116
2y

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

Simplificación de Expresiones Regulares

T E S I S

Que para obtener el título de

INGENIERO EN COMPUTACION

P r e s e n t a

Alejandro Augusto Rafael Trejo Ortiz

Director de Tesis: Guillermo Fernández Anaya

México, D F. 1996

TESIS CON
FALLA DE ORIGEN

PARA MIS PADRES

Índice General

Índice de Tablas

1. Introducción

Los conjuntos regulares tienen un papel central en la teoría de lenguajes. Los lenguajes denotados por estos conjuntos se conocen como **lenguajes regulares o de tipo 3.** Existen diversos métodos para especificar tales lenguajes, entre los cuales se destacan las expresiones regulares por ser una notación compacta y precisa. Sin embargo, al utilizar expresiones regulares como descriptores de lenguajes es frecuente toparse con fórmulas muy extensas. En tales casos se puede recurrir a las identidades de las expresiones regulares para la simplificación de dichas fórmulas.

Desafortunadamente, en muchas situaciones la manipulación "algebraica" de las expresiones regulares se convierte en una labor difícil, tediosa, propensa a errores y gran consumidora de tiempo, lo cual, dada la gran utilidad que tienen estas expresiones en diversas áreas de las ciencias de la computación, como ocurre en la teoría de los compiladores e incluso en el diseño de circuitos secuenciales (Brzozowski [2]), hace deseable el empleo de un método automático de simplificación. Este trabajo describe los procesos de investigación, análisis y diseño que condujeron a la creación de tal método.

1.1 Definición del Problema

El problema de simplificar expresiones regulares es, en lo que hace a su aspecto formal, similar al de simplificar otros objetos matemáticos. Por ello, y con el fin de obtener algunas luces que nos ayuden a limitar nuestro trabajo, resumimos a continuación algunas ideas relativas al problema general de la simplificación.

1.1.1 El problema general de la simplificación

Segun Buchberger [3], el problema de la simplificación tiene dos aspectos:

- obtener objetos equivalentes pero más simples (simplificación no-canónica).
- calcular representaciones únicas de objetos equivalentes (simplificación canónica).

Simplificación no canónica

La simplificación no-canónica, que se basa en "varios criterios intuitivos de simplicidad" (Buchberger [3]), como pueden ser la longitud de las expresiones, el significado para el usuario, la existencia o no de factorización, la cantidad de memoria necesaria para su representación, etc. (Sammet [11]), implica dos problemas:

- el concepto de "simplicidad" no tiene una adecuada definición (Sammet [11]).
- idealmente, se esperaría encontrar la expresión equivalente más simple (Geddes [5]).

Este último problema presenta algunas sutilezas, en palabras de Geddes [5]:

> El problema de especificar algorítmicamente la forma más simple para una expresión determinada es un problema muy difícil. Por ejemplo, cuando manipulamos polinomios del dominio $\mathbf{Z}[x, y]$ podemos demandar que todos los polinomios estén completamente expandidos (con los términos semejantes combinados apropiadamente), en cuyo caso la expresión
>
> $$(12x^2y - 4xy + 9x - 3) - (3x - 1)(4xy + 3)$$
>
> sería representada como el polinomio cero.
>
> Sin embargo, considérese la expresión
>
> $$(x + y)^{1000} - y^{1000}$$
>
> La forma expandida de este polinomio contendra mil términos y ya sea desde el punto de vista de ingeniería o desde las consideraciones de recursos de computadora, esta expresión será considerada "más simple" tal como está que en su forma expandida. Similarmente, la expresión
>
> $$x^{1000} - y^{1000}$$
>
> que está en forma expandida es "más simple" que su forma factorizada correspondiente en la cual $(x - y)$ es factor.

Más aún, al "no existir algoritmo que pueda incluir todas las reglas matemáticas" como afirma Korsvold [10], no se puede garantizar que se puede encontrar la forma más simple para diferentes tipos de expresiones. Esta es la misma opinión de Cameron [4] cuando dice que

> ... no existe conjunto finito de reglas de simplificación que pueda llevar a cabo todas las simplificaciones posibles

de una amplia gama de expresiones. Esto también significa que no existe algoritmo que pueda simplificar completamente una expresión arbitraria de un tipo dado.

Simplificación canónica

Con la simplificación canónica, sin embargo, se resuelven ambos problemas (Buchberger [3]):

> Un simplificador canónico trivialmente define una noción correspondiente de simplicidad: la forma canónica de un objeto puede ser llamada más simple que el objeto mismo.

Otro hecho en extremo importante es que expresiones equivalentes al someterse a un proceso de simplificación canónico eventualmente devienen en la misma expresión estándar (o una muy parecida), lo cual representa grandes ventajas (Cameron [4]).

Por estas razones, ya desde los primeros trabajos de simplificación "los investigadores reconocieron la necesidad de una forma canónica para las expresiones y proporcionaron la capacidad de producir y manejar dicha forma" (Sammet [11]), sin que ello significara el abandono de las simplificaciones no-canónicas. Por el contrario, Buchberger [3] afirma que "los procedimientos de simplificación no-canónicos son de gran importancia práctica en los sistemas de álgebra de computadora".

1.1.2 La simplificación de expresiones regulares

Desafortunadamente, en el dominio de las expresiones regulares no se cuenta con una forma canónica (Hunt [8]), por lo que todo intento de simplificación de estas expresiones debera restringirse al ámbito de la simplificación no-canónica. La afirmación anterior significa que, si emprendemos la tarea de crear un simplificador de expresiones regulares, debemos dar respuesta a los dos problemas propios de la simplificación no-canónica, y que ya fueron referidos en la subsección anterior.

Podemos intentar responder tales interrogantes, diciendo que, en lo que concierne a la definición del concepto de simplicidad, es posible establecer que una expresión regular está en forma simplificada si ya no puede simplificarse más de acuerdo a un conjunto de reglas dado, es decir, habiendo aplicado todas las reglas pertinentes en una expresión dada, ésta tendrá que estar en forma simplificada. Más adelante, en el apartado 2.3, volveremos a hacer mención de esto.

Sin embargo, resolver el otro aspecto, es decir, el relativo a encontrar la forma mínima o más simple de estas expresiones, está fuera de nuestro alcance. En primer lugar, tenemos la evidencia, ya mencionada, de que para un gran número de objetos matemáticos, no

existe algoritmo que simplifique totalmente una expresión arbitraria perteneciente a dicho dominio. Nada nos garantiza que en el dominio de las expresiones regulares este problema sea computable. Además, sabiendo que el problema de la equivalencia de expresiones regulares es NP-completo (vease Hunt [8]), podríamos esperar que probablemente aquél también lo sea.

En cualquier caso, es evidente que el hallar la forma mínima de las expresiones regulares, es un problema de profunda y prolongada investigación que rebasa, con mucho, nuestras posibilidades actuales.

1.2 Objetivo

Tomando en cuenta lo dicho hasta aqui, estamos en condiciones de afirmar que

> *el objetivo perseguido en la realización de este trabajo es desarrollar un programa de computadora que sea capaz de simplificar expresiones regulares comunes (es decir, aquellas que sólo contienen operadores de suma o unión, concatenación y cerradura) y en donde simplificar signifique que toda expresión regular obtenida como salida será menor (o, en el peor de los casos, igual) en longitud a la expresión de entrada correspondiente.*

2. Métodos de Simplificación

Si bien el problema que nos compete es la simplificación de expresiones regulares, la mayor parte de los esfuerzos dedicados a la simplificación se han realizado en el dominio de las expresiones algebraicas.

2.1 Simplificación de expresiones algebraicas

Sammet [11] hace una excelente reseña de los métodos utilizados por los creadores de los primeros sistemas de manipulación de fórmulas, y puesto que las "técnicas básicas de los trabajos pioneros todavía se usan", como afirma Buchberger [3], transcribimos a continuación un fragmento de tal documento.

> Uno de los tipos más importantes (y más comunes) de simplificación involucra el manejo de términos y factores que incluyen cero y uno. La frase eliminación de idénticos se presenta aqui como abreviacion de las transformaciones $A + 0 \mapsto A$, $A * 0 \mapsto 0$, $A * 1 \mapsto A$ (donde se suponen asociatividad y conmutatividad). Si también se realizan las transformaciones $A^0 \mapsto 1$, $A^1 \mapsto A$, $1^A \mapsto 1$ entonces se dice que se efectúa la eliminación completa de idénticos.
>
> Un trabajo significante y pionero en el área de simplificación fue realizado en Darmouth ... la fórmula era almacenada internamente en forma de árbol ... El grupo de Dartmouth omitía de sus expresiones a los cocientes, constantes no enteras y cualquier función trascendental excepto la exponencial. Su forma canónica consistía de una suma de productos, cada producto conteniendo a lo sumo tres factores: una constante, una sola variable elevada a una potencia, y una función exponencial que contenía sólo esa variable. Su rutina de simplificación consistia de dos partes, la primera ponía la fórmula en esta forma estándar pero no hacía ninguna simplificación. Esto lo lograba mediante iteraciones, y en particular cada vez que se topaba con un producto que tuviese una suma como uno de sus argumentos, se aplicaba la ley distributiva. La segunda parte de la rutina de simplificación de Dartmouth efectuaba realmente la simplificación de sumas y combinaba los términos semejantes. Se mejoraba la eficiencia marcando las partes de las expresiones que se consideraban

todavía relevantes, de tal suerte que se dejara de prestar atención a los componentes que ya no eran importantes.

