura "gscan"

La envoltura gscan se incluye principalmente por razones históricas, ésta proporciona una interfaz muy similar a la proporcionada en versiones anteriores de Alex. Dicha interfaz es muy general, permite que las acciones modifiquen el código de inicio y pasen valores de estados arbitrarios.

3.4. Análisis sintáctico

Las reglas que definen la estructura sintáctica de programas bien formados se pueden describir por medio de gramáticas independientes del contexto usando notación BNF (Backus Naur Form), de esta manera se obtiene una especificación sintáctica precisa, la cual puede ser utilizada para construir el analizador sintáctico de manera casi inmediata gracias a las herramientas generadoras de analizadores sintácticos, como lo es Happy[3].

3.4.1. Gramáticas independientes del contexto

Una gramática independiente del contexto consta de símbolos terminales, no-terminales, un símbolo inicial y producciones. Es decir, podemos ver una

[3]Happy puede ser obtenido en www.haskell.org/happy

gramática como una tupla $< S, NT, T, P >$ y $S \epsilon NT$, donde S es el símbolo inicial, NT es el conjunto que contiene a todos los símbolos no-terminales, T es el conjunto que contiene a todos los símbolos terminales y P es el conjunto de producciones. Los símbolos terminales son los símbolos básicos con los que se forman las cadenas, son los componentes léxicos que fueron reconocidos durante el análisis léxico. Los símbolos no-terminales son variables sintácticas que sirven para representar conjuntos de cadenas, y de esta manera definir el lenguaje generado por la gramática, así como imponer una estructura jerárquica en la misma. Existe un símbolo no-terminal especial que es considerado como símbolo inicial de la gramática, y el conjunto de cadenas que representa es el lenguaje definido por la gramática. Las producciones determinan cómo se pueden combinar los terminales y los no terminales para formar cadenas pertenecientes a la gramática.

Gramáticas ambiguas

Una gramática es ambigua si produce más de un árbol de análisis sintáctico para alguna frase. Es decir, dicha gramática produce más de una derivación por la izquierda o por la derecha para la misma frase. Afortunadamente existen técnicas para eliminar la ambigüedad de algunas gramáticas, haciendo uso de la precedencia y asociatividad de sus operadores.

3.4.2. Funciones del analizador sintáctico

Por lo general el analizador sintáctico recibe una cadena de componentes léxicos del analizador léxico y analiza dicha cadena para ver si puede ser generada por la gramática del lenguaje fuente. En algunas ocasiones, cuando los componentes léxicos son cadenas sencillas, el analizador sintáctico lleva

a cabo también la función del analizador léxico, por lo que no recibe como entrada una cadena de componentes léxicos generada por el analizador léxico, sino la cadena a analizar, sin que haya sido procesada por un analizador léxico. Al igual que el analizador léxico, el sintáctico deberá informar sobre cualquier error de sintaxis de una manera clara, así como ser capaz de recuperarse de un error rápidamente para poder continuar procesando el resto de su entrada.

Existen tres tipos generales de analizadores sintácticos para gramáticas: los universales, los descendentes y los ascendentes. Los métodos universales de análisis sintáctico pueden analizar cualquier gramática, sin embargo, estos métodos son demasiado ineficientes para usarlos en la producción de compiladores, para más información sobre el tema puede consultarse a Younger [30] y Earley [5]. Los métodos de análisis sintáctico empleados generalmente en los compiladores son descendentes o ascendentes.

Los analizadores sintácticos descendentes construyen árboles de análisis sintáctico desde arriba hacia abajo, y los analizadores sintácticos ascendentes lo hacen de abajo hacia arriba, examinando la entrada al analizador sintáctico de izquierda a derecha. Debido a las restricciones que deben imponerse a los métodos ascendentes y descendentes más eficientes para poder lograr una reducción en sus estados, éstos trabajan sólo con subclases de gramáticas, pero varias de estas subclases, como las gramáticas LL[4] y LR[5] son lo suficiente-

[4]Se dice que una gramática es LL cuando el análisis de la entrada es de izquierda a derecha (por eso la primera L, por left) y con una derivación por la izquierda (por eso la segunda L). En las gramáticas LL(k) debe poder reconocerse el uso de una producción viendo sólo los primeros k símbolos de los que se deriva su lado derecho. Estas gramáticas son usadas en métodos descendentes.

[5]Una gramática LR es una gramática para la que se puede construir una tabla de análisis sintáctico sin ambigüedades, es decir, para las que puede construirse un analizador sintáctico por desplazamiento y reducción. Estas gramáticas son usadas en métodos

mente expresivas para describir la mayoría de los lenguajes de programación.

En teoría, la salida de un analizador sintáctico es una representación del árbol de análisis sintáctico para la cadena de componentes léxicos que el analizador sintáctico recibió del analizador léxico. Pero en un analizador sintáctico de una sola pasada se llevan a cabo otras tareas durante el análisis sintáctico como llevar a cabo la revisión de tipos y otras clases de análisis semántico, además de generar código intermedio, el cual, como ya se mencionó anteriormente, no será necesario para el tipo de compilador a desarrollar.

3.4.3. Errores sintácticos

Al igual que en la fase de análisis léxico, durante el análisis sintáctico pueden presentarse errores, y éstos deben ser reportados de manera adecuada al usuario. Al detectar un error, el analizador sintáctico debe de informar al usuario de manera clara la existencia del error, pero no debe detenerse. Debe recuperarse del error y seguir analizando la entrada.

El manejador de errores debe al menos informar el lugar del programa fuente en donde el error fue detectado, ya que es muy probable que en realidad el error haya ocurrido en alguno de los componentes léxicos anteriores. También debe recuperarse de los errores porque el procesamiento posterior de la entrada podría revelar más errores.

Las estrategias de recuperación de errores comúnmente utilizadas son[1]:

Recuperación en modo de pánico. Desechar componentes léxicos de la entrada hasta encontrar uno perteneciente a un conjunto designado de

ascendentes.

componentes léxicos de sincronización. Estos componentes léxicos de sincronización son usualmente delimitadores como el punto y coma.

Recuperación a nivel de frase. Sustituir un prefijo de la entrada que resta por alguna cadena que le permita continuar al analizador. Sin embargo, se debe ser especialmente cuidadoso al insertar nuevas cadenas para corregir errores, ya que pueden ocasionar ciclos infinitos durante el análisis.

Producciones de error. Esta opción es viable si se conocen de antemano los errores más comúnmente realizados por los programadores del lenguaje fuente. Pueden agregarse producciones que generen las construcciones erróneas. Así puede usarse esta gramática aumentada para construir el analizador sintáctico, de manera que si usa una producción de error, se puedan generar diagnósticos de error apropiados para indicar la construcción errónea reconocida en la entrada.

Corrección global. Utilizar algoritmos para elegir una secuencia mínima de cambios para obtener una corrección global del programa con menor costo, es decir, con el menor número de transformaciones posible. Desafortunadamente la implementación de estos algoritmos es extremadamente costosa en términos de espacio y tiempo, por lo que esta técnica no es muy utilizada actualmente [1].

3.5. Happy: un generador de analizadores sintácticos en Haskell

Happy es un generador de analizadores sintácticos en Haskell. Toma un archivo que contiene la especificación en BNF de una gramática y produce un

módulo en Haskell que contiene un analizador sintáctico para dicha gramática.

Happy puede trabajar en conjunto con un analizador léxico proporcionado por el usuario o puede también analizar directamente un flujo de caracteres, aunque esto último no es recomendable debido a que el desempeño del analizador puede verse afectado.

De acuerdo a [17], los analizadores sintácticos generados por Happy son rápidos, generalmente más rápidos que su equivalente escrito con combinadores o herramientas similares. Happy es lo suficientemente poderoso como para analizar al mismo Haskell, de hecho existe un analizador sintáctico de Haskell escrito usando Happy, el cual puede ser obtenido de manera gratuita en internet.

La idea básica del funcionamiento de Happy puede ser descrita en tres pasos: (1) definir la gramática que se desea analizar en un archivo; (2) correr Happy sobre esa gramática para generar un módulo que pueda ser compilado en Haskell; y (3) usar dicho módulo como parte de un programa en Haskell.

Happy puede generar cuatro tipos de analizadores sintácticos partiendo de una gramática dada, esto es para permitir que los usuarios experimenten con diferentes formas de código funcional para conocer cuál es la que mejor se adapta a sus necesidades. De esta manera, las personas que escriban un compilador pueden usar diferentes tipos de analizadores para adaptarlos a sus compiladores. Los tipos de analizadores sintácticos que Happy puede generar son:

'standard' Haskell 98. Funciona con cualquier compilador que compile Haskell 98.

standard Haskell con arreglos. Ésta no es la opción por omisión porque se ha encontrado que éste genera analizadores más lentos que el anterior.

Haskell con extensiones GHC[6]. Ésta es una opción un poco más rápida que el 'standard' y sólo debe suarse si el código generado será compilado con GHC.

Haskell GHC con arreglos codificados como cadenas. Ésta es la opción más rápida y pequeña para los usuarios de GHC.

Happy también puede generar analizadores que muestren información pertinente para corregir errores, mostrando las transiciones de los estados y los componentes léxicos de entrada.

3.5.1. Estructura básica de un archivo de Happy

Al principio del archivo se encuentra un encabezado opcional de módulo, esto es simplemente un encabezado común de Haskell, rodeado por llaves. Este código es copiado en el módulo generado por Happy para que el usuario pueda incluir código en Haskell al principio del módulo generado, este código es por lo general para definir algunas directivas de importación. Por ejemplo:

```
{
module Parser where
import Lexer
import Tabla
}
```

[4] GHC es un compilador para Haskell 98, el cual acepta muchas extensiones del lenguaje, tales como concurrencia, excepciones, y extensiones al sistema de tipos

A continuación se declara el nombre de la función que Happy generará para llevar a cabo el análisis. Después se declara el tipo de los componentes léxicos que el analizador aceptará. Para indicar que el nombre de la función generada por Alex será `calc` y que el tipo de los componentes léxicos será `Token`, es necesario hacerlo de la siguiente manera:

```
%name calc
%tokentype { Token }
```

El analizador sintáctico será del tipo `[Token]` $\rightarrow \alpha$, donde α es el tipo del valor devuelto por el analizador sintáctico, determinado por las reglas que se definirán más abajo en el archivo.

Después se declaran todos los tipos diferentes de componentes léxicos que Happy podría recibir. En esta parte se señala tanto la manera en que el usuario se referirá a los componentes léxicos a lo largo de toda la especificación de la gramática, como el nombre del patrón que concuerda con dicho componente léxico. El analizador sintáctico esperará recibir un flujo de componentes léxicos, cada uno de los cuales deberá concordar con alguno de los patrones dados. Usualmente el valor del componente léxico es el componente léxico mismo, pero al usar el símbolo $$, se puede especificar que un atributo del componente léxico será el valor real, como puede ocurrir en caso de que varios lexemas concuerden con un mismo patrón de cierto componente léxico. Un ejemplo de la especificación de componentes léxicos en Happy es el siguiente:

```
%token
      let             { TokenLet }
      in              { TokenIn }
      int             { TokenInt $$ }
      var             { TokenVar $$ }
```

```
'='     { TokenEq }
'+'     { TokenPlus }
'-'     { TokenMinus }
'*'     { TokenTimes }
'/'     { TokenDiv }
'('     { TokenOB }
')'     { TokenCB }
```

En el ejemplo anterior se puede observar que si tenemos `TokenInt 3`, el valor del componente léxico `int` será 3.

Después de especificar los componentes léxicos, se deben incluir las producciones de la gramática, precedidas por el símbolo %%. Cada una de estas producciones consiste en un símbolo no terminal del lado izquierdo, seguido de dos puntos (:), seguido de una o más expansiones separadas por |. Cada una de las expansiones tiene asociado un código en Haskell, rodeado por llaves. Siguiendo con el ejemplo anterior, los componentes léxicos se especifican de la siguiente manera:

```
%%
Exp  : let var '=' Exp in Exp { Let $2 $4 $6 }
     | Exp1                   { Exp1 $1 }

Exp1 : Exp1 '+' Term          { Plus $1 $3 }
     | Exp1 '-' Term          { Minus $1 $3 }
     | Term                   { Term $1 }

Term : Term '×'Factor         { Times $1 $3 }
     | Term '/' Factor        { Div $1 $3 }
     | Factor                 { Factor $1 }
```

```
Factor
    : int                   { Int $1 }
    | var                   { Var $1 }
    | '(' Exp ')'           { Brack $2 }
```

En los analizadores sintácticos generados por Happy, cada símbolo, terminal o no terminal, tiene un valor. Se definen los valores de los componentes léxicos, y la gramática define los valores de los símbolos no terminales en términos de secuencias de otros símbolos (ya sean componentes léxicos o no terminales). En una producción como:

n : t_1 ... t_n { E }

cuando el analizador encuentra los símbolos t_1 ... t_n en el flujo de componentes léxicos, construye el símbolo n y le da el valor E, el cual puede hacer referencia a los valores t_1 ... t_n usando los símbolos $1 ... $n. El analizador reduce la entrada usando las reglas de la gramática hasta que sólo quede un símbolo: el primer símbolo definido en la gramática. El valor de este símbolo es el valor devuelto por el analizador. Al final puede agregarse opcionalmente un sección más de código en Haskell, rodeada por llaves. En esta sección de código opcional se pueden declarar los tipos de datos que representan las expresiones analizadas, así como la estructura de datos que se utilizará para los componentes léxicos. También puede definirse en esta sección la función `happyError`, aunque ésta no debe estar forzosamente declarada en *esa* sección de código opcional, ya que puede también ser importada de algún módulo auxiliar que se construya, lo que es obligatorio es que se tenga acceso a esa función.