! → Un buen número de rutinas de simplificación fueron escritas en LISP. La primera, descrita en la tesis de Goldberg sobre análisis de redes, parece haber sido sugerida por Rochester, y también fue usada por Edwards en su tesis. El esquema general usaba una rutina que primero simplificaba cada argumento de una expresión algebraica y despues llamaba al programa de simplificación apropiado de entre los existentes, a saber, uno por cada operador. Estos eran los siguientes: MAS, MENOS, MULT, RECIP, y POTEN. Ademas se combinaban los términos y productos semejantes y se efectuaba la eliminación completa de idénticos. Una función ORDENA ponía los elementos de la lista en forma canónica para acelerar el proceso. Se disponía de funciones especiales para quitar paréntesis en un producto de sumas, factorizar un sólo término de una suma y para eliminar términos de una suma si eran más pequeños que algún valor asignado.

Otro programa de simplificación en LISP fue escrito por Hart en 1961. Aqui las expresiones se ponían en forma estándar. Se realizaban las siguientes operaciones sobre ellas: combinación de términos numéricos, poner los términos repetidos bajo un operador de multiplicación, factorizar números y subexpresiones comunes, encontrar un denominador común, factores repetidos convertirlos en una expresión potencia, reunir signos negativos, cancelar términos de la forma Y/Y, cambiar signos, dividir por números, y efectuar eliminación completa de idénticos.

Un programa más ambicioso en LISP, escrito por Wooldrige, fue diseñado para usarse en línea. Este programa realizaba eliminación completa de idénticos y combinaba términos y productos semejantes, así como cancelación de inversos. Los polinomios eran manipulados separadamente con el fin de aprovechar su forma especial. Los términos podían factorizarse especificando las variables. El usuario también especificaba exactamente cuando deseaba que se realizara una expansión(es decir, escribir un producto de sumas como una suma de productos). Esta rutina en particular fue también usada en el sistema MATHLAB.

Un trabajo muy interesante fue hecho por Martin. En vez de hacer la simplificación directamente, él desarrolló un esquema de codificación hash usando aritmética de campo finito, principalmente con el propósito de comparar expresiones para determinar su igualdad. Desafortunada-

mente, como lo admite el mismo Martin, el método es probabilístico y por lo tanto corre el riesgo de declarar dos expresiones diferentes como iguales. Puesto que la eliminación de idénticos es hecha en una forma directa, el uso principal de la codificación hash es encontrar subexpresiones equivalentes. Martin establece que este programa parece ser más rápido y poderoso que los que usan ordenamiento canónico, pero no esta claro sobre que bases hace esta afirmación.

Uno de los desarrollos en gran escala en simplificación fue el hecho en FORMAC por Tobey, R. Bobrow y Zilles. Puesto que la simplificación automática está incluida en un sistema mayor, no es una rutina que pueda usarse por sí misma. La filosofía básica fue, despues de la ejecución de cada comando, automáticamente efectuar eliminación de idénticos, combinar términos y productos semejantes, cancelar inversos, etc. El usuario disponía de comandos aparte para realizar expansiones y factorizaciones en relación a una sola variable o sus potencias. El método básico usado era primero aplicar un conjunto de transformaciones que hicieran eliminación completa de idénticos y cambiar signos negativos y paréntesis para reducir el número de tipos de expresiones que pudiesen aparecer. Éstas eran entonces enviadas a otra rutina que ponía todo en forma canónica y durante el proceso combinaba y cancelaba términos y factores. Debido a que FORMAC podía manejar expresiones de una naturaleza muy general, incluyendo funciones elementales y anidamientos de ellas, la forma canónica era, naturalmente, muy compleja. Las expresiones ya simplificadas eran marcadas de tal forma que no se manejaban dos veces; esto tiene algunas semejanzas con la técnica Dartmouth, pero no es la misma.

Desde un punto de vista conceptual, el uso de relaciones laterales en la simplificación es un desarrollo interesante. Una relación lateral (algunas veces conocida como identidad de dominio) es una proposición de igualdad de dos expresiones. El sistema usa automáticamente esta proposición para reemplazar un lado por el otro.

Un forma primitiva de esto fue implementada en ALPAK. En este sistema era posible especificar un reemplazo de la forma $X^J = C$, donde $J >= 1$ y C es una función racional independiente de X. Sin embargo, las relaciones laterales no fueron implementadas de modo eficiente, aunque al menos funcionaba.

> Un intento más ambicioso fue desarrollado por Alpiar. Aquí es posible especificar una relación cualquiera; en otras palabras, cualesquier dos expresiones podían ser declaradas como iguales. El programa consideraba cada relación en turno en el orden en que habían aparecido en la entrada, e intentaría hallar ocurrencias del lado izquierdo de la relación en la expresión a simplificarse. Al encontrar una ocurrencia, el programa substituía el lado izquierdo por el lado derecho de la relación especificada por el usuario. Para determinar si la relación resultante después de la sustitución era más simple que antes, se aplicaban otras reglas. En general, estas reglas involucraban la longitud de la nueva expresión.

Knuth [9], a su vez, bosqueja un método mas. Partiendo de la representación de árbol general para fórmulas algebraicas, recomienda

> incluir un nuevo campo en cada nodo, que represente su coeficiente (para los sumandos) o su exponente (para los factores de un producto). Aplique identidades algebraicas, como reemplazar $\ln(u^v)$ por $v \ln u$, y elimine las operaciones de sustracción, división, exponenciación y negación cuando sea posible usando operaciones equivalentes de adición y multiplicación. Haga que la suma y el producto sean operaciones n-arias en vez de binarias; reuna términos semejantes ordenando a los operandos en orden de árbol; algunas sumas y productos ahora se reducirán a cero o a la unidad, presentándose, quizas, ulteriores simplificaciones. Otros ajustes, como reemplazar una suma de logaritmos por el logaritmo de un producto, se sugieren por sí mismos.

2.2 Discusión de los métodos

De los métodos apenas descritos podemos recoger tres observaciones:

1. La base de la simplificación de expresiones algebraicas es la existencia de varias reglas de simplificación para cada tipo de expresión matemática. Así, existen reglas de simplificación para la adición, la substracción, la multiplicación, etc.
2. En la mayor parte de los casos se hace simplificación canónica, es decir, las expresiones de entrada se convierten a forma canónica y en este proceso se reunen términos semejantes, se efectúa la eliminación completa de idénticos, etc.
3. Muchas de estas técnicas han sido implementadas en LISP.

Al extender estas observaciones al dominio de nuestro interés , podemos percatarnos de la existencia de diferentes tipos de expresiones regulares y de las reglas de simplificación asociadas a cada uno de estos tipos. Es decir, existen expresiones regulares de suma, de concatenación y de cerradura y al mismo tiempo, reglas para simplificar esas expresiones.

Por otro lado, si bien no existen formas canónicas para las expresiones regulares, creemos que es posible aplicar alguno de los métodos mencionados. En concreto, se trata del método Goldberg, el cual fue descrito por Sammet [11] y debidamente transcrito. Este método parece ser la formalización de la técnica que se emplea intuitivamente al simplificar "manualmente", lo cual hace que se pueda aplicar de manera muy general a una gran variedad de dominios. Además, el empleo de formas canónicas con esta técnica no es indispensable, como se vera más abajo. Esta última característica es la más importante para decidirnos a usar este método, puesto que carecemos de formas estándar en el dominio de las expresiones regulares, como ya fue apuntado anteriormente.

En cuanto a la observación referente a LISP como lenguaje de programación común en aplicaciones de computación simbólica, es un hecho que históricamente este lenguaje marcó un hito en esta vertiente de la computación; sin embargo, actualmente existe tal variedad de lenguajes y de tan diversas filosofías que sería una lástima permanecer atados a la rigidez de LISP.

2.3 El método a utilizar

La técnica que hemos seleccionado para nuestro sistema, y de la que Sammet [11] ha hecho una breve descripción, es conocida también como *método dentro-fuera* (Cameron [4]) En este método el primer paso es simplificar todas las subexpresiones de la expresión dada. Por ejemplo, la simplificación de la expresion regular

$$((a+a)\cdot(b+\varnothing))$$

comenzaría por simplificar las subexpresiones $(a+a)$ y $(b+\varnothing)$, cuyas formas simplificadas serian, a y b, respectivamente. Usando las subexpresiones simplificadas, el segundo paso es aplicar las reglas de simplificación a la expresión que resulta de sustituir las subexpresiones simplificadas en la expresión original, esto es,

$$((a+a)\cdot(b+\varnothing)) \tag{2.1}$$
$$(a+a)=a \tag{2.2}$$
$$(b+\varnothing)=b \tag{2.3}$$

Sustituyendo 2.2 y 2.3 en 2.1

$$((a) \cdot (b)) \tag{2.4}$$

y finalmente(puesto que ya no es posible simplificar más)

$$(a \cdot b) \tag{2.5}$$

Es importante señalar que con este método se particiona el conjunto de todas las expresiones en expresiones simples, las cuales no son susceptibles de simplificación, y expresiones compuestas. Estas últimas, en el caso de las expresiones regulares, se dividirán también en expresiones de suma, de concatenación y de cerradura, como ya fue apuntado arriba.

2.3.1 El algoritmo Simplif

Un algoritmo en seudocódigo para este método es el siguiente:

Entrada : Una expresión regular.

Salida : Una expresión simplificada equivalente a la expresión de entrada. Aquí, una expresión está en forma simplificada si ya no puede ser simplificada de acuerdo al conjunto de reglas dado.

```
simplif(expr)
BEGIN
 IF simple(expr) THEN simplif ← expr
 ELSE IF cerradura(expr) THEN
       simplif ← Haz-cerradura-simplif(simplif(operando(expr)))
 ELSE IF concat(expr) THEN
       simplif ← Haz-concat-simplif
           (simplif(operando1(expr), simplif(operando2(expr))))
 ELSE IF suma(expr) THEN
      simplif ← Haz-suma-simplif
           (simplif(operando1(expr), simplif(operando2(expr))))
END
```

Este algoritmo recursivo primero determina el tipo de la expresión regular en cuestión para después aplicar las reglas de simplificación pertinentes según el tipo de expresión. Las reglas de simplificación se implementan en las funciones `Haz-cerradura-simplif`, `Haz-concat-simplif` y `Haz-suma-simplif`. La recursividad se detiene al llegar a una expresión regular simple.