3.5.2. Análisis de secuencias en Happy

En Happy se pueden definir secuencias usando producciones recursivas como la siguiente:

```
prods   : prod { [$1] }
        | prods prod { $2:$1 }
```

Vale la pena resaltar que la definición anterior fue realizada utilizando recursión izquierda, con recursión derecha la definición quedaría de la siguiente manera:

```
prods   : prod { [$1] }
        | prod prods { $1:$2 }
```

La razón por la cual se utilizó recursión izquierda para llevar a cabo dicha definición, es que Happy trabaja de manera más eficiente al analizar reglas recursivas por la izquierda, esto ocurre porque los analizadores sintácticos que genera son ascendentes para gramáticas LALR[7]. El resultado de este proceso es un analizador sintáctico con una pila de tamaño constante. Si se utilizaran reglas recursivas por la derecha, se necesitaría un espacio para la pila proporcional al tamaño de la lista que va a ser analizada. Esto es de suma importancia si consideramos el caso en que secuencias muy largas requieren ser analizadas, en cuyo caso el código generado para reconocerlas también es muy largo.

[7]El término LALR viene del inglés lookahead-LR, que quiere decir análisis sintáctico LR con símbolo de anticipación. Los analizadores para las gramáticas LALR son más pequeños que aquélos para las gramáticas LR

3.5.3. Precedencias

Una forma de eliminar la ambigüedad de una gramática es definir precedencias entre los operadores para eliminar dicha ambigüedad.

Las instrucciones `%left` y `%right` seguidas de una lista de terminales, declaran que dichos componentes léxicos son asociativos por la izquierda y por la derecha, respectivamente. La precedencia de estos componentes léxicos con respecto a otros componentes léxicos es establecida por el orden de las instrucciones `%left` y `%right`: mientras más arriba estén en la lista, menos precedencia tienen. Una precedencia más alta hace que un operador esté ligado más estrechamente.

Si dos operadores tienen la misma precedencia, la asociatividad tiene la última palabra. Si los operadores son asociativos por la izquierda, entonces expresiones como $2 \oplus 8 \otimes 9$ serán analizadas como $(2 \oplus 8) \otimes 9$, y serán analizadas como $2 \oplus (8 \otimes 9)$ si son asociativos por la derecha. Happy también cuenta con la instrucción `%nonassoc`, la cual sirve para indicar que los operadores señalados no pueden ser usados juntos, el ejemplo más claro es el caso de los operadores lógicos '<' y '>'.

No sólo los símbolos terminales pueden tener precedencia, una regla en la gramática también puede tenerla. Si el último terminal en el lado derecho de la regla tiene asociada una precedencia, entonces ésa es la precedencia de la regla entera.

Las precedencias son usadas para resolver ambigüedades en la gramática. Si existe un conflicto de reducción/desplazamiento, entonces la precedencia de la regla y la del componente léxico de predicción son examinadas para resolver el conflicto.

- Si la precedencia de la regla es mayor, entonces el conflicto se resuelve optando por la reducción
- Si la precedencia del componente léxico de predicción es mayor, entonces el conflicto es resuelto optando por el desplazamiento.
- Si las precedencias son iguales, entonces
  - Si el componente léxico es asociativo por la izquierda, se opta por la reducción.
  - Si el componente léxico es asociativo por la derecha, se opta por el desplazamiento.
  - Si el componente léxico no es asociativo, entonces se produce un error.
- Si la regla o el componente léxico no tienen precedencia, entonces se opta por el desplazamiento.

Happy permite al usuario usar signaturas de tipos en el archivo que contiene la gramática para indicar el tipo de cada producción. Como se menciona en el manual de Happy [17], entre las ventajas de esta característica se encuentran:

- Incluir los tipos en la gramática ayuda a documentarla para que pueda ser comprendida fácilmente por alguien que lee el código.
- Arreglar los errores de tipos en el módulo generado puede volverse más fácil si Happy inserta las signaturas de tipo.
- Las signaturas de tipo ayudan al compilador de Haskell a compilar el analizador sintáctico más rápido.

3.5.4. Uso de mónadas en Happy

Happy brinda la posibilidad de utilizar mónadas en el analizador sintáctico generado. Esto puede ser útil por múltiples razones:

- Manejar errores ocurridos durante el análisis usando una mónada para las excepciones.
- Llevar la cuenta de los números de línea en el archivo de entrada.
- Llevar a cabo operaciones de entrada y salida durante el análisis.
- Utilizar estados en el analizador, esto es útil para analizar lenguajes dependientes del contexto.

Agregar soporte monádico al analizador sintáctico es muy sencillo con Happy. Lo único que se requiere es incluir la siguiente directiva en la sección de declaraciones del archivo que contiene la gramática.

```
monad { <type> } [ { <then> } { <return> } ]
```

donde `<type>` es el constructor de tipos para la mónada, `<then>` es la operación de ligadura de la mónada, y `<return>` es la operación de regreso. Si no se incluyen los nombres de las operaciones de ligadura y regreso, Happy asume que `<type>` es una instancia de la clase estándar `Monad` y usa una sobrecarga de nombres para las operaciones de ligadura y regreso (`>>=` y `return`).

Cuando la declaración de mónada es incluida en la gramática, Happy hace algunos cambios al analizador generado: los tipos de la función principal

del analizador y de la función `happyError` serán `Token`→`M` α, donde `M` es el constructor de tipos de la mónada, y la función debe ser polimórfica en α. En otras palabras, Happy agrega una aplicación a la operación `<return>` definida en la declaración.

La mayoría del tiempo, no se desea que una producción tenga este tipo: se tendrían que escribir muchos `returnP` en todas ellas. Pero, en algunas reglas de la gramática puede sí necesitarse el uso de mónadas, por eso Happy tiene una sintaxis especial para las acciones monádicas:

n : t_1 ... t_n { %<expr> }

El **%** en la acción indica que ésa es una acción monádica, con tipo `P a`, donde `a` es el tipo de regreso real de la producción. Cuando Happy reduce una de estas reglas, evalúa la expresión

<expr> **'then'** λ result . <sigue el análisis>

Happy usa `result` como el valor semántico real de la producción. Durante el análisis, varias acciones monádicas pueden ser reducidas, con lo cual se obtiene una secuencia como

<$expr_1$> **'then'** λr_1 . <$expr_2$> **'then'** λr_2 .... **return** <$expr_3$>

Las acciones monádicas son realizadas en el orden en el que son reducidas. Si se considera el análisis sintáctico como un árbol, entonces las reducciones suceden en el mismo orden que un recorrido a profundidad de izquierda a derecha.

3.6. Análisis semántico

El análisis semántico revisa el programa fuente para encontrar errores semánticos y reúne información sobre los tipos para su posterior uso durante la generación de código. La comprobación de tipos, la comprobación de flujo de control, la comprobación de unicidad y las comprobaciones relacionadas con nombres son algunas de las tareas de las que debe encargarse el análisis semántico.

El tipo de una construcción de un lenguaje se denota mediante una *expresión de tipo*, ésta puede ser un tipo básico o formarse aplicando un operador llamado *constructor de tipos* a otras expresiones de tipos. En muchos lenguajes, los tipos básicos son boolean, char, integer y real, hay también un tipo básico especial llamado *error_tipo*, que sirve para señalar errores durante la comprobación de tipos.

Un sistema de tipos consta de una serie de reglas para asignar expresiones de tipos a las distintas partes de un programa. Un comprobador de tipos implementa un sistema de tipos y puede especificarse basándose en la sintaxis especificada para el lenguaje fuente. Ésta es la razón por la cual la comprobación de tipos puede hacerse a la par del análisis sintáctico.

Al igual que en las etapas anteriores de análisis, es importante que el analizador sintáctico pueda recuperarse si encuentra un error, para poder seguir analizando en búsqueda de más errores, así como reportarlos oportuna y claramente. Es por esto que se incluye el tipo básico *error_tipo*.

Capítulo 4

TEX

4.1. Descripción

TEX es un programa para dar formato a textos, especialmente pensado para la creación de libros con alto contenido matemático. De hecho, TEX es el programa más poderoso para producir documentos científicos y técnicos de alta calidad. El texto que se le da como entrada es procesado para darle un formato adecuado, tal como poner líneas del mismo largo y páginas de un tamaño dado. Sin embargo, las capacidades de TEX van más allá de un programa para dar formato a textos; TEX es también un lenguaje de programación poderoso que permite al programador agregarle.

TEX sólo entiende algunas instrucciones básicas, suficientes para editar textos simples, pero permite que sean definidos nuevos comandos complejos y de alto nivel, basados en las instrucciones básicas.

Cuando TEX se ejecuta, lee un archivo llamado *archivo de formato*, el cual contiene las definiciones de los comandos de alto nivel, este archivo también

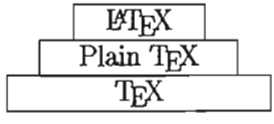
contiene los patrones para la división de palabras con guiones. Una vez hecho esto, comenzará a leer el archivo fuente, es decir, el archivo que contiene el texto a procesar junto con los comandos de formato.

Al escribir un manuscrito en TEX, se le está diciendo a la computadora con exactitud cómo será transformado dicho manuscrito a páginas cuya calidad tipográfica sea comparable a la de las mejores imprentas del mundo.

Editar textos sólo con TEX puede ser un poco laborioso. Es por eso que el creador de TEX, Donald E. Knuth ha creado un formato básico llamado *Plain TEX* [11], el cual interactúa con TEX a un nivel más simple. Pero Plain TEX sigue siendo difícil de utilizar debido a que para explotar todo su poder, se necesita tener un amplio conocimiento de las técnicas de programación de TEX, lo cual limita su uso a programadores expertos.

Por esa razón Leslie Lamport creó el formato LATEX [13], el cual provee comandos de alto nivel para la edición de textos más complejos. Con LATEX incluso los usuarios con poca o nula experiencia en programación pueden crear textos bastante complejos en poco tiempo, utilizando y explotando todas las características que ofrece TEX.

Como puede apreciarse en el siguiente diagrama, Plain TEX está basado en TEX y LATEX está basado en Plain TEX.



TEX tiene un carácter especial para indicar que cierta secuencia es una instrucción, dicho carácter es (en la mayoría de los casos) \. Sin embargo esto

puede ser cambiado modificando la categoría a la que pertenece \, lo cual podrá ser apreciado más adelante. Entonces cada vez que se quiere dar una instrucción a TEX, es necesario escribir el carácter \ seguido de una instrucción que indique lo que se desea hacer. Por ejemplo, una instrucción válida en TEX es \linebreak.

Como menciona Knuth [11], en TEX hay dos tipos de secuencias de control: el primer tipo son *palabras de control*, las cuales consisten de un carácter de escape (\) seguido por una o más letras, seguidas por un espacio o por algo que no sea una letra. En este tipo de secuencias de control es importante señalar que la palabra siguiente debe estar separada de la instrucción por un espacio, de otra forma TEX no podrá identificar en qué momento termina la palabra de control y en qué momento empieza la siguiente palabra.

El otro tipo de secuencias de control consiste en un carácter de escape seguido por un carácter que no sea una letra, por ejemplo \', estas secuencias son llamadas *símbolos de control*. En estos casos no es necesario poner un espacio que separe la secuencia de control de la letra que sigue, ya que las secuencias de control de este tipo siempre tienen exactamente un símbolo después del carácter de escape.

4.2. Análisis léxico

TEX puede reconocer 256 caracteres diferentes en un archivo, éstos son clasificados en 16 categorías, las cuales se pueden observar en el Cuadro 4.1

La categoría de cualquier carácter puede ser cambiada en cualquier momento, como se mencionó anteriormente para el caso de \, pero se recomienda

Categoría	Significado	Representación común
0	Carácter de escape	\
1	Principio de grupo	{
2	Final de grupo	}
3	Modo matemático	$
4	Tabulador de alineamiento	&
5	Fin de línea	<return>
6	Parámetro	#
7	Superíndice	^
8	Subíndice	_
9	Carácter ignorado	<null>
10	Espacio	␣
11	Letra	A,..., Z, a..., z
12	Otro carácter	Ninguno de los anteriores o siguientes
13	Carácter activo	~
14	Carácter de comentario	%
15	Carácter inválido	<delete>

Cuadro 4.1: Componentes léxicos de TEX

Carácter	Acción
\	Indicar el inicio de una instrucción
{	Indicar el principio de un ambiente
}	Indicar el final de un ambiente
$	Indicador de modo matemático
&	Tabulador
#	Indicador de parámetro
^	Indica que el siguiente carácter será un exponente
_	Indica que el siguiente carácter será un subíndice
%	Indicador de comentario
~	Protector de espacio

Cuadro 4.2: Caracteres especiales de TEX

siempre apegarse a las categorías estándar para evitar errores al escribir un archivo en TEX.

Como se puede notar, TEX tiene reservados algunos caracteres para propósitos especiales, tales como `\`, `{`, `}`, `$`, `&`, `#`, `%`, `^`, `_` y `~`. Al escribir un documento en TEX dichos caracteres no pueden ser usados de como los demás caracteres, ya que cada uno de ellos le indicará a TEX realizar una acción especial, las cuales se pueden encontrar en el Cuadro 4.2.