El uso de formas canónicas en este método reduce el conjunto de reglas de simplificación al disminuir el número de casos especiales que requieren reglas de simplificación adicionales, o en palabras de

Sammet [11] "acelera el proceso". Por supuesto, el no disponer de formas canónicas no impide la utilización del método.

Cameron [4] demuestra que este método, para un conjunto dado de reglas, efectúa sistemáticamente todas las simplificaciones posibles de acuerdo a dichas reglas.

2.3.2 Ventajas y desventajas del método

Las principales ventajas del *método dentro-fuera* sobre las otras técnicas mencionadas son las siguientes:

- Con este método es posible aplicar sistemáticamente todas las reglas disponibles en la expresión a simplificar.
- No requerimos de formas canónicas.
- Es muy fácil de entender e implementar en el dominio de las expresiones regulares debido a la naturaleza inductiva de éstas.

Este método tiene también desventajas:

- Puesto que el algoritmo del método es recursivo, puede objetarse que una vez implementado hará un excesivo uso de recursos de cómputo.
- Si bien pueden o no usarse formas estándar, el no hacerlo implica cuidar gran cantidad de detalles en la implementación de las reglas de simplificación, y ya que toda forma canónica es, entre otras cosas, una suerte de ordenamiento de las expresiones, al no estar éstas ordenadas se corre el riesgo de hacer simplificaciones diferentes, es decir, simplificar más una expresión con una forma determinada, que otra que, si bien es equivalente, tiene una apariencia distinta. Esto no implica, por supuesto, que se obtengan resultados incorrectos, sólo más o menos simplificados según la expresión de entrada.
- Como ya se mencionó, este método garantiza que se aplicarán todas las reglas suceptibles de aplicarse en una expresión; sin embargo, es posible que, para algunos casos, algunas reglas aplicadas previamente impidan la aplicación posterior de otras que, eventualmente, representarían una mayor simplificación.

3. Aplicación del Método

Una aplicación exitosa del método requiere del establecimiento de un modelo apropiado. En nuestro caso el modelo de las expresiones regulares se hace a partir de la técnica de ingeniería de software conocida como abstracción de datos, la cual implica la definición de una estructura de datos apropiada y un conjunto de operaciones o funciones sobre dicha estructura de datos.

3.1 El Tipo de Datos Abstracto Expresión Regular

De la definición de expresiones regulares(Hopcroft [7]), podemos decir que existen, básicamente, dos tipos de éstas:

1. expresiones regulares simples, por ejemplo, *abab*;
2. expresiones compuestas, formadas a partir de expresiones regulares simples y que pueden ser, a su vez, de tres tipos:
   (a) expresiones regulares de suma, por ejemplo, $S + T$;
   (b) expresiones regulares de concatenación, por ejemplo, $S \cdot T$;
   (c) expresiones regulares de cerradura, por ejemplo, R^*.

Las expresiones regulares de suma pueden representarse de la siguiente forma:

```
operando1 operador operando2
```

en donde operando1 y operador2 pueden ser expresiones regulares simples o compuestas y el operador es +.

Algo similar ocurre al representar expresiones regulares de concatenación, salvo que el operador es ·, es decir,

```
operando1 operador operando2
```

En cuanto a las expresiones regulares de cerradura, estas pueden ser

```
operando operador
```

y de manera semejante a las expresiones de suma y concatenación, operando puede ser expresion regular simple o compuesta y el operador es *.

3.1.1 Estructura de datos para las Expresiones Regulares

Desde esta óptica, la estuctura de datos que represente a las expresiones regulares debe tener los siguientes elementos:

1. El tipo de expresión regular, ya sea simple, de suma, de concatenación o de cerradura;
2. En caso de ser expresión regular simple, la cadena de caracteres;
3. Si se trata de expresiones regulares de suma y concatenación, operando1 y operando2;
4. Si se trata de expresión regular de cerradura, operando.

3.1.2 Las operaciones para expresiones regulares

Para manipular a las expresiones regulares, tal y como las hemos representado aqui, requerimos de un conjunto de operaciones, que a continuación se mencionan:

VariableP :	Determinar si la expresión regular es una expresión simple.
SumP :	Determinar si la expresión regular es una expresión de suma.
ConcatenationP :	Determinar si la expresión regular es una expresión de concatenación.
ClosureP :	Determinar si la expresión regular es una expresión de cerradura.
VoidP :	Determinar si la expresión regular es la expresión correspondiente al conjunto vacío.
EmptyP :	Determinar si la expresión regular es la expresión correspondiente a la cadena vacía.
Operand :	Seleccionar el operando de una expresión de cerradura.
Operand1 :	Seleccionar el operando1 de una expresión de suma o concatenación.
Operand2 :	Seleccionar el operando2 de una expresión de suma o concatenación.
MakeClosure :	Construir una expresión de cerradura.
MakeSum :	Construir una expresión de suma.
MakeConcatenation :	Construir una expresión de concatenación.
MakeVariable :	Construir una expresión simple.

3.2 Las reglas de simplificación

Aunado al tipo de datos abstracto Expresión Regular, y con el fin de realizar nuestro simplificador, requerimos de los conjuntos de reglas de simplificación para los diversos tipos de expresiones regulares.

Las tablas 3.1, 3.2 y 3.3 de la página 18 corresponden a las reglas de simplificación de las expresiones regulares (veáse Hopcroft [7], Aho [1]).

3.3 Poniendo todo en conjunción

Ahora estamos en condiciones de conjuntar el tipo de datos abstracto Expresión Regular, las reglas de simplificación de las expresiones regulares y el algoritmo simplif. Para no parecer repetitivos, esta labor la referiremos en el capítulo siguiente y en los archivos fuentes del Apéndice B.2. Sin embargo, es importante señalar aquí que si bien no tenemos formas canónicas, como ya ha sido ampliamente repetido, nosotros consideramos conveniente establecer ciertas convenciones que nos permitieran aligerar la tarea de Implementar las reglas de simplificación. Por ejemplo, en los casos donde la conmutatividad es válida, hemos optado por utilizar sólo una de las posibles fórmulas, convirtiendo los otros casos a esta forma elegida.

Por otro lado, es instructivo observar cómo reacciona el método ante diversas expresiones regulares de entrada. En general, la respuesta es la esperada, es decir, se obtienen expresiones simplificadas; sin embargo, no sucede así con expresiones de la forma $R+S+R$. Esta expresión es, en realidad, $((R+S)+R)$, si recordamos las leyes de asociatividad y precedencia de las expresiones regulares. Parecería lógico que el método simplificara esta expresión a $R+S$, pero no sucede así, la deja sin cambio, lo cual es consecuencia de la carencia de formas canónicas en nuestro dominio. Esto sucede porque, como es sabido, el método primero tratará de simplificar $R+S$, cuya forma simplificada es $R+S$. Después simplificará R [1], que tampoco es susceptible de simplificación y finalmente simplificará a la expresión de suma formada por la también expresión de suma $R+S$ y R, es decir, la expresión original. Pero $R+S$ es diferente de R y, por tanto, no se puede aplicar la regla $R+R \mapsto R$. De la misma forma, ninguna otra regla es aplicable.

Este problema podría resolverse si supiésemos que R [2] es sumando de la subexpresión $R+S$. En tal caso, simplemente podríamos sustituir esta expresión, es decir, la segunda R de la expresión original, por $\emptyset$ para aplicar luego la regla $\emptyset+R \mapsto R$. En otras palabras, la expresión

[1] La segunda R de la expresión original, claro está.
[2] Lo mismo que la nota anterior.

Tabla 3.1: Reglas para las expresiones de suma

$$\begin{aligned}
\varnothing + R &= R \\
R + \varnothing &= R \\
R + R &= R \\
\lambda + R \cdot R^* &= R^* \\
R + S &= S + R \\
P \cdot Q + P \cdot R &= P \cdot (Q + R) \\
P \cdot Q + R \cdot Q &= (P + R) \cdot Q
\end{aligned}$$

Tabla 3.2: Reglas para las expresiones de concatenación

$$\begin{aligned}
\varnothing \cdot R &= \varnothing \\
R \cdot \varnothing &= \varnothing \\
\lambda \cdot R &= R \\
R \cdot \lambda &= R \\
R^* \cdot R^* &= R^* \\
R \cdot R^* &= R^* \cdot R \\
P^* \cdot (Q \cdot P^*)^* &= (P + Q)^* \\
(P^* \cdot Q)^* \cdot P^* &= (P + Q)^*
\end{aligned}$$

Tabla 3.3: Reglas para las expresiones de cerradura

$$\begin{aligned}
\lambda^* &= \lambda \\
\varnothing^* &= \lambda \\
(R^*)^* &= R^* \\
(\lambda + R)^* &= R^* \\
(P^* + Q^*)^* &= (P + Q)^* \\
(P^* \cdot Q^*)^* &= (P + Q)^*
\end{aligned}$$

regular original sería transformada en primer lugar a $R + S + \varnothing$ y después a su forma simplificada final $R + S$. Esta es la aproximación que utilizamos al encarar este problema(véase Apéndice B.2.5).

4. Implementación

El desarrollo de nuestro sistema para simplificar expresiones regulares implicó los siguientes pasos:

1. Elección del lenguaje de programación.
2. Desarrollo de las representaciones adecuadas para el tipo de datos abstracto Expresión Regular en el lenguaje elegido.
3. Implementación del algoritmo simplif en el lenguaje seleccionado.
4. Construcción de módulos de entrada y salida.
5. Diseño del programa principal que coordine todos los módulos anteriores y proporcione la interfaz al usuario.

A continuación describimos como fueron cubiertos cada uno de estos requerimientos. Pueden encontrarse detalles adicionales en los archivos fuentes que se presentan en el Apéndice B.2.

4.1 Elección del Lenguaje

El sistema se desarrolló en el lenguaje de programación Pascal (específicamente Turbo Pascal 6.0), ya que este lenguaje posee una serie de características que facilitan la solución del problema tal y como lo hemos bosquejado. Entre dichas características podemos mencionar que el Pascal se pliega perfectamente a la noción de tipo de datos abstracto con su estricta verificación de tipos. Además, al disponer de registros (estructuras) variantes, la representación de la estructura de datos para las expresiones regulares es inmediata. El Pascal también nos proporciona muy buenas herramientas para el manejo de cadenas de caracteres y, al mismo tiempo, permite el uso de funciones recursivas, ambos aspectos muy importantes en esta aplicación. Por otra parte, al usar las instrucciones *Mark* y *Release*, se simplifica en gran medida la administración de memoria.

4.2 Representación de Expresión Regular como tipo de datos abstracto

El módulo `rExpressions` proporciona la implementación del tipo de datos abstracto Expresión Regular, formada a partir de la estructura variante `RegExprCell` y las funciones de acceso de datos `MakeClosure(expr1)`, `MakeSum(expr1, expr2)`, `MakeConcatenation(expr1,expr2)`, etcétera.

4.3 Implementación de Simplif

El módulo `Simplif` contiene el corazón de nuestro programa, y consiste en la implementación en Pascal del algoritmo Simplif, junto con sus funciones asociadas.

4.4 Entrada y Salida

Los módulos relacionados con la entrada y salida de las expresiones regulares se construyeron en tres etapas:

1. Definición de una gramática inambigua para las expresiones regulares.
2. Construcción de los módulos de entrada que convierten la notación inducida por la gramática a la representación interna de las expresiones regulares.
3. Elaboración de los módulos de salida que convierten las representaciones internas de las expresiones regulares a la notación adoptada para las mismas.

A continuación describimos cada una de ellas.

4.4.1 Gramática

La ambiguedad es un problema común de las gramáticas que usan operadores infijos(Aho [1]). Tal es el caso de la siguiente gramática para las expresiones regulares:

$$R ::= R + R \| R \cdot R \| R^* \| (R) \| a \| b$$

Afortunadamente, al adoptar las convenciones de precedencia y asociatividad de los operadores(véase Hopcroft [7], Aho [1]), podemos resolver tales ambiguedades. De esta forma, obtenemos la gramática inambigua para las expresiones regulares de la tabla 4.1 que se encuentra en la página 22.

4.4.2 Entrada

Ésta consiste, básicamente, de dos etapas: Análisis léxico y análisis sintáctico (parseo).

Análisis Léxico

Como es bien sabido, el analizador léxico o scanner recibe un flujo de caracteres de entrada, del cual obtiene una serie de tokens que serán, a su vez, la entrada del parser.

Los caracteres de entrada válidos en nuestro caso se incluyen en la tabla 4.2.

Ya que los teclados comunes no incluyen símbolos para el conjunto vacío ni tampoco para la cadena vacía, acordamos la siguiente convención: al conjunto vacío lo representaremos con el símbolo @ y a la

Tabla 4.1: Gramática

⟨expr⟩	::=	⟨sum⟩‖⟨term⟩
⟨term⟩	::=	⟨concat⟩‖⟨factor⟩
⟨factor⟩	::=	⟨simple⟩‖⟨closure⟩
⟨simple⟩	::=	⟨variable⟩‖⟨parentesis⟩
⟨sum⟩	::=	⟨expr⟩ + ⟨term⟩
⟨concat⟩	::=	⟨term⟩ · ⟨factor⟩
⟨closure⟩	::=	⟨simple⟩*
⟨variable⟩	::=	⟨alfanum⟩{⟨alfanum⟩}
		‖∅‖λ
⟨parentesis⟩	::=	(⟨expr⟩)
⟨alfanum⟩	::=	a‖b‖c‖...‖z‖
		A‖B‖C‖...‖Z‖
		0‖1‖2‖...‖9

Tabla 4.2: Caracteres de entrada

⟨mas⟩	::=	+
⟨asterisco⟩	::=	*
⟨punto⟩	::=	·
⟨parentesis-izquierdo⟩	::=	(
⟨parentesis-derecho⟩	::=	)
⟨blanco⟩	::=	**blank**
⟨nuevalinea⟩	::=	**newline**
⟨letra⟩	::=	a‖b‖c‖...‖z
		‖A‖B‖C‖...‖Z
		‖∅‖λ
⟨digito⟩	::=	0‖1‖2‖...‖9

cadena vacía la representaremos con el símbolo %.

Para efectuar el proceso de análisis léxico es necesaria la inclusión del símbolo de la concatenación, a pesar de que generalmente éste se omite en los textos de matemáticas discretas y teoría de compiladores. Por lo anterior, en nuestro sistema lo representamos con un · .

Los tokens son los mostrados en la tabla 4.3.

Tabla 4.3: Tokens para las expresiones regulares

⟨regular⟩	::=	⟨op-suma⟩
		‖⟨op-concatenacion⟩
		‖⟨op-cerradura⟩
		‖⟨paren-lzq⟩
		‖⟨paren-der⟩
		‖⟨variable⟩
		‖⟨void⟩
		‖⟨empty⟩
⟨op-suma⟩	::=	+
⟨op-concatenacion⟩	:=	·
⟨op-cerradura⟩	::=	*
⟨paren-izq⟩	::=	(
⟨paren-der⟩	::=	)
⟨variable⟩	::=	$a‖b‖c‖\ldots‖z$
		$‖A‖B‖C‖\ldots‖Z$
		$‖0‖1‖2‖\ldots‖9‖\{a‖\ldots z‖$
		$A‖\ldots‖Z‖0‖\ldots‖9\}$
⟨empty⟩	::=	λ
⟨void⟩	::=	$\emptyset$

El proceso de entrada de expresiones regulares es iniciado por el módulo `InputStreams`, el cual proporciona la abstracción de flujo de entrada que el analizador léxico propiamente dicho ha de usar.

El flujo de entrada se representa mediante un buffer llamado `CurrentChar` y un procedimiento denominado `Advance`, que lee el siguiente carácter de entrada. También se dispone de otra función cuyo objetivo es inicializar el flujo de entrada.

El análisis léxico de expresiones regulares, implementado en el módulo `LexicalAnalizer`, funciona esencialmente através del análisis case

del primer carácter de entrada. La interfaz que el analizador léxico proporciona al parser consiste de un buffer llamado `CurrentToken` que contiene el token actual, y un procedimiento **`Advance`** que avanza al siguiente token del flujo de entrada. El final del flujo de tokens se marca con un token especial, el `EndOfStream`.

Parser

Una vez que el analizador léxico determinó el flujo de tokens de la entrada, el parser se encarga de descubrir como pueden agruparse jerárquicamente esos tokens para reflejar la gramática.

Si bien puede diseñarse un parser que construya árboles de derivación reales, nosotros diseñamos un parser que construye las estructuras de datos que representan a las expresiones regulares, esto es, dada la expresión regular $a+b\cdot c$, nuestro parser regresa como salida la estructura de datos que resulta de una operación **`MakeSum`** de la expresión regular simple a y la expresión regular compuesta que resulta de la operación **`MakeConcatenation`** con las expresiones regulares simples b y c, como muestra la siguiente figura:

Algunos autores consideran apropiado, en el caso de las gramática de operadores, el empleo de la técnica denominada de parseo descendente-recursivo (véase Aho [1]), la cual es sumamente senci-lla y fácil de implementarse "a mano". Debido a estas razones, hemos recurrido a tal técnica. En este método, se hace un conjunto de procedimientos de parseo mutuamente recursivos, uno por cada regla de la gramática. Por tanto, en nuestro caso, tenemos los procedimientos **`ParseTerm`**, **`ParseSimple`**, **`ParseVariable`**, **`ParseFactor`**, **`ParseExpression`**, etcétera.

Es importante señalar que en la gramática mostrada en la tabla 4.1 hay dos reglas recursivas por la izquierda, a saber, ⟨term⟩ y ⟨expr⟩. Existen diversas maneras de enfrentar este problema, una de ellas es descrita en Aho [1] y consiste en transformar las reglas de la gramática de tal suerte que se elimine la recursividad por la izquierda. Otra aproximación propuesta por Holub [6] no transforma las reglas sino que toma en cuenta que la recursión debe eventualmente terminar en algún nivel. Para ejemplificar lo anterior, veamos el caso de `ParseTerm`, donde la solución al problema de recursividad por

la izquierda es reconocer que aunque una ⟨concat⟩ pueda tener un primer operando que sea a su vez una ⟨concat⟩, la recursión debe eventualmente terminar con un ⟨factor⟩ como el primer operando en algún nivel. De esta forma, al parsear un ⟨term⟩, primero parseamos un ⟨factor⟩. Esto nos proporciona el ⟨term⟩ en el nivel más bajo de la recursión. Si el siguiente token es ·, completamos el parseo de la ⟨concat⟩. Esto nos da el ⟨term⟩ en el siguiente nivel de la recursión. Este proceso se repite mientras el siguiente token en el flujo de entrada sea ·. De manera semejante se procede en `ParseExpression`. Estas ideas se concretan en el módulo `Parser` .