Si lo que en realidad se quiere hacer es escribir esos símbolos, se deberá hacer de la manera en que se muestra en el Cuadro 4.3.

Cuando TEX lee una línea de texto de un archivo, convierte dicha línea de texto en una lista de componentes léxicos. Un componente léxico es un carácter con el símbolo de la categoría correspondiente o una secuencia de control, por ejemplo, el texto `\raisebox {12 ex}` es convertido a la lista de componentes léxicos:

Qué escribir	Qué se obtiene
\backslash	\
\{	{
\}	}
\$	$
\&	&
\#	#
_	_
\%	%
\^e	ê
\~n	ñ

Cuadro 4.3: Cómo escribir caracteres especiales de TEX

$$\boxed{\texttt{raisebox}}\ \{_1\ \texttt{1}_{12}\ \texttt{2}_{12}\ \sqcup_{10}\ \texttt{e}_{11}\ \texttt{x}_{11}\ \}_2$$

Los subíndices indican la categoría a la que pertenece cada uno de los componentes léxicos. Como puede verse raisebox no tiene ninguna categoría asignada porque es una secuencia de control y no un carácter. También hay que notar que el espacio después de raisebox es ignorado porque sucede a una secuencia de control.

Una vez que TEX ha formado la lista de componentes léxicos correspondiente a la entrada recibida, procederá a las siguientes fases de análisis para dar formato al texto, basándose únicamente en la lista de componentes léxicos.

4.3. Cajas

TEX construye documentos partiendo de formas básicas. Las formas básicas en TEX son *cajas*, cada carácter que TEX procesa es convertido en una caja que contiene a dicho carácter. Después *pega* caracteres en una nueva caja más grande. Las cajas en TEX son formas rectangulares, las cuales tienen tres medidas asociadas que son *altura*, *ancho* y *profundidad*, además de un *punto de referencia* que servirá para alinear las cajas para crear páginas complejas.

Desde el punto de vista de TEX, un carácter de una fuente es una caja, de hecho, la caja más simple que existe. Dicho carácter tiene ya establecidos los valores de altura, ancho y profundidad. TEX usa estos valores para pegar las cajas y determinar la ubicación de los puntos de referencia para todos los caracteres de una página.

No siempre los caracteres están por completo dentro de una caja, por ejemplo, en el caso de los caracteres en itálicas, éstos salen un poco hacia la derecha en la parte superior, pero TEX les da un tratamiento adecuado basándose en otro dato que tiene todo carácter: la corrección itálica. Éste es un número especificado para cada carácter, el cual indica cuánto se extiende dicho carácter hacia la derecha del límite de su caja.

Cualquier cosa contenida en una página construida por TEX está formada por simples cajas de caracteres pegadas. TEX puede pegar cajas de dos maneras distintas: verticalmente u horizontalmente. Cuando TEX construye una lista horizontal de cajas, las alinea para que sus puntos de referencia aparezcan en el mismo renglón, es por eso que las bases de caracteres adyacentes son acomodadas como debe ser. Cuando TEX construye una lista vertical de cajas,

las alinea para que sus puntos de referencia aparezcan en la misma columna vertical.

Cuando TEX crea un bloque de línea, crea una caja horizontal, conocida en TEX como "hbox" porque los componentes de la línea son acomodados de manera horizontal. Si a TEX se le da una instrucción como `\hbox{Una prueba}`, TEX creará una caja horizontal que contenga a las cajas correspondientes a cada uno de los caracteres. Las cajas horizontales forman líneas de una página, estas cajas horizontales pueden ser pegadas poniéndolas en una caja vertical, llamada en TEX "vbox".

Además TEX escoge las divisiones de línea de manera automática, el usuario no debe estar insertando las instrucciones `hbox` y `vbox` a menos que desee tener control absoluto sobre el lugar en que será colocada cada una de las líneas.

Una página completa es por sí misma una caja construida a partir de una lista vertical de cajas más pequeñas que representan las líneas de texto. Cada línea de texto es, a su vez, una caja hecha de una lista horizontal de cajas que representan a cada carácter. En situaciones más complejas, como la construcción de fórmulas matemáticas, se pueden tener cajas dentro de cajas dentro de cajas, y seguir así hasta cualquier nivel. Pero estas construcciones están basadas en el mismo principio de tener cajas horizontales y verticales pegadas entre sí para formar nuevas cajas.

4.4. Pegamento

Para poder adherir las cajas, TEX utiliza un *pegamento*. Pero el pegamento no sólo se encarga de pegar las cajas, se encarga de dejar un espacio ade-

cuado entre cada una de ellas. Por ejemplo, la separación entre palabras no corresponde a una caja vacía, sino a parte del pegamento entre las palabras.

Cuando TEX construye una caja a partir de una lista horizontal o vertical de cajas más pequeñas, hay pegamento entre las cajas pequeñas. El pegamento tiene tres atributos: su espacio *natural*, su capacidad de *extensión* y su capacidad de *contracción*. Basándose en estos valores, TEX determina la cantidad de pegamento que se debe poner entre las palabras al construir una línea tomando en cuenta la longitud que ésta debe tener.

El proceso de determinar la cantidad de pegamento que deberá usarse cuando una caja se forma a partir de una lista vertical u horizontal es conocido como *fijar el pegamento*. Una vez que el pegamento ha sido fijado, es imposible cambiarlo, no puede ser extendido ni contraído de nuevo, y la caja resultante no puede ser descompuesta en partes.

Tanto el espacio natural, como el de extensión y el de contracción pueden ser modificados por el usuario para modelar el resultado a sus necesidades. El espacio *natural* debe ser la cantidad de espacio que se vea mejor; la capacidad de *extensión* debe ser la máxima cantidad de espacio que puede ser agregado al espacio natural antes de que la disposición de las cajas empiece a verse mal; la capacidad de *contracción* debe ser la máxima cantidad de espacio que se le puede quitar al espacio natural para que la composición de las cajas no se vea mal.

En el caso de Plain TEX, éste pone espacio extra al final de una oración; aumenta automáticamente la capacidad de extensión y reduce la de contracción después de signos de puntuación. Esto es porque generalmente resulta mejor incrementar el espacio después de signos de puntuación, que incrementarlo entre dos palabras cualquiera, cuando debe "estirarse" una línea para alcanzar

los márgenes deseados. El problema se presenta cuando TEX no determina de manera adecuada qué es un fin de oración y qué no lo es. Por ejemplo, con las abreviaturas, como Dra., es necesario indicarle que inserte un espacio normal por medio de la instrucción ~.

4.5. Modos

Al procesar un texto, TEX puede encontrarse en alguno de los seis modos siguientes:

Modo vertical. Al construir la lista vertical principal, a partir de la cual las páginas de salida son construidas.

Modo vertical interno. Al construir una lista vertical para una `vbox` (caja vertical).

Modo horizontal. Al construir una lista horizontal para un párrafo.

Modo horizontal restringido. Al construir una lista horizontal para una `hbox` (caja horizontal).

Modo matemático. Al construir una fórmula matemática para colocarse en una lista horizontal.

Modo matemático de exhibición. Al construir una fórmula matemática para colocarse en una línea por sí sola, interrumpiendo temporalmente el párrafo actual.

4.5.1. Modo vertical y vertical interno

TEX se encuentra en uno de los modos verticales cuando está preparando una lista de cajas y pegamento que serán colocadas verticalmente una debajo de la otra en la página.

Cuando TEX empieza a procesar un archivo, se encuentra en modo vertical, listo para empezar a construir páginas. Si estando en este modo se define un pegamento o una caja, éste será colocado en la página actual abajo de lo que se ha definido anteriormente.TEX ignora los espacios en blanco o las líneas en blanco cuando está en modo vertical o en modo vertical interno. Sin embargo \␣ será tomado como el comienzo de un párrafo y el párrafo comenzará con un espacio en blanco inmediatamente después de la sangría. Si mientras TEX se encuentra en modo vertical o vertical interno se encuentra el primer componente léxico de un párrafo, entonces se cambia a modo horizontal.

4.5.2. Modo horizontal y horizontal restringido

TEX se encuentra en uno de los modos horizontales cuando está preparando una lista de cajas y pegamento, que serán colocadas de manera horizontal una junto a otra con los puntos de referencia alineados.

El modo horizontal se usa para hacer los párrafos. Y la mayor parte del tiempo es ocupado en este modo, con breves usos de modo vertical para el espacio entre párrafos.

4.5.3. Modo matemático y matemático de exhibición

TEX se encuentra en uno de los modos matemáticos cuando está leyendo una fórmula. Si un componente léxico indicador de inicio de modo matemático ($) es encontrado en modo horizontal, TEX cambia a modo matemático y procesa la fórmula hasta encontrar el indicador de término de modo matemático (otro $), después pega el texto de la fórmula al párrafo actual y regresa al modo horizontal.

Si lo que se encuentra son dos indicadores de inicio de modo matemático consecutivos ($$), TEX interrumpe el párrafo donde se encuentra, y lo pone en la caja vertical externa, después procesa la fórmula matemática en modo matemático de exhibición, luego pone la caja resultante en la caja vertical externa y regresa al modo horizontal para continuar procesando el párrafo.

Capítulo 5

Convertidor de IIFTEX a LATEX

5.1. Planteamiento

El Departamento de Publicaciones del IIF requiere realizar una actualización de su acervo, ya que se encuentra en formatos que ya no son útiles hoy en día, como IIFTEX. Por esta razón no pueden acceder a ellos, ni transformarlos a formatos como HTML, PS o PDF para brindar versatilidad en la forma en que éstos pueden ser consultados y modificados.

Es de especial interés para este Departamento, contar con versiones en LATEX de todos sus documentos, los cuales se encuentran en IIFTEX. Realizar esta tarea a mano sería imposible, debido al tamaño de su acervo. Además sería posible que quien estuviera llevando a cabo la conversión se equivocara fácilmente por el número de operaciones que deben manejarse, esto sin mencionar la gran cantidad de tiempo que debería invertirse en esa ardua tarea. Es por eso que solicitaron al asesor de esta tesis y a mí, apoyo para resolver este problema.

Se desea automatizar el proceso de conversión entre estos dos tipos de archivos. El programa deberá correr en Windows XP[1], sistema operativo utilizado en el Departamento de Publicaciones del IIF. El programa deberá incluir los patrones de conversiones de instrucciones más comúnmente utilizados en el IIF, sin embargo, se requiere que el usuario pueda personalizar la manera en que serán convertidos los archivos. Es decir, si el usuario desea que cierta instrucción sea convertida de una manera especial, éste puede especificárselo al compilador, el cual deberá tomar en cuenta las condiciones de conversión establecidas por el usuario.

El usuario podrá indicar el nombre que se desee dar al archivo de salida, generado por el convertidor de IIFTEX a LATEX, al cual llamaremos de ahora en adelante `iif2latex`. También se podrá especificar la clase de documento de LATEX que se desea generar, ya sea `book`, `article` o alguna de las otras clases de LATEX, como también las propias clases generadas por el Departamento de Publicaciones, como por ejemplo la clase `critica`.

Si se encuentra, durante el análisis de un archivo, una instrucción que no esté definida por omisión y que tampoco haya sido definida de manera especial por el usuario, se deberá traducir literalmente, agregando un comentario que indique que esa instrucción no fue reconocida por el programa. Esto con la finalidad de que el usuario sepa que parte del archivo no fue transformado y pueda corregir el error.

Preferentemente el programa no deberá detenerse al encontrar un error, deberá seguir analizando el archivo hasta poder mostrar un número de errores pertinente. Además, para llevar a cabo el reporte de errores deberá señalar la naturaleza del error y la parte del archivo donde ocurrió.

[1] Aunque corre en cualquier plataforma que soporte el compilador de Haskell de Glasgow (GHC)

Se deberán sentar las bases para la construcción posterior de una interfaz conveniente para los usuarios del programa ya mencionado. Durante esta etapa el programa funcionará a través de filtros y se correrá desde la terminal del sistema.

En el Apéndice A se encuentra la descripción de los requerimientos del programa.

5.2. Desarrollo

5.2.1. Elección de herramientas y lenguaje

Dado que existen múltiples herramientas hoy en día para generar analizadores léxicos y sintácticos en diversos lenguajes, se debía hacer una selección entre los generadores de analizadores existentes para Haskell. Se decidió utilizar Alex y Happy, ya que los analizadores generados por cada uno de ellos pueden adaptarse entre sí. Además el uso de ambos es sencillo, y brindan la posibilidad de la utilización de mónadas a lo largo del programa.

5.2.2. Descripción de expresiones regulares para el análisis léxico

Tomando en cuenta la estructura léxica utilizada por TEX, se llevó a cabo la definición de las expresiones regulares que servirían para especificar a Alex los componentes léxicos que debería reconocer. Una vez realizada la definición de las expresiones regulares, se procedió a expresarlas por medio de la sintaxis de Alex para su posterior procesamiento.