4.4.3 Salida

Las funciones de salida tienen como propósito fundamental invertir el proceso de parseo y darle cierto formateo elemental a las expresiones regulares. Su implementación se organizó en dos módulos: `OutputStreams` y `PrettyPrinter`.

En el módulo `OutputStreams` se definen e implementan las abstracciones de formateo básico, tales como añadir un espacio en blanco entre ciertos tokens consecutivos, ejecutar un salto de línea forzado, etcétera; mientras que el módulo `PrettyPrinter` efectúa en realidad el proceso inverso del parser.

4.5 Programa Principal

El programa principal está definido en el módulo `regexpr`. En éste, el loop más externo lee y procesa los comandos del usuario, los cuales pueden ser:

/ayuda : Proporciona información acerca del uso del sistema.

/info : Proporciona información de los realizadores.

/salir : Termina el programa y regresa al entorno del sistema operativo.

Asumimos que si la cadena introducida por el usuario no es antecedida por el símbolo "/", entonces es una expresión regular. Será cuestión de los módulos de entrada determinar si es una fórmula regular bien formada.

5. Resultados

La manera natural de probar nuestro sistema es através de diversos ejemplos, los cuales de muestran a continuación.

5.1 Ejemplos

$$\varnothing + R + \lambda + R \cdot R^* \mapsto \lambda + R + R \cdot R^* \tag{5.1}$$
$$\varnothing + R + P \cdot Q + P \cdot R \mapsto R + P \cdot Q + P \cdot R \tag{5.2}$$
$$\varnothing + R + P \cdot Q + R \cdot Q \mapsto R + P \cdot Q + R \cdot Q \tag{5.3}$$
$$\varnothing + R + \varnothing + R \mapsto R \tag{5.4}$$
$$\varnothing + R + R + \varnothing \mapsto R \tag{5.5}$$
$$\varnothing + R + \lambda \cdot R \mapsto R \tag{5.6}$$
$$\varnothing + R + R \cdot \lambda \mapsto R \tag{5.7}$$
$$\varnothing + R + R^* \cdot R^* \mapsto R + R^* \tag{5.8}$$
$$\varnothing + R + R \cdot R^* \mapsto R + R \cdot R^* \tag{5.9}$$
$$\varnothing + R + R^* \cdot R \mapsto R + R \cdot R^* \tag{5.10}$$
$$\varnothing + R + P^* \cdot (Q \cdot P^*)^* \mapsto R + (P + Q)^* \tag{5.11}$$
$$\varnothing + R + (P^* \cdot Q)^* \cdot P^* \mapsto R + (P + Q)^* \tag{5.12}$$
$$\varnothing + R + \lambda^* \mapsto \lambda + R \tag{5.13}$$
$$\varnothing + R + \varnothing^* \mapsto \lambda + R \tag{5.14}$$
$$\varnothing + R + (R^*)^* \mapsto R + R^* \tag{5.15}$$
$$\varnothing + R + (\lambda + R)^* \mapsto R + R^* \tag{5.16}$$
$$\varnothing + R + (P^* + Q^*)^* \mapsto R + (P + Q)^* \tag{5.17}$$
$$\varnothing + R + (P^* \cdot Q^*)^* \mapsto R + (P + Q)^* \tag{5.18}$$
$$\varnothing + \lambda + R \cdot (\lambda + R)^* \mapsto R^* \tag{5.19}$$
$$R + R + \lambda + R \cdot R^* \mapsto \lambda + R + R \cdot R^* \tag{5.20}$$
$$R + R + S + R \mapsto R + S \tag{5.21}$$
$$R + R + P \cdot Q + P \cdot R \mapsto R + P \cdot Q + P \cdot R \tag{5.22}$$
$$R + R + P \cdot Q + R \cdot Q \mapsto R + P \cdot Q + R \cdot Q \tag{5.23}$$
$$R + R + \varnothing \cdot R \mapsto R \tag{5.24}$$
$$R + R + R \cdot \varnothing \mapsto R \tag{5.25}$$
$$R + R + \lambda \cdot R \mapsto R \tag{5.26}$$

$$R + R + R \cdot \lambda \mapsto R \tag{5.27}$$
$$R + R + R^* \cdot R^* \cdot R + R^* \tag{5.28}$$
$$R + R + R \cdot R^* \mapsto R + R \cdot R^* \tag{5.29}$$
$$R + R + R^* \cdot R \mapsto R + R \cdot R^* \tag{5.30}$$
$$R + R + P^* \cdot (Q \cdot P^*)^* \mapsto R + (P + Q)^* \tag{5.31}$$
$$R + R + (P^* \cdot Q)^* \cdot P^* \mapsto R + (P + Q)^* \tag{5.32}$$
$$\lambda + R \cdot R^* + \varnothing + R \mapsto R^* + R \tag{5.33}$$
$$\lambda + R \cdot R^* + R + R \mapsto R^* + R \tag{5.34}$$
$$\lambda + R \cdot R^* + P \cdot Q + P \cdot R \mapsto R^* + P \cdot Q + P \cdot R \tag{5.35}$$
$$\lambda + R \cdot R^* + P \cdot Q + R \cdot Q \mapsto R^* + P \cdot Q + R \cdot Q \tag{5.36}$$
$$\lambda + R \cdot R^* + \varnothing \cdot R \mapsto R^* \tag{5.37}$$
$$\lambda + R \cdot R^* + R \cdot \varnothing \mapsto R^* \tag{5.38}$$
$$\lambda + R \cdot R^* + \lambda \cdot R \mapsto R^* + R \tag{5.39}$$
$$\lambda + R \cdot R^* + R \cdot \lambda \mapsto R^* + R \tag{5.40}$$
$$\lambda + R \cdot R^* + R^* \cdot R^* \mapsto R^* \tag{5.41}$$
$$\lambda + R \cdot R^* + R \cdot R^* \mapsto R^* + R \cdot R^* \tag{5.42}$$
$$\lambda + R \cdot R^* + R^* \cdot R \mapsto R^* + R \cdot R^* \tag{5.43}$$
$$\lambda + R \cdot R^* + P^* \cdot (Q \cdot P^*)^* \mapsto R^* + (P + Q)^* \tag{5.44}$$
$$\lambda + R \cdot R^* + (P^* \cdot Q)^* \cdot P^* \mapsto R^* + (P + Q)^* \tag{5.45}$$
$$\lambda + R \cdot R^* + \lambda^* \mapsto \lambda + R^* \tag{5.46}$$
$$\lambda + R \cdot R^* + \varnothing^* \mapsto \lambda + R^* \tag{5.47}$$
$$\lambda + R \cdot R^* + (R^*)^* \mapsto R^* \tag{5.48}$$
$$\lambda + R \cdot R^* + (\lambda + R)^* \mapsto R^* \tag{5.49}$$
$$\lambda + R \cdot R^* + (P^* + Q^*)^* \mapsto R^* + (P + Q)^* \tag{5.50}$$
$$\lambda + R \cdot R^* + (P^* \cdot Q^*)^* \mapsto R^* + (P + Q)^* \tag{5.51}$$
$$P \cdot Q + P \cdot R + \varnothing + R \mapsto P \cdot (Q + R) + R \tag{5.52}$$
$$P \cdot Q + P \cdot R + R + \varnothing \mapsto P \cdot (Q + R) + R \tag{5.53}$$
$$P \cdot Q + P \cdot R + R + R \mapsto P \cdot (Q + R) + R \tag{5.54}$$
$$P \cdot Q + P \cdot R + \lambda + R \cdot R^* \mapsto \lambda + P \cdot (Q + R) + R \cdot R^* \tag{5.55}$$
$$P \cdot Q + P \cdot R + P \cdot Q + P \cdot R \mapsto P \cdot (Q + R) \tag{5.56}$$
$$P \cdot Q + P \cdot R + P \cdot Q + R \cdot Q \mapsto P \cdot (Q + R) + R \cdot Q \tag{5.57}$$
$$P \cdot Q + P \cdot R + \varnothing \cdot R \mapsto P \cdot (Q + R) \tag{5.58}$$
$$P \cdot Q + P \cdot R + R \cdot \varnothing \mapsto P \cdot (Q + R) \tag{5.59}$$
$$P \cdot Q + P \cdot R + \lambda \cdot R \mapsto P \cdot (Q + R) + R \tag{5.60}$$
$$P \cdot Q + P \cdot R + R \cdot \lambda \mapsto P \cdot (Q + R) + R \tag{5.61}$$

$$P \cdot Q + P \cdot R + R^* \cdot R^* \mapsto P \cdot (Q + R) + R^* \tag{5.62}$$

$$P \cdot Q + P \cdot R + P^* \cdot (Q \cdot P^*)^* \mapsto P \cdot (Q + R) + (P + Q)^* \tag{5.63}$$

$$P \cdot Q + P \cdot R + (P^* \cdot Q)^* \cdot P^* \mapsto P \cdot (Q + R) + (P + Q)^* \tag{5.64}$$

$$P \cdot Q + P \cdot R + \lambda^* \mapsto \lambda + P \cdot (Q + R) \tag{5.65}$$

$$P \cdot Q + P \cdot R + \varnothing^* \mapsto \lambda + P \cdot (Q + R) \tag{5.66}$$

$$P \cdot Q + P \cdot R + (R^*)^* \mapsto P \cdot (Q + R) + R^* \tag{5.67}$$

$$P \cdot Q + P \cdot R + (\lambda + R)^* \mapsto P \cdot (Q + R) + R^* \tag{5.68}$$

$$P \cdot Q + P \cdot R + (P^* + Q^*)^* \mapsto P \cdot (Q + R) + (P + Q)^* \tag{5.69}$$

$$P \cdot Q + P \cdot R + (P^* \cdot Q^*)^* \mapsto P \cdot (Q + R) + (P + Q)^* \tag{5.70}$$

$$P \cdot Q + R \cdot Q + \varnothing + R \mapsto (P + R) \cdot Q + R \tag{5.71}$$

$$P \cdot Q + R \cdot Q + R + \varnothing \mapsto (P + R) \cdot Q + R \tag{5.72}$$

$$P \cdot Q + R \cdot Q + R + R \mapsto (P + R) \cdot Q + R \tag{5.73}$$

$$P \cdot Q + R \cdot Q + \lambda + R \cdot R^* \mapsto \lambda + (P + R) \cdot Q + R \cdot R^* \tag{5.74}$$

$$P \cdot Q + R \cdot Q + P \cdot Q + P \cdot R \mapsto (P + R) \cdot Q + P \cdot R \tag{5.75}$$

$$P \cdot Q + R \cdot Q + P \cdot Q + R \cdot Q \mapsto (P + R) \cdot Q \tag{5.76}$$

$$P \cdot Q + R \cdot Q + \varnothing \cdot R \mapsto (P + R) \cdot Q \tag{5.77}$$

$$P \cdot Q + R \cdot Q + R \cdot \varnothing \mapsto (P + R) \cdot Q \tag{5.78}$$

$$P \cdot Q + R \cdot Q + \lambda \cdot R \mapsto (P + R) \cdot Q + R \tag{5.79}$$

$$P \cdot Q + R \cdot Q + R \cdot \lambda \mapsto (P + R) \cdot Q + R \tag{5.80}$$

$$P \cdot Q + R \cdot Q + R^* \cdot R^* \mapsto (P + R) \cdot Q + R^* \tag{5.81}$$

$$P \cdot Q + R \cdot Q + P^* \cdot (Q \cdot P^*)^* \mapsto (P + R) \cdot Q + (P + Q)^* \tag{5.82}$$

$$P \cdot Q + R \cdot Q + (P^* \cdot Q)^* \cdot P^* \mapsto (P + R) \cdot Q + (P + Q)^* \tag{5.83}$$

$$P \cdot Q + R \cdot Q + \lambda^* \mapsto \lambda + (P + R) \cdot Q \tag{5.84}$$

$$P \cdot Q + R \cdot Q + \varnothing^* \mapsto \lambda + (P + R) \cdot Q \tag{5.85}$$

$$P \cdot Q + R \cdot Q + (R^*)^* \mapsto (P + R) \cdot Q + R^* \tag{5.86}$$

$$P \cdot Q + R \cdot Q + (\lambda + R)^* \mapsto (P + R) \cdot Q + R^* \tag{5.87}$$

$$P \cdot Q + R \cdot Q + (P^* + Q^*)^* \mapsto (P + R) \cdot Q + (P + Q)^* \tag{5.88}$$

$$P \cdot Q + R \cdot Q + (P^* \cdot Q^*)^* \mapsto (P + R) \cdot Q + (P + Q)^* \tag{5.89}$$

$$\varnothing \cdot R + \varnothing + R \mapsto R \tag{5.90}$$
$$\varnothing \cdot R \cdot R + \varnothing \mapsto \varnothing \tag{5.91}$$
$$\varnothing \cdot R + R + R \mapsto R \tag{5.92}$$
$$\varnothing \cdot R + \lambda + R \cdot R^* \mapsto R^* \tag{5.93}$$
$$\varnothing \cdot R + P \cdot Q + P \cdot R \mapsto P \cdot (Q + R) \tag{5.94}$$
$$\varnothing \cdot R + P \cdot Q + R \cdot Q \mapsto (P + R) \cdot Q \tag{5.95}$$
$$\varnothing \cdot R + \varnothing \cdot R \mapsto \varnothing \tag{5.96}$$
$$\varnothing \cdot R + R \cdot \varnothing \mapsto \varnothing \tag{5.97}$$
$$\varnothing \cdot R + \lambda \cdot R \mapsto R \tag{5.98}$$
$$\varnothing \cdot R + R \cdot \lambda \mapsto R \tag{5.99}$$
$$\varnothing \cdot R + R^* \cdot R^* \mapsto R^* \tag{5.100}$$
$$\varnothing \cdot R + P^* \cdot (Q \cdot P^*)^* \mapsto (P + Q)^* \tag{5.101}$$
$$\varnothing \cdot R + (P^* \cdot Q)^* \cdot P^* \mapsto (P + Q)^* \tag{5.102}$$
$$\varnothing \cdot R + \lambda^* \mapsto \lambda \tag{5.103}$$
$$\varnothing \cdot R + \varnothing^* \mapsto \lambda \tag{5.104}$$
$$\varnothing \cdot R + (R^*)^* \mapsto R^* \tag{5.105}$$
$$\varnothing \cdot R + (\lambda + R)^* \mapsto R^* \tag{5.106}$$
$$\varnothing \cdot R + (P^* + Q^*)^* \mapsto (P + Q)^* \tag{5.107}$$
$$\varnothing \cdot R + (P^* \cdot Q^*)^* \mapsto (P + Q)^* \tag{5.108}$$
$$(\lambda + \lambda \cdot (\lambda + R \cdot R^*))^* \mapsto R^* \tag{5.109}$$
$$(\lambda)^* \cdot 0 + \lambda \mapsto \lambda + 0 \cdot 0 \tag{5.110}$$
$$\lambda + 1^* \cdot 011^* \cdot (1^* \cdot 011)^*)^* \mapsto \lambda + 1^* \cdot 011^* \cdot (1 + 011)^* \tag{5.111}$$
$$\lambda + R \cdot R^* \mapsto R^* \tag{5.112}$$
$$0^* \cdot 1(\lambda + (0 + 1)0^* \cdot 1)^* \cdot (0 + 1) \cdot (00)^* + 0 \cdot (00)^* \mapsto 0^* \cdot 1 \cdot ((0 + 1) \cdot 0^* \cdot 1)^* \cdot (0 + 1) + 0) \cdot 00^* \tag{5.113}$$

$$\varnothing + R \cdot (R + \varnothing) + R + R \cdot (\lambda + R \cdot R^*) \cdot (P \cdot Q + P \cdot R) \mapsto R + R \cdot R^* \cdot (P \cdot (Q + R)) \tag{5.114}$$

$$\lambda^* \cdot \varnothing^* + (R^*)^* \cdot (\lambda + R)^* + \lambda \cdot (P \cdot Q + P \cdot R)^* \mapsto \lambda + R^* + (P \cdot (Q + R))^* \tag{5.115}$$

5.2 Discusión de resultados

- Todos los ejemplos muestran algún grado de simplificación, es decir, en cada uno de ellos el lado izquierdo de la expresión es mayor en longitud que su correspondiente lado derecho.

- Existen ejemplos en donde son notorias las consecuencias de no haber empleado formas canónicas. Tal es el caso de 5.1 y 5.33, en los cuales las expresiones de entrada son a todas luces equivalentes pero la expresión simplificada obtenida para 5.33 es más reducida.

- El ejemplo 5.113 está tomado de Hopcroft [7] y es interesante comparar la expresión simplificada que este autor presenta y que es la siguiente:

$$0^* \cdot 1 \cdot ((0+1) \cdot 0^* \cdot 1)^* \cdot (0+1) \cdot (00)^* + 0 \cdot (00)^* \tag{5.116}$$

  y la mostrada en 5.113.

  En este caso, nuestro sistema llevo al cabo un paso más en la simplificación al aplicar la regla $P \cdot Q + R \cdot Q \mapsto (P+R) \cdot Q$ a la subexpresión $(0+1) \cdot (00)^* + 0 \cdot (00)^*$.

- Por otro lado, el ejemplo 5.111 comprueba lo que se dijo anteriormente acerca de que algunas veces la aplicación de ciertas reglas de simplificación impide la utilización de otras que significarían mayores simplificaciones. El ejemplo mencionado posee la misma forma que el mostrado en 5.112; sin embargo, ya que el método primero simplifica las subexpresiones de una expresión, éste comenzará por intentar simplificar el término $(1^* \cdot (011)^*)^*$. Esta subexpresión es de la forma $(P^* \cdot Q^*)^*$ que, como se sabe, puede simplificarse a $(P+Q)^*$ o, para esta instancia en particular, $(1 + 011)^*$. Después de haber realizado esta transformación, el método ya no podrá aplicar la regla que se muestra en 5.112 y que sería más apropiada en este caso.

6. Conclusiones y comentarios

- En este trabajo hemos aplicado un método de simplificación que originalmente fue utilizado en expresiones algebraicas al dominio de las expresiones regulares.
- El método aplicado, al ser esencialmente recursivo, se adapta muy bien a la naturaleza de las expresiones regulares, ya que éstas se definen inductivamente.
- A pesar de no contar con formas canónicas para las expresiones regulares, hemos logrado un programa que realiza simplificaciones aceptables de dichas expresiones.
- Asi mismo, la carencia de formas canónicas causa que el problema de la simplificación se complique en gran medida, ya que entonces es necesario tener en cuenta muchos casos especiales de simplificación.
- Por otra parte, proponemos una gramática lnambigua para las expresiones regulares, la cual fue utilizada en la elaboración de nuestro sistema.
- Finalmente, esperamos que este trabajo pueda servir de base para el desarrollo de un sistema de mayores alcances.

Apéndice

A.1 Definiciones básicas

A.1.1 Cadenas, alfabetos y lenguajes

Un *símbolo* es una entidad abstracta que no se define formalmente, de la misma forma que punto y línea no se definen en geometría. Ejemplos de símbolos usados con frecuencia son las letras y los dígitos. Una *cadena* (o palabra) es una secuencia finita de símbolos yuxtapuestos. La *cadena vacía*, denotada por ϵ [1], es la cadena que consiste de cero símbolos.