El archivo de Alex es el siguiente:

Primero se encuentra el encabezado, el cual será copiado al archivo de salida. Aquí únicamente se especifica el nombre del módulo a generar y las funciones y constructores que serán exportados. En este caso serán exportados: el tipo de datos `Token` *con todos sus constructores, el tipo de datos* `AlexPosn` *con todos sus constructores, la función* `alexScanTokens` *y la función* `token_posn`.

```
{
module Lexer (Token(..), AlexPosn(..), alexScanTokens, token_posn)
    where
}
```

Ahora se le indica el tipo de envoltura que se desea utilizar, la cual será la envoltura `posn`. *Se decidió utilizar la envoltura* `posn` *proporcionada por Alex, ya que facilita el manejo de número de línea y columna para cada componente léxico. En esta fase del compilador no se utilizarán mónadas, ya que para la funcionalidad que se necesita, las mónadas no son necesarias. En donde se hará un gran uso de mónadas será en la fase de análisis sintáctico y en algunos módulos auxiliares.*

```
%wrapper "posn"
```

Después se encuentran las definiciones de conjuntos de caracteres que se usarán para definir los componentes léxicos

```
$escapeCharacter = \\       Carácter de escape
$beginningGroup = \{        Carácter de inicio de grupo
$endGroup = \}              Carácter de final de grupo
$mathShift = \$             Carácter de modo matemático
$alignmentTab = &           Carácter de tabulación
$endOfLine = \n             Carácter de fin de línea
```

```
$parameter = \#              Carácter de parámetro
$superscript = \^            Carácter de superíndice
$space = \                   Carácter de espacio
$letter = [a-zA-Z]           Letras de la A a la Z en mayúsculas y
    minúsculas
$commentCharacter = \%       Carácter de comentario
$equals = \=                 Carácter de igualdad
$otherCharacter = [^$escapeCharacter $beginningGroup $endGroup
    $mathShift $alignmentTab $endOfLine $parameter $superscript
    $space $letter $commentCharacter $equals]
                             Otros caracteres
$digito = [0-9]              Dígitos
$char = [$letter $otherCharacter] # $digito
                             Letras y otros caracteres, menos los dígitos
$charEspecialInstruccion = ~$letter
                             Todo lo que no sea una letra

Ahora siguen las definiciones de macros
@item ="item"                La cadena "item"
@instruccionItem = $escapeCharacter @item
                             \ seguido de la cadena "item"
@palabraInstruccion = $letter+
                             Una o más letras
@numero =$digito+            Uno o más dígitos
@unidad ="pt" | "em" | "mm" | "in"
                             Las cadenas "pt", "em", "mm" e "in"
@numeroUnidad = @numero @unidad
                             Un número seguido de una unidad de
                                 medida
@word = $char+               Uno o más caracteres
@instruction = $escapeCharacter @palabraInstruccion
```

```
                                \ seguido de una palabra de instrucción
              | $escapeCharacter $charEspecialInstruccion
                                o \ seguido de un carácter especial de
                                    instrucción
@doubleMathShift = $mathShift $mathShift
                                Dos caracteres de modo matemático juntos
@instruccionEspecial =$escapeCharacter $charEspecialInstruccion
    $letter
                                \ seguido de un carácter especial de
                                    instrucción, seguido de una letra

Ahora se coloca la definición de los componentes léxicos que deberá
    reconocer el analizador generado por Alex. Aquí se hace uso de los
    conjuntos de caracteres y macros que se definieron en la sección
    anterior. Es importante señalar que todas las acciones deben tener el
    mismo tipo, en este caso, el tipo de las acciones es
    AlexPosn→String→Token
tokens :-
    @instruccionItem { toke (λp s. Item p)}
    @instruction { toke (λp s. Instruction p (tail s))}
    $escapeCharacter { toke (λp s. EscapeCharacter p)}
    $beginningGroup { toke (λp s. BeginningGroup p)}
    $endGroup { toke (λp s. EndGroup p)}
    $parameter { toke (λp s. Parameter p)}
    $superscript { toke (λp s. Superscript p)}
    $space { toke (λp s. Space p)}
    $letter { toke (λp s. Letter p (head s))}
    $commentCharacter { toke (λp s. CommentCharacter p)}
    $otherCharacter { toke (λp s. OtherCharacter p (head s))}
    $endOfLine { toke (λp s. EndOfLine p)}
    $alignmentTab { toke (λp s. AlignmentTab p) }
```

```
    $mathShift { toke (λp s. MathShift p) }
    @doubleMathShift { toke (λp s. DoubleMathShift p) }
    @instruccionEspecial { toke (λp s. InstruccionEspecial p (tail s))
        }
    $equals { toke (λp s. Equals p) }
    @word { toke (λp s. Word p s) }
    @numeroUnidad { toke (λp s. NumeroUnidad p s) }
```

Después se encuentra la segunda parte de código opcional. En ésta se definen algunas funciones y tipos de datos que se utilizarán durante el análisis léxico.

```
{
```

Función auxiliar `toke`*, empleada en las acciones asociadas a cada componente léxico*

```
toke f p s = f p s
```

Tipo de datos `Token`*.*

```
data Token =
    Item AlexPosn |
    EscapeCharacter AlexPosn |
    BeginningGroup AlexPosn |
    EndGroup AlexPosn |
    MathShift AlexPosn |
    AlignmentTab AlexPosn |
    EndOfLine AlexPosn |
    Parameter AlexPosn |
    Superscript AlexPosn |
    Subscript AlexPosn |
    IgnoredCharacter AlexPosn |
    Space AlexPosn |
```

```
    Letter AlexPosn Char |          Recibe también un carácter porque
        este patrón puede concordar con más de un lexema, al igual que
        OtherCharacter, Instruction, InstruccionEspecial, Word y
        NumeroUnidad
    OtherCharacter AlexPosn Char |
    CommentCharacter AlexPosn |
    InvalidCharacter AlexPosn |
    Instruction AlexPosn String |
    DoubleMathShift AlexPosn |
    InstruccionEspecial AlexPosn String |
    Word AlexPosn String |
    Equals AlexPosn |
    NumeroUnidad AlexPosn String
    deriving (Eq, Show)

Función token_posn, sirve para recuperar la posición en que fue
    encontrado un componente léxico.
token_posn (Item p) =p
token_posn (EscapeCharacter p) =p
token_posn (BeginningGroup p) =p
token_posn (EndGroup p) =p
token_posn (MathShift p) =p
token_posn (AlignmentTab p) =p
token_posn (EndOfLine p) =p
token_posn (Parameter p) =p
token_posn (Superscript p) = p
token_posn (IgnoredCharacter p) =p
token_posn (Space p) =p
token_posn (Letter p s) = p
token_posn (OtherCharacter p s) =p
token_posn (CommentCharacter p) =p
```

```
token_posn (InvalidCharacter p) = p
token_posn (Instruction p s) = p
token_posn (DoubleMathShift p) =p
token_posn (InstruccionEspecial p s) = p
token_posn (Equals p) =p
token_posn (Word p s) =p
token_posn (NumeroUnidad p s) =p
}
```

La clase de caracteres `otherCharacter` se refiere a todos aquellos caracteres que no tienen un significado especial para TEX, ni son letras ni espacios. La clase de caracteres `char` incluye a las letras, los caracteres especiales y a `item`. La clase `item` representa simplemente a la cadena "item". Se decidió poner esta cadena separada de las demás para poderla manejar de una manera sencilla en la fase de análisis sintáctico durante la conversión de listados, `instruccionItem` representa a la cadena "\item", que es propiamente la instrucción que indica un nuevo elemento en un listado.

El componente `palabraInstruccion` sirve para reconocer aquellas cadenas que pueden ser instrucciones en IIFTEX, es decir, aquellas cadenas que no contengan números ni símbolos especiales, únicamente letras. El componente léxico `word` expresa cualquier número de cualquier clase de caracteres. La última y antepenúltima producciones expresan la forma en que pueden ser construidas instrucciones en IIFTEX. `doubleMathShift` fue incluido como componente léxico para facilitar el reconocimiento del modo matemático de exhibición durante el análisis sintáctico.

En el caso de TEX, éste ignora todos los comentarios (señalados por %) durante el análisis léxico, ya que no le son de ninguna utilidad para la compilación del archivo que procesa. Sin embargo, para la construcción de `iif2latex`

no se ignorarán los comentarios, ya que sí se desea conservarlos. Y cualquier instrucción que sea encontrada en un comentario no será transformada a su equivalente en LaTeX, este hecho resulta evidente más adelante, en la Sección 5.2.3, referente al análisis sintáctico.

5.2.3. Descripción de la gramática para el análisis sintáctico

Para hacer un diseño adecuado de la gramática necesaria para llevar a cabo el análisis sintáctico fue necesario tomar en cuenta las siguientes consideraciones:

1. Hay esencialmente cinco modos en los que IIFTEX puede encontrarse al procesar la entrada:

   Modo matemático. Este modo abarcará lo que en TEX es considerado como dos modos distintos: modo matemático y modo matemático de exhibición.

   Modo texto. Este modo estará activo cuando se procese texto únicamente, no instrucciones.

   Modo comentario. Este modo estará activo cuando se procese un comentario.

   Modo instrucción. Este modo ocurrirá cuando se tenga que procesar una instrucción.

   Modo de listado. Éste ocurrirá al hacer una lista.

2. Empezamos por tener producciones para cada uno de los modos descritos anteriormente.

3. Primero comenzaremos por el modo más sencillo: el modo texto. En este modo puede pasar cualquiera de las siguientes cosas:

   - Encontrar una letra, una palabra o un espacio (tabulador, salto de línea o espacio en blanco), con lo cual seguiría en modo texto.
   - Encontrar un $ o $$, lo que lo haría cambiar a modo matemático.
   - Encontrar una instrucción, lo que lo haría cambiar a modo instrucción.
   - Encontrar un %, lo que lo haría cambiar a modo comentario.

   Es importante señalar que al cambiar a algún modo, una vez que el modo termine de procesarse, éste deberá regresar el control al modo texto.

4. A continuación deben definirse las producciones del modo comentario, éste comienza al encontrar un % y termina cuando encuentra un salto de línea. Cualquier carácter que se encuentre como comentario será escrito exactamente igual al archivo de salida, sin importar si es una instrucción o una fórmula.

5. Ahora podemos definir el modo matemático, dentro del modo matemático podemos tener números, letras, instrucciones, pero no deben sobrar $ (o $$ en caso de que se esté en modo matemático de exhibición). El modo matemático puede ocurrir únicamente en el modo texto, es decir, únicamente estando en modo texto podemos cambiar a modo matemático.

6. Ahora seguiremos con el modo que probablemente sea el más complejo de todos: el de instrucción. El modo instrucción debe ser especificado con mucho cuidado, ya que mientras se está en modo instrucción, es posible cambiar a cualquiera de los otros modos. Además es necesario

considerar los diferentes tipos de instrucciones que pueden encontrarse en IIFTEX, que son:

- Instrucciones que no reciben ningún parámetro, como es el caso de \vfilly \break.
- Instrucciones cuyo único parámetro es el carácter siguiente a la instrucción, como en el caso de \' y \~.
- Instrucciones que reciben uno o más parámetros encerrados en llaves.
- Instrucciones que se encuentran entre llaves, junto con su parámetro, como en el caso de {\bfseries Este texto es el parámetro de bfseries y aparecerá en negritas}.
- Instrucciones de asignación, es decir, aquéllas en las que se utiliza el símbolo =.
- Instrucciones en las que su parámetro va inmediatamente después de ellas, seguido de una unidad de medida, como en el caso de \vskip4pt. Para reconocer este tipo de instrucciones se introdujo en las expresiones regulares para el análisis el componente léxico llamado `numeroUnidad`.
- Un último caso es el de los listados, los cuales consisten de un grupo de instrucciones "\item".

Se tendrá una producción para cada uno de los distintos tipos de instrucciones mencionados.

Debido a algunos conflictos de desplazamiento/reducción, fue necesario señalar la precedencia de los símbolos terminales para poder resolver los conflictos de manera adecuada. Como en el caso de la producción ModoInstruccionDentroLlaves, la cual servirá para reconocer instrucciones que se encuentran

entre llaves, junto con su parámetro, ya que en ésta se introducían conflictos de reducción/desplazamiento. Éstos fueron resueltos dando precedencias a los componentes léxicos `beginningGroup` e `instruction`. Se señala que `beginningGroup` tiene mayor precedencia que `instruction`, de esta manera se consigue que en la regla en la que ocurría el conflicto de reducción/desplazamiento, se decida por el desplazamiento, que es lo deseado en estas circunstancias.

A continuación se muestra la gramática a utilizar para llevar a cabo el análisis sintáctico, lista para ser procesada por Happy, junto con sus funciones auxiliares.

```
-- Primero se indica el nombre del módulo a generar, junto con las
--    funciones que exportará y los módulos que necesita importar.
{module Parser (analiza) where
import Lexer
import IO
import ParserAux
import GeneradorInstrucciones
}

-- Después se indica el nombre que desea dársele a la función que Happy
--    generará para llevar a cabo el análisis sintáctico basándose en la
--    gramática.
%name parse

-- A continuación se le indica el tipo del que son los componentes léxicos.
--    Debe ser Token porque deben conservarse los mismos tipos
--    empleados durante el análisis léxico, para que ambos analizadores
--    sean compatibles.
%tokentype {Token}
```

Ahora se especifica que se utilizará un analizador sintáctico que usa mónadas. El constructor será `IO` y las funciones de ligado y regreso serán `>>=` y `return`, respectivamente. Se usará la mónada `IO` para poder llevar a cabo un manejo adecuado de la entrada y salida y de los mensajes que se mostrarán al usuario en caso de que ocurra un error.