Un *alfabeto* es un conjunto finito de símbolos. Un *lenguaje (formal)* es un conjunto de cadenas de símbolos de algún alfabeto.

A.1.2 Autómatas finitos

Autómatas finitos determinísticos

Formalmente denotamos un *autómata finito determinístico (DFA)* por una 5-tupla $(Q, \Sigma, \delta, q_0, F)$, donde Q es un conjunto finito de estados, Σ es un alfabeto de entrada finito, $q_0 \in Q$ es el estado inicial, $F \subseteq Q$ es el conjunto de estados finales, y δ es la función de transición que mapea $Q \times \Sigma$ a Q. Esto es, $\delta(q, a)$ es un estado para cada estado q y cada símbolo de entrada a.

Autómatas finitos no determinísticos

Formalmete denotamos un *autómata finito no determinístico (NFA)* por una 5-tupla $(Q, \Sigma, \delta, q_0, F)$, donde Q, Σ, q_0, y F (estados, entradas, estado inicial, y estados finales) tienen el mismo significado que para un DFA, pero δ es un mapeo de $Q \times \Sigma$ a 2^Q.

Autómatas finitos no determinísticos con transiciones ϵ

Se define formalmente un *autómata finito no determinístico con transiciones* ϵ como una quíntupla $(Q, \Sigma, \delta, q_0, F)$ con todos los componentes como antes, pero δ , la función de transición, mapea $Q \times (\Sigma \cup \{\epsilon\})$ a 2^Q.

[1] En ocasiones también denotada por λ.

Lenguajes regulares

De manera general, se dice que una cadena x es aceptada por un autómata finito $M = (Q, \Sigma, \delta, q_0, F)$ si $\delta(q_0, x) = p$ para alguna p en F. El lenguaje aceptado por M, designado $L(M)$, es el conjunto $\{x|\delta(q_0, x) \in F\}$. Un lenguaje es un conjunto regular (o sólo regular) si es el conjunto aceptado por algún autómata finito.

Expresiones regulares

Sean Σ un conjunto finito de símbolos y L, L_1 y L_2 conjuntos de cadenas de Σ^*. La concatenación de L_1 y L_2, denotada L_1L_2, es el conjunto $\{xy|x \in L_1, y \in L_2\}$. Esto es, las cadenas en L_1L_2 se forman al elegir una cadena de L_1 y agregarle una cadena de L_2, en todas laas combinaciones posibles. Definimos $L^0 = \{\epsilon\}$ y $L^i = LL^{i-1} \forall i \geq 1$. La cerradura de Kleene (o sólo cerradura) de L, denotada L^*, es el conjunto

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Sea Σ un alfabeto. Las *expresiones regulares* sobre Σ y los conjuntos que ellos denotan se definen recursivamente como sigue:

1. $\varnothing$ es una expresión regular y denota el conjunto vacío.
2. ϵ es una expresión regular y denota el conjunto $\{\epsilon\}$.
3. Para cada a en Σ, a es una expresión regular y denota al conjunto $\{a\}$.
4. Si r y s son expresiones regulares denotando los lenguajes R y S, respectivamente, entonces $(r + s), (rs)$ y (r^*) son expresiones regulares que denotan a los conjuntos $R \cup S, RS$, y R^*, respectivamente.

Al escribir expresiones regulares es posible omitir paréntesis si se supone que * tiene mayor precedencia que la concatenación o +, y que la concatenación tiene mayor precedencia que +. Asf mismo, se puede suponer la asociatividad por la izquierda de los tres operadores (Hopcroft [7], Aho [1]).

La equivalencia de DFA's y NFA's

Hopcroft [7] prueba que para cada NFA es posible construir un DFA equivalente, o lo que es lo mismo, si L es un conjunto aceptado por un autómata finito no determinístico entonces existe un autómata finito determinístico que acepta a L.

Minimización de autómatas finitos

También Hopcroft [7] demuestra que para cada conjunto regular existe un único DFA (salvo isomorfismo) con un mínimo número de estados. El mismo autor proporciona un algoritmo para obtener tal DFA mínimo y comenta la existencia de otros algoritmos con el mismo propósito.

B.2 Archivos Fuentes

B.2.1 Módulo rExpressions

```
UNIT rExpressions;

(*  En este modulo se implemento
    el tipo de datos abstracto Expresion Regular. *)

INTERFACE

TYPE
  StringPointer = ^ STRING;
  RegExprType = ^ RegExprCell;
  rExprKind = (Variable, Closure, Concatenation, Sum);
  RegExprCell =
    RECORD
      CASE exprKind : rExprKind OF
        Variable : (varName : StringPointer);
        Closure : (operand : RegExprType);
        Concatenation, Sum: (operand1, operand2 : RegExprType)
    END;

CONST

  TheUndefinedValue : RegExprType = NIL;

VAR TheEmptyString : RegExprType;
    TheVoidSet : RegExprType;

FUNCTION VariableP(rExpr : RegExprType) : BOOLEAN;

FUNCTION SymbolString(rExpr : RegExprType) : StringPointer;

FUNCTION MakeVariable(x : STRING) : RegExprType;

FUNCTION ClosureP(rExpr : RegExprType) : BOOLEAN;

FUNCTION MakeClosure(rExpr : RegExprType) : RegExprType;

FUNCTION ConcatenationP(rExpr : RegExprType) : BOOLEAN;

FUNCTION MakeConcatenation(rExpr1, rExpr2 : RegExprType ) :
                                                RegExprType;

FUNCTION SumP(rExpr : RegExprType) : BOOLEAN;
```

```
FUNCTION MakeSum(rExpr1, rExpr2 : RegExprType ) : RegExprType;

FUNCTION Operand(exp : RegExprType) : RegExprType;

FUNCTION Operand1(exp : RegExprType) : RegExprType;

FUNCTION Operand2(exp : RegExprType) : RegExprType;

FUNCTION Equal(expr1, expr2 : RegExprType) : BOOLEAN;

FUNCTION EmptyP(expr : RegExprType) : BOOLEAN;

FUNCTION VoidP(expr : RegExprType) : BOOLEAN;

IMPLEMENTATION

FUNCTION VariableP(rExpr : RegExprType) : BOOLEAN;

BEGIN
  VariableP := rExpr^.exprKind = Variable
END;


FUNCTION ClosureP(rExpr : RegExprType) : BOOLEAN;

BEGIN
  ClosureP := rExpr^.exprKind = Closure
END;


FUNCTION ConcatenationP(rExpr : RegExprType) : BOOLEAN;

BEGIN
  ConcatenationP := rExpr^.exprKind = Concatenation
END;


FUNCTION SumP(rExpr : RegExprType) : BOOLEAN;

BEGIN
  SumP := rExpr^.exprKind = Sum
END;

FUNCTION SymbolString(rExpr : RegExprType) : StringPointer;

BEGIN
```

```
  SymbolString := rExpr^.varName
END;

FUNCTION MakeVariable(x : STRING) : RegExprType;

VAR newExpr : RegExprType;

BEGIN
  NEW(newExpr);
  WITH newExpr^ DO
    BEGIN
      exprKind := Variable;
      GetMem(varName, Length(x) + 1);
      varName^ := x
    END;
  MakeVariable := newExpr
END;


FUNCTION Operand(exp : RegExprType) : RegExprType;

BEGIN
  Operand := exp^.operand
END;


FUNCTION Operand1(exp : RegExprType) : RegExprType;

BEGIN
  Operand1 := exp^.operand1
END;

FUNCTION Operand2(exp : RegExprType) : RegExprType;

BEGIN
  Operand2 := exp^.operand2
END;


FUNCTION MakeClosure(rExpr : RegExprType) : RegExprType;

VAR newExpr : RegExprType;

BEGIN
  NEW(newExpr);
  WITH newExpr^ DO
```

```
    BEGIN
      exprKind := Closure;
      operand := rExpr
    END;
  MakeClosure := newExpr
END;

FUNCTION MakeConcatenation(rExpr1, rExpr2 : RegExprType) :
                                                RegExprType;

VAR newExpr : RegExprType;

BEGIN
  NEW(newExpr);
  WITH newExpr^ DO
    BEGIN
      exprKind := Concatenation;
      operand1 := rExpr1;
      operand2 := rExpr2
    END;
  MakeConcatenation := newExpr
END;

FUNCTION MakeSum(rExpr1, rExpr2 : RegExprType) :
                                        RegExprType;

VAR newExpr : RegExprType;

BEGIN
  NEW(newExpr);
  WITH newExpr^ DO
    BEGIN
      exprKind := Sum;
      operand1 := rExpr1;
      operand2 := rExpr2
    END;
  MakeSum := newExpr
END;

FUNCTION Equal(expr1, expr2 : RegExprType) : BOOLEAN;

BEGIN
  IF (VariableP(expr1)) AND (VariableP(expr2)) THEN
    Equal := SymbolString(expr1)^ = SymbolString(expr2)^
  ELSE IF (ClosureP(expr1)) AND (ClosureP(expr2)) THEN
    Equal :=Equal(Operand(expr1), Operand(expr2))
```

```
  ELSE IF (SumP(expr1)) AND (SumP(expr2)) THEN
    Equal := ((Equal (Operand1(expr1), Operand1(expr2)))
          AND (Equal(Operand2(expr1), Operand2(expr2))))
          OR ((Equal(Operand1(expr1), Operand2(expr2)))
          AND (Equal(Operand2(expr1), Operand1(expr2))))
  ELSE IF (ConcatenationP(expr1)) AND
                          (ConcatenationP(expr2)) THEN
    Equal := (Equal(Operand1(expr1), Operand1(expr2)))
           AND (Equal(Operand2(expr1), Operand2(expr2)))
  ELSE Equal := FALSE
END;

FUNCTION EmptyP(expr : RegExprType) : BOOLEAN;

BEGIN
  IF VariableP(expr) THEN
     EmptyP := SymbolString(expr)^ = '%'
  ELSE EmptyP := FALSE
END;

FUNCTION VoidP(expr : RegExprType) : BOOLEAN;

BEGIN
  IF VariableP(expr) THEN
     VoidP := SymbolString(expr)^ = '@'
  ELSE VoidP := FALSE
END;

BEGIN
  NEW(TheEmptyString);
  NEW(TheVoidSet);
  WITH TheEmptyString^ DO
    BEGIN
      exprKind := Variable;
      GetMem(varName, Length('%') + 1);
      varName^ := '%'
    END;
  WITH TheVoidSet^ DO
    BEGIN
      exprKind := Variable;
      GetMem(varName, Length('@') + 1);
      varName^ := '@'
    END
END .
```