%monad {IO} {>>=} {return}

Ahora se indica la precedencia de algunos símbolos terminales para resolver conflictos de reducción desplazamiento

%left AUX Éste no es un símbolo terminal, se usa en la definición de la gramática para indicar la precedencia de una regla

%right instruction

%right beginningGroup

%left numeroUnidad

%right mathShift space letter otherCharacter commentCharacter instruction

doubleMathShift instruccionEspecial word

%right superscript subscript equals

Ahora se enumeran los distintos componentes léxicos que recibirá del analizador léxico.

%token

escapeCharacter { EscapeCharacter _}

beginningGroup { BeginningGroup _}

endGroup { EndGroup _}

mathShift { MathShift _}

alignmentTab { AlignmentTab _}

endOfLine { EndOfLine _}

parameter { Parameter _}

superscript { Superscript _}

subscript { Subscript _}
ignoredCharacter { IgnoredCharacter _}
space { Space _}
letter { Letter _ $$ } *Aquí es utilizado el símbolo* **$$** *para indicar que ése será el valor del componente léxico* `letter`*, sucede lo mismo en las demás ocurrencias de* **$$** *en los siguientes componentes léxicos.*
otherCharacter { OtherCharacter _$$ }
commentCharacter { CommentCharacter _}
invalidCharacter { InvalidCharacter _}
instruction { Instruction _ $$ }
doubleMathShift { DoubleMathShift _}
instruccionEspecial { InstruccionEspecial _ $$ }
equals {Equals _}
word { Word _$$ }
item { Item _ }
numeroUnidad { NumeroUnidad _$$ }

Este símbolo indica el principio de la definición de la gramática
%%

Basándose en las consideraciones hechas anteriormente, se diseñó la gramática. Como se había mencionado anteriormente, se tiene una producción por cada uno de los modos en que puede encontrarse `iif2latex`*. También hay algunas producciones que sirven como auxiliares, cuando algunas reglas se repiten en varias producciones. Tal es el caso de la producción* `CaracteresComunes`*, la cual es usada en* `ModoTextoSinSalto`*,* `ModoComentario` *y* `ModoComentario`*. Lo mismo ocurre con la producción* `ComunesMat`*, que es usada en* `ModoMatemático` *y* `ModoComentario`*. Como se podrá apreciar, todas las producciones hacen uso de mónadas. Esto es, por un lado, para*

poder desplegar mensajes de error al usuario y también para procesar la tabla que contiene las conversiones a realizar con cada una de las instrucciones. Debido a que el usuario puede modificar la tabla de conversión a utilizar, ésta debe ser capaz de soportar la mónada de entrada y salida para integrar las conversiones proporcionadas por el usuario.

```
ModoTexto ::{ IO String }
ModoTexto : ModoTexto endOfLine { concatIO $1 (return "\n") }
          | endOfLine { return "\n" }
          | ModoTexto ModoListado endOfLine endOfLine { concatIO
            (concatIO (concatIO $1 (return "\\begin{itemize}\n"))
            $2)(return "\\end{itemize}\n\n") }
          | ModoListado endOfLine endOfLine { concatIO (concatIO (
            return "\\begin{itemize}\n") $1) (return "\\end{
            itemize}\n\n") }
          | ModoTexto ModoTextoSinSalto %prec AUX{ concatIO $1
            $2 }
          | ModoTextoSinSalto %prec AUX { $1 }

ModoListado ::{ IO String }
ModoListado : ModoListado item ModoTextoSinSalto endOfLine {
    concatIO (concatIO (concatIO $1 (return "\\item ")) $3) (return
    "\n") }
            | item ModoTextoSinSalto endOfLine { concatIO (
              concatIO (return "\\item ") $2) (return "\n") }

CaracteresComunes ::{ IO String }
CaracteresComunes : CaracteresComunes word { concatIO $1 (return
    $2) }
                  | word { return $1 }
```

```
                | CaracteresComunes letter { concatIO $1 (return [
                  $2]) }
                | letter { return [$1] }
                | CaracteresComunes space { concatIO $1 (return "
                  ") }
                | space { return " " }
                | CaracteresComunes otherCharacter { concatIO $1 (
                  return [$2]) }
                | otherCharacter { return [$1] }

Comentario :: { IO String }
Comentario : commentCharacter ModoComentario endOfLine {
    concatIO (concatIO (return "%\"") $2) (return "\"\n") }

ComunesMat ::{ IO String }
ComunesMat : ComunesMat superscript { concatIO $1 (return "^") }
           | superscript { return "^" }
           | ComunesMat subscript { concatIO $1 (return "_") }
           | subscript { return "_" }
           | ComunesMat equals { concatIO $1 (return "=") }
           | equals { return "=" }

ModoTextoSinSalto ::{ IO String }
ModoTextoSinSalto : ModoTextoSinSalto mathShift ModoMatematico
    mathShift { concatIO (concatIO (concatIO $1 (return "$")) $3) (
    return "$") }
                | mathShift ModoMatematico mathShift { concatIO
                  (concatIO (return "$") $2) (return "$") }
                | ModoTextoSinSalto doubleMathShift
                  ModoMatematico doubleMathShift { concatIO (
                  concatIO (concatIO $1 (return "$$")) $3) (
```

```
                    return "$$") }
                | doubleMathShift ModoMatematico
                    doubleMathShift { concatIO (concatIO (return
                    "$$") $2) (return "$$") }
                | ModoTextoSinSalto numeroUnidad { concatIO $1 (
                    return $2) }
                | numeroUnidad { return $1 }
                | ModoTextoSinSalto CaracteresComunes %prec
                    AUX { concatIO $1 $2 }
                | CaracteresComunes %prec AUX { $1 }
                | ModoTextoSinSalto Comentario { concatIO $1 $2 }
                | Comentario { $1 }
                | ModoTextoSinSalto ModoInstruccion { concatIO $1
                    $2 }
                | ModoInstruccion { $1 }

ModoInstruccion :: { IO String }
ModoInstruccion : ModoInstruccionLlaves { $1 }
                | instruction { buscarInstruccion (armaInstruccion $1
                    []generaCadenaInstruccion) }
                | instruccionEspecial {buscarInstruccion (
                    armaInstruccion [head $1] [tail $1]
                    generaCadenaInstruccionSimple) }
                | ModoInstruccionDentroLlaves { $1 }
                | InstruccionAsignacion { concatIO (return "
                    Asignación ") $1 }
                | InstruccionPegada { concatIO (return "Pegada ") $1
                    }

ModoComentario ::{ IO String }
```

```
ModoComentario : ModoComentario escapeCharacter { concatIO $1 (
    return "\\") }
                | escapeCharacter { return "\\" }
                | ModoComentario beginningGroup { concatIO $1 (
                    return "{") }
                | beginningGroup { return "{" }
                | ModoComentario endGroup { concatIO $1 (return
                    "}") }
                | endGroup { return "}" }
                | ModoComentario mathShift { concatIO $1 (return "$
                    ") }
                | mathShift { return "$" }
                | ModoComentario alignmentTab { concatIO $1 (
                    return "&") }
                | alignmentTab { return "&" }
                | ModoComentario parameter { concatIO $1 (return
                    "#") }
                | parameter { return "#" }
                | ModoComentario ignoredCharacter { $1 }
                | ignoredCharacter { return "" }
                | ModoComentario commentCharacter { concatIO $1 (
                    return "%")}
                | commentCharacter { return "%" }
                | ModoComentario instruction { concatIO (concatIO $1
                    (return "\\")) (return $2) }
                | instruction { concatIO (return "\\") (return $1) }
                | ModoComentario doubleMathShift { concatIO $1 (
                    return "$$") }
                | doubleMathShift { return "$$" }
                | ModoComentario instruccionEspecial { concatIO (
                    concatIO $1 (return "\\")) (return $2) }
```

```
                | instruccionEspecial { concatIO (return "\\") (
                    return $1) }
                | MododoComentario item { concatIO $1 (return "\\item
                    ") }
                | item { return "\\item" }
                | MododoComentario invalidCharacter { $1 }
                | invalidCharacter { return "" }
                | MododoComentario CaracteresComunes %prec AUX{
                    concatIO $1 $2 }
                | CaracteresComunes %prec AUX { $1 }
                | MododoComentario ComunesMat %prec AUX {
                    concatIO $1 $2 }
                | ComunesMat %prec AUX { $1 }

MododoMatematico ::{ IO String }
MododoMatematico : MododoMatematico beginningGroup { concatIO $1 (
    return "{") }
                | beginningGroup { return "{" }
                | MododoMatematico endGroup { concatIO $1 (return
                    "}") }
                | endGroup { return "{" }
                | MododoMatematico endOfLine { concatIO $1 (return "
                    \n") }
                | endOfLine { return "\n" }
                | MododoMatematico CaracteresComunes %prec AUX {
                    concatIO $1 $2 }
                | CaracteresComunes %prec AUX { $1 }
                | MododoMatematico Comentario { concatIO $1 $2 }
                | Comentario { $1 }
                | MododoMatematico ComunesMat %prec AUX {
                    concatIO $1 $2 }
```

```
                | ComunesMat %prec AUX { $1 }
                | ModoMatematico ModoInstruccion { concatIO $1 $2 }
                | ModoInstruccion { $1 }

ModoInstruccionLlaves :: { IO String }
ModoInstruccionLlaves : instruction Llaves { buscarInsMon (
    armaInsMon $1 $2 generaCadenaInstruccion) }

Llaves :: { [IO String] }
Llaves : Llaves beginningGroup ModoTexto endGroup { $1 ++[$3] }
       | beginningGroup ModoTexto endGroup { [$2] }

ModoInstruccionDentroLlaves ::{ IO String }
ModoInstruccionDentroLlaves : beginningGroup instruction ModoTexto
    endGroup {buscarInsMon (armaInsMon $2 [$3]
    generaCadenaInstruccion) }

InstruccionAsignacion :: { IO String }
InstruccionAsignacion : instruction equals instruction { concatIO (
    concatIO (buscarInstruccion (armaInstruccion $1 []
    generaCadenaInstruccion)) (return "=\\")) (buscarInstruccion (
    armaInstruccion $3 []generaCadenaInstruccion))}
                      | instruction equals numeroUnidad { concatIO (
                          buscarInstruccion (armaInstruccion $1 []
                          generaCadenaInstruccion))(concatIO (return
                          "=") (return $3)) }

InstruccionPegada :: { IO String }
InstruccionPegada : instruction numeroUnidad { concatIO (
    buscarInstruccion (armaInstruccion $1 []generaCadenaInstruccion)
    ) (return $2) }
```

Una vez definidas las producciones de la gramática, puede incluirse el segundo bloque de código opcional. En este caso, dicho bloque servirá para definir funciones auxiliares para el manejo y recuperación de errores. También se encontrará aquí la definición de la función que llamará tanto al analizador léxico como al sintáctico para llevar el análisis y conversión de la entrada.

{

La función ***happyError*** *será llamada en caso de encontrar un error al hacer el análisis. Si se encuentra un error, el programa emitirá un mensaje avisando al usuario la línea y columna aproximada –porque en algunos casos el error puede no ser detectado oportunamente– en que se encuentra dicho error y después seguirá analizando la entrada. Es importante señalar que en caso de que ocurra un error, no será generado ningún archivo de salida.*

```
happyError :: [Token] → IO a
happyError []=error "No se generó archivo de salida porque se
    encontró error al final del archivo de entrada"
happyError tks =do
                putStr ("Error: " ++(errorAuxLinCol tks) ++"\n")
                parse tks
                error "No se generó archivo de salida porque se
                    encontraron errores en el archivo de entrada"

errorAuxLinCol :: [Token] → String
errorAuxLinCol tks =case tks of
          tk:_ → errorAux tk l c
                                  where
                                  AlexPn _l c = token_posn tk
              [] → "al final del archivo"
```

```
errorAux :: Token → Int → Int → String
errorAux tk l c =
    case tk of
        BeginningGroup _ → "{ no esperado en L" ++show l ++":C"
            ++show c
        EscapeCharacter _ → "\\ no esperado en L" ++show l ++":C
            " ++show c
        EndGroup _ → "} no esperado en L" ++show l ++":C" ++
            show c
        MathShift _ → "$ no esperado en L" ++show l ++":C" ++
            show c
        AlignmentTab _ → "& no esperado en L" ++show l ++":C"
            ++show c
        EndOfLine _ → "FIN DE LÍNEA no esperado en L" ++show l
            ++":C" ++show c
        Parameter _ → "# no esperado en L" ++show l ++":C" ++
            show c
        Superscript _ → "^ no esperado en L" ++show l ++":C" ++
            show c
        Subscript _ → "_ no esperado en L" ++show l ++":C" ++
            show c
        IgnoredCharacter _ → "Carácter no válido en L" ++show l
            ++":C" ++show c
        Space _ → "espacio no esperado en L" ++show l ++":C" ++
            show c
        Letter _ caracter → "Carácter \"" ++[caracter] ++"\" no
            esperado en L" ++show l ++":C" ++show c
        OtherCharacter _caracter → "Carácter \"" ++[caracter] ++"
            \" no esperado en L" ++show l ++":C" ++show c
```

```
        CommentCharacter _ → "% no esperado en L" ++show l ++":
            C" ++show c
        InvalidCharacter _  → "Carácter no válido en L" ++show l
            ++":C" ++show c
        Instruction _ ins  → "Instrucción \"" ++ins ++"\" no
            esperada en L" ++show l ++":C" ++show c
        DoubleMathShift _ → "$$ no esperado en L" ++show l ++":C
            " ++show c
        Equals _  → "= no esperado en L" ++show l ++":C" ++show
            c
        Item _  → "\\item no esperado en L" ++show l ++":C" ++
            show c
        Word _word  → "Palabra \"" ++word ++"\" no esperada en L"
             ++show l ++":C" ++show c
        Item _ → "\\item no esperado en L" ++show l ++":C" ++
            show c
        NumeroUnidad _num  → "Unidad " ++num ++" no esperada
            en L" ++show l ++":C" ++show c

analiza cadena = parse (alexScanTokens cadena)

}
```

5.2.4. Descripción de módulos auxiliares

Además de los módulos `Lexer` y `Parser`, generados por Alex y Happy respectivamente, para obtener la funcionalidad deseada, fue necesario crear varios módulos más que se ocuparan de tareas específicas. A continuación se describe cada uno de estos módulos de manera breve. En el Apéndice B se encuentra el código completo de cada uno de estos módulos.

ConfigLexer. Se encarga de hacer el análisis léxico del archivo de configuración que de manera opcional el usuario puede proporcionar para reescribir las definiciones dadas por omisión a las instrucciones a transformar.

ConfigParser. Se encarga de hacer el análisis sintáctico del archivo de configuración.

GeneradorInstrucciones. Sirve para dar el formato adecuado a las instrucciones que `iif2latex` escribirá en el archivo de salida.

JuntaListasConfig. Es el encargado de unir las diferentes tablas de equivalencia de instrucciones. Por ejemplo, si el usuario incluye un archivo de configuración, debe unirse la tabla del usuario con la tabla por omisión del programa.

Main. Es el módulo principal, tiene la función `main`, la cual se encarga de llamar a `comienza` del módulo `ParámetrosLíneaDeComandos`.

NumeroCadena. Lleva a cabo operaciones entre números y cadenas, tales como transformar un número a cadena o viceversa.

ParametrosLineaDeComandos. Junta y valida los parámetros dados por el usuario desde la línea de comandos.

ParamsAux. Sólo tiene definido el tipo Params.

ParserAux. Contiene algunas funciones que le serán útiles al analizador sintáctico al estar haciendo la conviersión, también contiene la tabla de equivalencias para convertir las instrucciones de manera apropiada.

Transformador. Contiene la función que se encarga de leer el archivo fuente y después escribir el resultado de la conversión en el archivo de salida.

5.3. Ejemplos de uso de IIF2LATEX

Las distintas opciones que pueden indicársele a IIFLATEX son las siguientes:

-o Fija el nombre del archivo a transformar.

-d Fija el nombre que se dará al archivo de salida.

-c Fija la clase que se le dará al archivo resultante.

-a Fija el código de caracteres en el que se encuentra el archivo de entrada, esto para poder llevar a cabo la sustitución de caracteres especiales del español.

-f Fija el nombre del archivo de configuración

-h Despliega la ayuda del programa

Es necesario que siempre se especifiquen los nombres del archivo de entrada y el de salida, los cuales no deben ser iguales. Todos los demás parámetros son opcionales.

5.3.1. Ejemplo de uso No. 1

La opción más simple que puede utilizarse es la siguiente:

```
iif2latex -h
```

con la cual se obtiene como resultado:

IIF2LATEX

AYUDA

.Opciones:

−o <nombreEntrada> Fija el nombre del archivo sobre el cual se desea hacer la transformación

−d <nombreSalida> Fija el nombre del archivo en el que deberá guardarse el resultado

−c <claseDocumento> Fija la clase a utilizar . Si no especifica , se tomará book

−a <codigo> Fija el código a considerar si se desea llevar a cabo la conversión de acentos y demás caracteres especiales

−f <nombreConfig> Fija el nombre del archivo de configuración

−h Despliega esta lista de opciones

Es necesario que SIEMPRE se especifiquen: el nombre del archivo de entrada y el de salida.

Los demás parámetros son opcionales

Los nombres de los archivos de entrada y salida no deben ser los mismos

El mismo resultado se obtiene si no se le da ninguna opción a IIFLATEX.

Para procesar un archivo con IIF2LATEX, la manera más simple de hacerlo es con:

```
iif2latex -o ejemplo.tex -d salidaEjemplo.tex
```

De esta forma se procesa el archivo "ejemplo.tex" y el resultado es guardado en el archivo "salidaEjemplo.tex". Como no se le dan opciones adicionales, la

clase de este documento será *book*, no se considerará ningún archivo de configuración y tampoco se llevará a cabo la conversión de caracteres especiales utilizados en el español.

Si se desea especificar que la clase a utilizar será *article* y que debe llevarse a cabo la conversión de caracteres especiales del español usando el código de caracteres 8859, deberá hacerse de la siguiente manera:

```
iif2latex -o ejemplo.tex -d salidaEjemplo.tex -a 8859 -c article
```

Si además se necesita utilizar un archivo de configuración, esto debe ser especificado de la siguiente manera:

```
iif2latex -o ejemplo.tex -d salidaEjemplo.tex -a 8859 -c article
-f config.txt
```

Si el contenido del archivo "ejemplo.tex" es:

```
%cambiar todas las pautas por patrones
\pageno=25
\capit{2}{EL LOGRO DE DARWIN}{Philip Kitcher}
\cornon{El logro de Darwin}

\secin{1.}{Un triunfo de la herej\'ia}

\noindent Entre 1844, cuando Charles Darwin confes\'o por primera vez a
Joseph Hooker sus ideas heterodoxas sobre "la cuesti\'on de las
especies ", y 1871, a\~no en que Thomas Henry Huxley crey\'o conveniente
declarar que "en una docena de a\~nos {\it El origen de las especies\/} ha
tenido como efecto una revoluci\'on tan completa en la ciencia biol\'ogica
como los {\it Principia\/} lo tuvieron en astronom\'ia", la biolog\'ia
experiment\'o la transformaci\'on m\'as importante de su historia.\nota{(
    Huxley
```

```
1896, p.~120). El pasaje se cita en (F. Darwin 1888, III, p.~132). Tres
a\~nos antes de que se escribiera (es decir, en 1868), Darwin se sinti\'o
preparado para hablar de "la creencia casi universal en la evoluci\'on (de
alg\'un modo) de las especies" (F. Darwin 1903, I, p.~304).

\sec{2.}{Antes de Darwin}

\noindent Para comprender el contexto en el que se recibieron y
discutieron las ideas de Darwin, necesitamos identificar algunos de los
principales temas del pensamiento de los primeros decenios del siglo
\vyv{XIX} sobre la historia de la Tierra y de la vida. Desde fines del
siglo \vyv{XVIII} se hab\'ia vuelto cada vez m\'as evidente para los
entendidos que el periodo transcurrido desde que surgi\'o la vida en la
tierra era mucho mayor de lo que tradicionalmente se hab\'ia supuesto.\break

\cita ¿Por qu\'e los organismos que pertenencen al grupo $G$ y que viven
en el medio $M$ tienen la propiedad $P$?\fincita
```

Y el contenido del archivo de configuración *config.txt* es:

```
\secin #2 →\section #2
```

El resultado de `iif2latex -o ejemplo.tex -d salidaEjemplo.tex -a 8859 -c article -f config.txt` es:

```
\documentclass{article}
\begin{document}
%cambiar todas las pautas por patrones
\setcounter{page}{25}
\chapter{EL LOGRO DE DARWIN}{Philip Kitcher}\setcounter{chapter
    }{2}
```

```
%% IIF2LaTeX: Instrucción no encontrada en tabla de equivalencias, se
    transcribe literalmente %%
\cornon{El logro de Darwin}

\section{Un triunfo de la herejía}

\noindent Entre 1844, cuando Charles Darwin confesó por primera vez a
Joseph Hooker sus ideas heterodoxas sobre "la cuestión de las
especies ", y 1871, año en que Thomas Henry Huxley creyó conveniente
declarar que "en una docena de años \textit{ El origen de las especies\/} ha
tenido como efecto una revolución tan completa en la ciencia biológica
como los \textit{ Principia\/} lo tuvieron en astronomía", la biología
experimentó la transformación m s importante de su historia.\footnote{(Huxley
1896, p.~120). El pasaje se cita en (F. Darwin 1888, III, p.~132). Tres
años antes de que se escribiera (es decir, en 1868), Darwin se sintió
preparado para hablar de "la creencia casi universal en la evolución (de
alg£n modo) de las especies". (F. Darwin 1903, I, p.~304)}.

\section{Antes de Darwin}

\noindent Para comprender el contexto en el que se recibieron y
discutieron las ideas de Darwin, necesitamos identificar algunos de los
principales temas del pensamiento de los primeros decenios del siglo
\textsc{XIX} sobre la historia de la Tierra y de la vida. Desde fines del
siglo \textsc{XVIII} se había vuelto cada vez m s evidente para los
entendidos que el periodo transcurrido desde que surgió la vida en la
tierra era mucho mayor de lo que tradicionalmente se había supuesto.\break

\begin{quotation}¿Por qué los organismos que pertenencen al grupo $G$ y
    que viven
en el medio $M$ tienen la propiedad $P$?
```

```
\end{quotation}

\end{document}
```

Capítulo 6

Conclusiones

El compilador realizado cumple con los requerimientos señalados. Se usó Haskell para realizar dicho compilador, utilizando los generadores de analizadores léxicos y sintácticos Alex y Happy. Se mostró que Haskell es un lenguaje poderoso que puede ser usado en aplicaciones reales, obteniendo un buen desempeño.

El código necesitado para poner en marcha el compilador es corto, las funciones definidas son claras y concisas. La organización que Haskell ofrece por medio de módulos fue de gran ayuda para el diseño –y por lo tanto para la implementación– del compilador. Al estructurar el programa en módulos fue posible tener una organización clara del programa desde el principio del desarrollo, en la fase de diseño.

Los compiladores construidos en Haskell son eficientes y fáciles de realizar. Lo que requiere de especial cuidado es la especificación de las expresiones regulares y la gramática para el analizador léxico y sintáctico, respectivamente, ya que de la buena especificación de éstas depende el adecuado funcionamiento

y desempeño del compilador, porque si se define con cuidado la gramática para el análisis sintáctico, se tendrán menos estados y menos reglas. Una vez que se han definido éstas adecuadamente, basta con dar las definiciones pertinentes en archivos que después procesarán Alex y Happy.

En las primeras versiones de la gramática para llevar a cabo el análisis sintáctico, se tenían 131 reglas y 176 estados. Debido al tiempo invertido en mejorar la definición de la gramática y al uso de precedencias para evitar los conflictos de reducción/desplazamiento, fue posible reducir la gramática hasta lograr que el analizador sintáctico constara únicamente de 93 reglas y 122 estados.

Otras consideraciones importantes que se deben tener es el tipo de analizadores que se desean realizar, si se usarán mónadas, qué técnica de recuperación de errores se adapta mejor a los errores comúnmente cometidos por los futuros usuarios del compilador y qué otras funciones se requiere tener, para poder diseñar los módulos auxiliares que utilizará el compilador.

Una de las dificultades más grandes experimentadas durante el desarrollo del proyecto fue el manejo adecuado de las mónadas, en particular la mónada de entrada y salida IO, ya que al ser una mónada de un solo sentido, una vez que se define una función que utiliza dicha mónada, todas las funciones que estén relacionadas con dicha función deben considerar el manejo de la mónada IO.

En el caso del compilador realizado, la mónada IO forma una parte esencial desde el principio del programa, porque es necesario procesar los parámetros que el usuario da en la entrada, desde la línea de comandos. Por esto resulta indispensable manipular la mónada IO a lo largo de todo el programa.

Haskell, además de brindar elegancia y comodidad al programar, brinda estabilidad al programa, ya que ocurren menos errores, los cuales son fáciles de

identificar. Si a esto agregamos todas las características que Haskell, como lenguaje funcional posee, como son la evaluación perezosa y el uso de mónadas, resulta evidente que en Haskell tenemos una gran herramienta para desarrollar software en gran variedad de ámbitos.

Además de mostrar lo que se puede lograr con Haskell, el compilador realizado proporciona herramientas y consejos para solucionar problemas a los que se debe enfrentar un desarrollador de software al aventurarse en proyectos realizados utilizando programación funcional.

Lo único que resta por hacer es poner el software generado a la disposición de la comunidad, bajo la licencia GNU. Al hacer esto, los objetivos se habrán cumplido por completo: no sólo se habrá demostrado que Haskell es poderoso también para realizar programas de tamaño mediano y grande, sino también se habrá brindado a la comunidad, guías, ejemplos y posibles soluciones de algunos problemas a los que hay que enfrentarse en el proceso de construcción de un compilador, y, en general al realizar programación a gran escala en un lenguaje funcional, como Haskell.

Apéndice A

Requerimientos del convertidor de IIF a LaTeX

1. Sistema Operativo en el que deberá funcionar: Windows XP
2. Tipo de interfaz: No gráfica, por medio de filtros desde la terminal del sistema.
3. Debe transformar archivos en formato IIFTeX en archivos en formato LaTeX, sin introducir errores.
4. Debe ser estable.
5. El sistema generado debe estar documentado.
6. Debe diseñarse pensando siempre en las posibles adaptaciones que se le puedan hacer en un futuro, procurando que sean sencillas de realizar.
7. Ofrece la oportunidad de cambiar los caracteres especiales, por ejemplo, tener á en lugar de \'a. Para este fin, el usuario deberá señalar el

código de caracteres en el que se encuentra el archivo a convertir para poder hacer la sustitución de dichos caracteres de acuerdo al código especificado.

8. Se debe permitir al usuario cambiar las equivalencias definidas en el programa, esto será hecho por medio de archivos de configuración. En éstos, el usuario escribirá la instrucción (o instrucciones) cuya equivalencia desea cambiar y en seguida señalará a qué instrucción desee que sea convertida.

9. El usuario proporciona al sistema el nombre del archivo de entrada y el de salida; de manera opcional proporcionará también la clase de documento a generar y el código de caracteres en que se encuentra el archivo, esto sólo en caso de que se desee hacer la transformación de caracteres especiales.

10. En caso de no encontrarse una instrucción en la definición del convertidor, y tampoco encontrarse en el archivo de configuración dado por el usuario (si es que existe), debe ponerse un comentario de la forma:

    %%%IIF2LaTeX: Instrucción no encontrada, se transcribe literalmente %%%

    para indicarle al usuario que se encontró una instrucción no definida y el programa la va a ignorar.

Apéndice B

Módulos auxiliares

B.1. GeneradorInstrucciones

```
module GeneradorInstrucciones where
import ParamsAux
import NumeroCadena

generaCadenaInstruccion nombre parametrosActuales params ="\\" ++
    nombre ++(sacaParamsATransformar parametrosActuales params)

generaCadenaInstruccionBeginEnd ::String → Params → Params →
    String
generaCadenaInstruccionBeginEnd nombre parametros params ="\\
    begin{" ++nombre ++"}" ++(sacaParamsATransformarAsignacion
    parametros params) ++"\\end{" ++nombre ++"}"

sacaParamsATransformar ::Params → Params → String
```

```
sacaParamsATransformar (param:params) parametros =(
    analizaParamATransformar param parametros) ++(
    sacaParamsATransformar params parametros)
sacaParamsATransformar []_=""

sacaParamsATransformarAsignacion ::Params → Params → String
sacaParamsATransformarAsignacion (param:params) parametros =(
    analizaParamATransformarAsignacion param parametros) ++(
    sacaParamsATransformarAsignacion params parametros)
sacaParamsATransformarAsignacion []_=""

analizaParamATransformar ::String → Params → String
analizaParamATransformar param parametros =
    if (((length param) ≥2) ∧((head param) =='#'))
        then (obtenParametro (tail param) parametros)
        else (agregaLlaves param)

analizaParamATransformarAsignacion ::String → Params → String
analizaParamATransformarAsignacion param parametros =
    if (((length param) ≥2) ∧((head param) =='#'))
        then (obtenParametroAsignacion (tail param) parametros)
        else param

obtenParametro :: String → Params → String
obtenParametro cadNumero parametros =agregaLlaves (parametros !! (
    num − 1))
                    where num =stringAInt cadNumero

agregaLlaves :: String → String
agregaLlaves cadena = "{" ++cadena ++"}"
```

```
obtenParametroAsignacion ::String → Params → String
obtenParametroAsignacion cadNumero parametros =parametros !! (num
    - 1)
                        where num =stringAInt cadNumero

generaCadenaInstruccionAsignacion ::String → Params → Params
    → String
generaCadenaInstruccionAsignacion nombre parametros params ="\\"
    ++nombre ++"=" ++(sacaParamsATransformarAsignacion
    parametros params)

generaCadenaInstruccionSimple ::String → Params → Params →
    String
generaCadenaInstruccionSimple nombre parametros params ="\\" ++
    nombre ++(sacaParamsATransformarAsignacion parametros params
    )

transcribe :: String → Params → Params → String
transcribe nombre parametros params =nombre ++(
    sacaParamsATransformarAsignacion parametros params)

generaCadenaItem ::String → Params → Params → String
generaCadenaItem nombre parametrosActuales params ="\\begin{
    itemize} \n" ++(sacaParamsItemize params) ++"\\end{itemize}"

sacaParamsItemize :: Params → String
sacaParamsItemize []=""
sacaParamsItemize (param:params) ="\\item " ++param ++"\n" ++(
    sacaParamsItemize params)
```

B.2. JuntaListasConfig

```
module JuntaListasConfig where

juntaListasConfig [] listaOriginal = listaOriginal
juntaListasConfig (elemConfig:listaConfig) listaOriginal =
    juntaListasConfig listaConfig (buscaSustituyeEnLista elemConfig
        []listaOriginal)

buscaSustituyeEnLista config principioLista [] = principioLista ++ [
    config]
buscaSustituyeEnLista config principioLista (insLista:restoLista) =
    if((instruccionConfig == instruccion) ∧ (noParamsConfig ==
        noParams))
        then (principioLista ++ (config:restoLista))
        else (buscaSustituyeEnLista config (principioLista ++
                [insLista]) restoLista)
    where instruccionConfig =obtenNombreInstruccion config
          instruccion = obtenNombreInstruccion insLista
          noParamsConfig =obtenNoParams config
          noParams =obtenNoParams insLista

obtenNombreInstruccion (nombreInstruccion, _, _) =nombreInstruccion

obtenNoParams (_, noParams, _) =noParams
```

B.3. Main

```
module Main where
import ParametrosLineaDeComandos
```

```
main =comienza
```

B.4. NumeroCadena

```
module NumeroCadena (stringAInt) where
import Data.Char

stringAInt :: String → Int
stringAInt as = stringAIntAux as 0

stringAIntAux :: String → Int → Int
stringAIntAux []_= 0
stringAIntAux as count =charToNum (last as) ×(10^count) +
    stringAIntAux (init as) (count + 1)

charToNum ::Char → Int
charToNum c
    | isDigit c = (ord c − 48)
    | otherwise =0
```

B.5. ParámetrosLíneaDeComandos

```
module ParametrosLineaDeComandos where
import IO
import System
import Lexer
import TransformaParams
import ConfigParser
import Transformador
import ParamsAux
```

```
import ParserAux
import GeneradorInstrucciones
```

Función principal. Se encarga de leer los argumentos de la línea de comandos,llenar la tupla de manera correcta y llamar a la función aplicaParametros paraque pueda llevarse a cabo la transformación

```
comienza =do putStr "\n\t\tEmTeX2LaTeX\n\n"
            argumentos ← System.getArgs
            aplicaParametros (revisaArgumentos(llenaTuplaArgumentos
                argumentos ("error", "", "book", 0, "", False)))
            return ()
```

Basándose en la tupla que contiene toda la información, llama ala función preConvierte con los parámetros adecuados.

```
aplicaParametros("error","","book",0,"",False) =
    do putStr "\n\t\tAYUDA\n\nOpciones:\n"
       putStr "-o <nombreEntrada>\tFija el nombre del archivo
           sobre el cual se desea hacer la transformación\n"
       putStr "-d <nombreSalida>\tFija el nombre del archivo en el
           que deberá guardarse el resultado\n"
       putStr "-c <claseDocumento>\tFija la clase a utilizar. Si no
           especifica , se tomará book\n"
       putStr "-a <codigo>\tFija el código a considerar si se desea
           llevar a cabo la conversión de acentos y demás caracteres
           especiales\n"
       putStr "-f <nombreConfig>\tFija el nombre del archivo de
           configuración\n"
       putStr "-h \t\t\tDespliega esta lista de opciones\n\n"
```

```
        putStr "Es necesario que SIEMPRE se especifiquen: el nombre
            del archivo de entrada y el de salida.\n"
        putStr "Los demás parámetros son opcionales\n\n"
        putStr "Los nombres de los archivos de entrada y salida no
            deben ser los mismos\n\n"
        return ()

aplicaParametros (nomOrig, nomDest, tipoDoc, 0, [], False) =
    convierte nomOrig nomDest tipoDoc
aplicaParametros (nomOrig, nomDest, tipoDoc, a, [], False) =
    convierte nomOrig nomDest tipoDoc
aplicaParametros (nomOrig, nomDest, tipoDoc, 0, conf, False) =
    convierte nomOrig nomDest tipoDoc
aplicaParametros (nomOrig, nomDest, tipoDoc, a, conf, False) =
    convierte nomOrig nomDest tipoDoc

revisaArgumentos :: (String, String, String, Int, String, Bool) .
        (String, String, String, Int, String, Bool)
revisaArgumentos (nombreOrigen, nombreDestino, tipo, codigo, config,
    coment)
    | nombreOrigen ==nombreDestino =error "El nombre del archivo a
        transformar no debe ser igual al nombre del archivo destino"
    | otherwise =(nombreOrigen, nombreDestino, tipo, codigo, config,
        coment)

type Funcion = String  → Params  → Params  → String

extraeParams :: Params  → String
extraeParams []= ""
extraeParams (params:parametros) =(agregaLlaves params) ++(
    extraeParams parametros)
```

B.6. ParamsAux

```
module ParamsAux where

type Params =[String]
```

B.7. ParserAux

```
module ParserAux where
import ParamsAux
import JuntaListasConfig
import GeneradorInstrucciones
import System
import ConfigParser

buscarInsMon instruccion =do ins ← instruccion
                    buscarInstruccion ins

buscarInstruccion instruccion = buscarInstruccionAuxAux instruccion
    laTabla

buscarInstruccionAuxAux instruccion tabla =do tab ← tabla
                                return (buscarInstruccionAux instruccion
                                    tab)

buscarInstruccionAux :: Instruccion  → Tabla  → String
buscarInstruccionAux instruccion ((instruccionVieja , params,
    instruccionesNuevas):restoTabla) =
        if ((nombreInstruccion ==instruccionVieja) ∧
            ((length parametros) ==params))
        then (transforma instruccionesNuevas parametros)
```

```
        else (buscarInstruccionAux instruccion restoTabla)
    where
        nombreInstruccion =daNombreInstruccion instruccion
        parametros =daParamsInstruccion instruccion

buscarInstruccionAux instruccion []= mensajeError instruccion

mensajeError :: Instruccion  → String
mensajeError instruccion = "%% EmTeX2LaTeX: Instrucción no
    encontrada
    en tabla de equivalencias, se transcribe literalmente %%\n" ++
    (funcion nombre params params)
                where funcion =daFuncion instruccion
                      nombre =daNombreInstruccion instruccion
                      params =daParamsInstruccion instruccion

transforma :: [Instruccion]  → Params  → String
transforma []_ = ""
transforma ((nombre,param,funcion):insNuev) params =(funcion nombre
     param params) ++(transforma insNuev params)

daNombreInstruccion ::Instruccion  → String
daNombreInstruccion (nombre, _, _) =nombre

daParamsInstruccion :: Instruccion  → Params
daParamsInstruccion (_, param, _) =param

laTabla :: IO Tabla
laTabla = do tablaParam ← armaTablaParametros
         tabla ←return (juntaListasConfig tablaParam tablaOriginal)
         return tabla
```

```
type Instruccion = (String, Params, Funcion)

type Tabla =[EntradaTabla]

daFuncion :: Instruccion  → Funcion
daFuncion (_, _, funcion) = funcion

armaTablaParametros =do codigo ← tablaCodigo
                        config  ← tablaConfig
                        return (juntaListasConfig codigo config)

tablaOriginal :: Tabla
tablaOriginal = [("pageno", 1, [("setcounter", ["page","#1"],
    generaCadenaInstruccion)]),
        ("capit", 3, [("chapter", ["#2", "#3"],
            generaCadenaInstruccion),
                ("setcounter", ["chapter", "#1"],
                    generaCadenaInstruccion)]),
        ("vyv", 1, [("textsc", ["#1"], generaCadenaInstruccion)]),
        ("nota", 1, [("footnote", ["#1"], generaCadenaInstruccion)])
            ,
        ("bye", 0, []),
        ("it", 1, [("textit", ["#1"], generaCadenaInstruccion)]),
        ("sc", 1, [("textsc", ["#1"], generaCadenaInstruccion)]),
        ("bf", 1, [("textbf", ["#1"], generaCadenaInstruccion)]),
        ("tt", 1, [("texttt", ["#1"], generaCadenaInstruccion)]),
        ("sf", 1, [("textsf", ["#1"], generaCadenaInstruccion)]),
        ("sl", 1, [("textsl", ["#1"], generaCadenaInstruccion)]),
        ("break", 0, [("break", [], generaCadenaInstruccion)]),
        ("let", 0, [("let", [], generaCadenaInstruccion)]),
```

```
("null", 0, [("null", [], generaCadenaInstruccion)]),
(" leftline ", 1, [(" flushleft ", ["#1"],
        generaCadenaInstruccionBeginEnd)]),
(" rightline ", 1, [(" flushright ", ["#1"],
        generaCadenaInstruccionBeginEnd)]),
(" vfill ", 0, [(" vfill ", [], generaCadenaInstruccion)]),
("eject", 0, []),
("parindent", 1, [("parindent", ["#1"],
        generaCadenaInstruccionAsignacion)]),
("/", 0, [("/", [], generaCadenaInstruccion)]),
("'", 1, [("'", ["#1"], generaCadenaInstruccionSimple)]),
("TeX", 0, [("TeX", [], generaCadenaInstruccion)]),
("par", 0, [("par", [], generaCadenaInstruccion)]),
(" centerline ", 1, [("center", ["#1"],
        generaCadenaInstruccionBeginEnd)]),
("noindent", 0, [("noindent", [], generaCadenaInstruccion)]),
("-", 1, [("-", ["#1"], generaCadenaInstruccionSimple)]),
("thinspace", 0, [("thinspace", [], generaCadenaInstruccion)]
    ),
("hbox", 1, [("hbox", ["#1"], generaCadenaInstruccion)]),
("gi", 0, [("---", [], transcribe)]),
("S", 0, [("S", [], generaCadenaInstruccion)]),
("vskip", 0, [("vskip", [], generaCadenaInstruccion)]),
("dots", 0, [("dots", [], generaCadenaInstruccion)]),
("le", 0, [("le", [], generaCadenaInstruccion)]),
("hskip", 0, [("hskip", [], generaCadenaInstruccion)]),
("{", 0, [("{", [], generaCadenaInstruccion)]),
("}", 0, [("}", [], generaCadenaInstruccion)]),
("_", 0, [("_", [], generaCadenaInstruccion)]),
("#", 0, [("#", [], generaCadenaInstruccion)]),
("%", 0, [("%", [], generaCadenaInstruccion)]),
```

```
        ("&", 0, [("&", [], generaCadenaInstruccion)]),
        ("$", 0, [("$", [], generaCadenaInstruccion)]),
        ("vbox", 0, [("vbox", [], generaCadenaInstruccion)]),
        ("langle", 0, [("langle", [], generaCadenaInstruccion)]),
        ("rangle", 0, [("rangle", [], generaCadenaInstruccion)]),
        ("vert", 0, [("mid", [], generaCadenaInstruccion)]),
        ("vbox", 1, [("vbox", ["#1"], generaCadenaInstruccion)]),
        ("baselineskip", 0, [("baselineskip", [],
            generaCadenaInstruccion)]),
        ("copyright", 0, [("copyright", [], generaCadenaInstruccion)]
            ),
        ("~", 1, [("~", ["#1"], generaCadenaInstruccionSimple)]),
        ("item", 1, [("", [], generaCadenaItem)])]

type Funcion = String → Params → Params → String

type EntradaTabla =(String, Int, [Instruccion])

tablaCodigo =do tupla ← tuplaArgs
        tabla ← return (obtenTablaCodigo (obtenCodigo tupla))
        return tabla

tablaConfig = do tupla ← tuplaArgs
        tabla ← obtenTablaConfig (obtenConfig tupla)
        return tabla

tuplaArgs = do argumentos ← System.getArgs
            return (llenaTuplaArgumentos argumentos
                ("error", "", "book", 0, "", False))

obtenTablaCodigo codigo =case codigo of
```

```
                0  → []
                8859  → tabla8859
                850  → tabla850
                437  → tabla437

obtenCodigo (_,_,_,codigo,_,_) = codigo

obtenTablaConfig config = do case config of
                [] → return []
                nombre  → leeConfig nombre

obtenConfig (_,_,_,_,config,_) = config
```

Recibe un arreglo de cadenas (que será la lista de parámetrosdados desde la línea de comandos) y una tupla, donde deberán almacenarselos datos que contiene el arreglo de cadenas. Regresa una nueva tupla,donde se encuentra guardada la información contenida en el arreglode cadenas.

```
llenaTuplaArgumentos :: [String]  → (String, String, String, Int,
        String, Bool)  → (String, String, String, Int, String, Bool
            )
llenaTuplaArgumentos []tupla =tupla
llenaTuplaArgumentos ["-h"] tupla =("error", "","book",0,"", False)
llenaTuplaArgumentos (a:b:c:resto) tupla=
    case a of
        "-o"  → llenaTuplaArgumentos (c:resto) (
            agregaNombreOriginal tupla b)
```

```
        "-d"  → llenaTuplaArgumentos (c:resto) (
            agregaNombreDestino tupla b)
        "-c"  → llenaTuplaArgumentos (c:resto) (
            agregaTipoDocumento tupla b)
        "-f"  → llenaTuplaArgumentos (c:resto) (agregaConfig tupla b)
        "-a"  → llenaTuplaArgumentos (c:resto) (agregaCodigos tupla
            b)
        "-l"  → llenaTuplaArgumentos (b:c:resto) (
            marcaComentariosVerdadero tupla)
        _  → ("error", "", "book", 0, "",False)
llenaTuplaArgumentos (a:b:resto) tupla =
    case a of
        "-o"  → llenaTuplaArgumentos resto (agregaNombreOriginal
            tupla b)
        "-d"  → llenaTuplaArgumentos resto (agregaNombreDestino
            tupla b)
        "-c"  → llenaTuplaArgumentos resto (agregaTipoDocumento
            tupla b)
        "-f"  → llenaTuplaArgumentos resto (agregaConfig tupla b)
        "-a"  → llenaTuplaArgumentos resto (agregaCodigos tupla b)
        "-l"  → llenaTuplaArgumentos (b:resto) (
            marcaComentariosVerdadero tupla)
        _  → ("error", "", "book", 0, "",False)

marcaComentariosVerdadero (nomOrig, nomDest, tipoDoc, codigos,
    config, coment) =(nomOrig, nomDest, tipoDoc, codigos, config,
    True)

Agrega a la tupla que contiene todos los datos para llevar a cabo la
    transformación, el nombre del archivo a transformar
agregaNombreOriginal (nomOrig, nomDest, tipoDoc, codigos, config,
```

coment) nombre =(nombre, nomDest, tipoDoc, codigos, config, coment)

Agrega a la tupla que contiene todos los datos para llevar a cabo la transformación, el nombre del archivo de salida

agregaNombreDestino (nomOrig, nomDest, tipoDoc, codigos, config, coment) nombre =(nomOrig, nombre, tipoDoc, codigos, config, coment)

Agrega a la tupla que contiene todos los datos para llevar a cabo la transformación, los códigos de caracteres

agregaCodigos (nomOrig, nomDest, tipoDoc, codigos, config, coment) codigo =(nomOrig, nomDest, tipoDoc, (obtenCodigos codigo), config, coment)

Agrega a la tupla que contiene todos los datos para llevar a cabo la transformación, el tipo de documento que se desea generar

agregaTipoDocumento(nomOrig, nomDest, tipoDoc, codigos, config, coment) tipo =(nomOrig, nomDest, tipo, codigos, config, coment)

Agrega a la tupla que contiene todos los datos para llevar a cabo la transformación, el nombre del archivo de configuración

agregaConfig (nomOrig, nomDest, codOrig, codDest, config, coment) nombre =(nomOrig, nomDest, codOrig, codDest, nombre, coment)

```
tabla8859 = [("'", 1, [("", ["#1"], generaAcento8859)])]

tabla850 = [("'", 1, [("", ["#1"], generaAcento850)])]

tabla437 = [("'", 1, [("", ["#1"], generaAcento437)])]
```

```
generaAcento8859 :: String → Params → Params → String
generaAcento8859 nombre parametros (param:params) =
    case param of
        "A"  → "\193"
        "E"  → "\201"
        "I"  → "\205"
        "O"  → "\211"
        "U"  → "\218"
        "a"  → "\225"
        "e"  → "\233"
        "i"  → "\237"
        "o"  → "\243"
        "u"  → "\250"
        _  → "\\'" ++param

generaAcento850 :: String → Params → Params → String
generaAcento850 nombre parametros (param:params) =
    case param of
        "a"  → "\160"
        "i"  → "\161"
        "o"  → "\162"
        "u"  → "\163"
        "A"  → "\181"
        "I"  → "\214"
        "O"  → "\224"
        "U"  → "\233"
        "e"  → "\130"
        "E"  → "\144"
        _  → "\\'" ++param

generaAcento437 :: String → Params → Params → String
```

```
generaAcento437 nombre parametros (param:params) =
    case param of
        "a"  → "\160"
        "i"  → "\161"
        "o"  → "\162"
        "u"  → "\163"
        "e"  → "\130"
        "E"  → "\144"
        _  → "\\'" ++param

leeConfig nombre =do contenido ← readFile nombre
                  tabla  ← return (analizaConfig contenido)
                  return tabla
```

Recibe dos cadenas (que representan códigos de caracteres) yregresa un par para identificar los códigos que recibió como entrada,de acuerdo a la columna que ocupan en la tabla de conversión.

```
obtenCodigos :: String  → Int
obtenCodigos "8859" =8859
obtenCodigos "437" =437
obtenCodigos "850" =850
obtenCodigos _= error "código no válidos"

armaInsMon instruccion params funcion =do par ←
    cambiaListaIOAIOLista params
                    return (armaInstruccion instruccion par funcion
                        )

cambiaListaIOAIOLista ::[IO a]  → IO [a]
```

```
cambiaListaIOAIOLista (a:resto) =do elem ← a
                        concatIO (return [elem]) (
                            cambiaListaIOAIOLista resto)
cambiaListaIOAIOLista []=return []

armaInstruccion :: String → Params → Funcion → Instruccion
armaInstruccion instruccion params funcion =(instruccion, params,
    funcion)

concatIO cad1 cad2 =do uno ← cad1
            dos ← cad2
            return (uno ++dos)
```

B.8. Transformador

```
module Transformador where
import Parser

convierte nomOrig nomDest tipoDoc =do contenido ← readFile
    nomOrig
                        resultado ← analiza contenido
                                    res1 ← resultado
                        putStr "Parseo finalizado"
                        writeFile nomDest (agregaEncabezados
                            tipoDoc (res1 ++"\n\\end{document}"))
                        return ()

agregaEncabezados tipoDoc cadena ="\\documentclass{" ++tipoDoc
    ++"}\n
    \\begin{document}\n" ++cadena
```

Bibliografía

[1] Alfred V. Aho. *Compiladores. Principios, técnicas y herramientas.* Addison-Wesley Longman, 1998. Traducción al español de *Compilers. Principles, techniques and tools*, publicado en 1986.

[2] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, Abril 1978.

[3] Manuel M. T. Chakravarty and Gabriele Keller. An approach to fast arrays in Haskell. *Lecture Notes for The Summer School and Workshop on Advanced Functional Programming 2002*, pages 85–102, 2003.

[4] Chris Dornan, Isaac Jones, and Simon Marlow. Alex user guide. Disponible en http://www.haskell.org/alex/doc/html/alex.html, 2003.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] Daniel Friedman. *Essentials of Programming Languages.* The MIT Press, 2001.

[7] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs.... An Experiment in Software Prototyping Productivity. Disponible en http://www.haskell.org/papers/NSWC/jfp.ps, 4 de Julio 1994.

[8] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[9] Graham Hutton and Erik Meijer. Monadic parser combinator. Technical Report NOTICS-TR-96-4, Departement of Computer Science, University of Nottingham, 1996.

[10] Thomas Johnsson. Efficient Compilation of Lazy Evaluation. En *Proceedings of the ACM Conference on Compiler Construction*, págs. 58–69. ACM Press, Junio 1984.

[11] Donald E. Knuth. *The TEXbook.* Addison–Wesley, Reading, Massachusetts, tercera edición, 1986.

[12] Donald E. Knuth. *TEX:The Programm.* Addison–Wesley, Reading, Massachusetts, tercera edición, 1986.

[13] Helmut Kopka. *A Guide to LATEX. Document Preparation for Beginners and Advanced Users.* Addison–Wesley, 1999.

[14] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, Marzo 1966.

[15] John Launchbury. A natural semantics for lazy evaluation. En *Proc ACM Principles of Programming Languages.* ACM Press, Enero 1993.

[16] Gavin Maltby. An introduction to TEX and friends. Disponible en http://stommel.tamu.edu/~baum/tex/maltby-intro.ps, Noviembre 1992.

[17] Simon Marlow and Andy Gill. Happy user guide. Disponible en http://www.haskell.org/happy/doc/html/index.html, 2001.

[18] Jeff Newbern. All about monads. Disponible en http://www.nomaware.com/monads/html/index.html.

[19] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, Abril 2003.

[20] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. En John Launchbury, Erik Meijer, and Tim Sheard, editores, *Advanced Functional Programming*, volumen 1129 de *LNCS-Tutorial*, págs. 184–207. Springer-Verlag, 1996.

[21] Sebastian Sylvan. Why does Haskell matter? Disponible en http://www.haskell.org/complex/why_does_haskell_matter.html.

[22] Simon Thompson. Higher-order + Polymorphic = Reusable. Disponible en http://www.cs.ukc.ac.uk/pubs/1997/224, Mayo 1997.

[23] Simon Thompson. *Haskell. The Craft of Functional Programming.* Addison–Wesley, 1999.

[24] Simon Thompson and Steve Hill. Functional programming through the curriculum. En Pieter H. Hartel and Rinus Plasmeijer, editores, *Functional Programming Languages in Education*, número 1022 en Lecture Notes in Computer Science, págs. 85–102. Springer-Verlag, Diciembre 1995.

[25] Philip Wadler. Comprehending monads. En *1990 ACM Conference on Lisp and Functional Programming*, págs. 61–78. ACM Press, Junio 1990.

[26] Philip Wadler. Monads for functional programming. En *Marktoberdorf Summer School on Program Design Calculi*, volumen 118 de *NATO ASI Series F: Computer and Systems Sciences.* Springer-Verlag, Agosto 1992.

[27] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 23(3):240–263, Septiembre 1997.

[28] Jon Warbrick. Essential LaTeX. Disponible en http://stommel.tamu.edu/~baum/tex/essential.ps, 1992.

[29] Glynn Winskel. *The Formal Semantics of Programming Languages. An Introduction.* The MIT Press, 1993.

[30] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.