B.2.2 Módulo InputStreams

```
UNIT InputStreams;

(* Este modulo proporciona la abstraccion
   del flujo de entrada usada por el
   analizador lexico y el parser.          *)

INTERFACE

  USES Auxiliar;

CONST
  EndOfLineChar = #13;
  EndOfFileChar = #04;

TYPE
  InputStream =
    OBJECT
      CurrentLine : STRING;
      LineNo, CharPos : INTEGER;
      FUNCTION CurrentChar : CHAR;
      PROCEDURE Advance; VIRTUAL;
      CONSTRUCTOR Init(line1 : STRING);
    END;

  InputStreamPtr = ^ InputStream;

IMPLEMENTATION


FUNCTION InputStream.CurrentChar : CHAR;

BEGIN
  CurrentChar := CurrentLine[CharPos]
END;


PROCEDURE InputStream.Advance;

BEGIN
  IF CharPos < Length(CurrentLine) THEN
                        CharPos := CharPos + 1
  ELSE
    BEGIN
      LineNo := LineNo + 1;
```

```
      CharPos := 1;
      IF EOF(input) THEN
                  CurrentLine := EndOfFileChar
      ELSE
        BEGIN
          RdLine(CurrentLine);
          CurrentLine := CurrentLine + EndOfLineChar
        END
    END
END;


CONSTRUCTOR InputStream.Init(line1 : STRING);

BEGIN
  CharPos := 1;
  LineNo := 1;
  CurrentLine := line1 + EndOfLineChar
END;

END .
```

B.2.3 Módulo LexicalAnalyzer

```
UNIT LexicalAnalyzer;

(*  En este modulo  esta implementado el analizador lexico
    para las expresiones regulares.  *)

INTERFACE

  USES InputStreams;

TYPE
  TokenKind =
    (PlusOp, ConcatenationOp, ClosureOp,
    LeftParen, RightParen, VariableSymbol,
    EndOfStream, BadToken);

(* La siguiente funcion regresa el nombre del token
   como un string, se usa en mensages de error.     *)

FUNCTION TokenName(itsKind : TokenKind) : STRING;

TYPE
  TokenStream =
```

```
    OBJECT
      charStream : InputStreamPtr;
      CurrentToken :
        RECORD
          TokenPos : INTEGER;
          CASE kind : TokenKind OF
            VariableSymbol : (SymbolString : STRING)
        END;
      BadTokenMessage : STRING[72];
      FUNCTION CurrentTokenKind : TokenKind; VIRTUAL;

      FUNCTION CurrentSymbolString : STRING; VIRTUAL;
      FUNCTION CurrentTokenLine : INTEGER;
      FUNCTION CurrentTokenPosition : INTEGER;
      PROCEDURE Advance; VIRTUAL;
      PROCEDURE Terminate;
      CONSTRUCTOR Init(stream : InputStreamPtr);
    END;
  TokenStreamPtr = ^ TokenStream;
  InteractiveTokenStream =
    OBJECT (TokenStream)
      CurrentTokenNotSet : BOOLEAN;
      ParenLevel : INTEGER;

      FUNCTION CurrentTokenKind : TokenKind; VIRTUAL;

      FUNCTION CurrentSymbolString : STRING; VIRTUAL;
      PROCEDURE Advance; VIRTUAL;
      CONSTRUCTOR Init(stream : InputStreamPtr);
    END;
  InteractiveTokenStreamPtr = ^ InteractiveTokenStream;

IMPLEMENTATION


FUNCTION TokenName(itsKind : TokenKind) : STRING;

BEGIN
  CASE itsKind OF
    EndOfStream: TokenName := '<fin-de-entrada>';
    LeftParen: TokenName := '(';
    RightParen: TokenName := ')';
    PlusOp: TokenName := '<operador de union>';
    ConcatenationOp : TokenName :=
                        '<operador-de-concatenacion>';
    ClosureOp : TokenName := '<operador-de-cerradura>';
```

```
    VariableSymbol: TokenName := '<simbolo-variable>'
  END
END;

CONST
  TabChar = #9;
  LineFeedChar = #10;


FUNCTION TokenStream.CurrentTokenKind : TokenKind;

BEGIN
  CurrentTokenKind := CurrentToken.kind
END;


FUNCTION TokenStream.CurrentSymbolString : STRING;

BEGIN
  CurrentSymbolString := CurrentToken.SymbolString
END;


PROCEDURE TokenStream.Advance;


PROCEDURE LexError(ErrMsg : STRING);

BEGIN
  IF CurrentToken.kind <> BadToken THEN
    BEGIN
      CurrentToken.kind := BadToken;
      BadTokenMessage := ErrMsg
    END;
END;

PROCEDURE GetVariableSymbol;

BEGIN
  WITH CurrentToken DO
    BEGIN
      kind := VariableSymbol;
      SymbolString := '';
      REPEAT
        SymbolString :=
                SymbolString + charStream^.CurrentChar;
```

```
        charStream^.Advance;
      UNTIL NOT (charStream^.CurrentChar IN
       ['0' .. '9', 'A' .. 'Z', 'a' .. 'z', '%', '@']);
    END
END;


PROCEDURE SetTokenAndAdvance(theKind : TokenKind);

BEGIN
  CurrentToken.kind := theKind;
  charStream^.Advance
END;

BEGIN
  WHILE charStream^.CurrentChar IN
                      [' ', TabChar, EndOfLineChar] DO
    charStream^.Advance;
  CurrentToken.TokenPos := charStream^.CharPos;
  CASE charStream^.CurrentChar OF
    '+': SetTokenAndAdvance(PlusOp);
    '(': SetTokenAndAdvance(LeftParen);
    ')': SetTokenAndAdvance(RightParen);
    '.': SetTokenAndAdvance(ConcatenationOp);
    '*': SetTokenAndAdvance(ClosureOp);
    'A' .. 'Z', 'a' .. 'z', '0' .. '9','%','@':
                                      GetVariableSymbol;
    EndOfFileChar: CurrentToken.kind := EndOfStream
  ELSE LexError('Caracter invalido en la entrada.')
  END
END;


FUNCTION TokenStream.CurrentTokenPosition : INTEGER;

BEGIN
  CurrentTokenPosition := CurrentToken.TokenPos
END;


FUNCTION TokenStream.CurrentTokenLine : INTEGER;

BEGIN
  CurrentTokenLine := charStream^.LineNo
END;
```

```
PROCEDURE TokenStream.Terminate;

BEGIN
  CurrentToken.kind := EndOfStream
END;


CONSTRUCTOR TokenStream.Init(stream : InputStreamPtr);

BEGIN
  charStream := stream;
  Advance
END;


FUNCTION InteractiveTokenStream.CurrentTokenKind :
                                                TokenKind;

BEGIN
  IF CurrentTokenNotSet THEN
    BEGIN
      TokenStream.Advance;
      CurrentTokenNotSet := FALSE
    END;
  CurrentTokenKind := CurrentToken.kind
END;


FUNCTION InteractiveTokenStream.CurrentSymbolString :
                                                    STRING;

BEGIN
  IF CurrentTokenNotSet THEN
    BEGIN
      TokenStream.Advance;
      CurrentTokenNotSet := FALSE
    END;
  CurrentSymbolString := CurrentToken.SymbolString
END;


PROCEDURE InteractiveTokenStream.Advance;

BEGIN
```

```
  CASE CurrentToken.kind OF
    LeftParen: ParenLevel := ParenLevel + 1;
    RightParen: ParenLevel := ParenLevel - 1;
  END;
  WHILE charStream^.CurrentChar IN [' ', TabChar] DO
        charStream^.Advance;
  IF (ParenLevel = 0) AND
       (charStream^.CurrentChar = EndOfLineChar) THEN
    BEGIN
      CurrentToken.kind := EndOfStream;
      CurrentToken.TokenPos := charStream^.CharPos
    END
  ELSE CurrentTokenNotSet := TRUE
END;


CONSTRUCTOR InteractiveTokenStream.Init
                              (stream : InputStreamPtr);

BEGIN
  charStream := stream;
  CurrentTokenNotSet := TRUE;
  ParenLevel := 0
END;

END .
```

B.2.4 Módulo Parser

```
UNIT Parser;

(*  Este modulo proporciona el parser para
    las expresiones regulares,
    implementado de acuerdo a la tecnica
    de parseo recursivo-descendente          *)

INTERFACE
  USES rExpressions, LexicalAnalyzer;


VAR FailureLineNo, FailureCharNo : INTEGER;
    FailureMessage : STRING[72];

(*  ParsesAsExpr determina si el flujo de entrada dado
    puede ser parseado como una expresion regular
    "bien formada". Si ese es el caso, la expresion resultante
```

```
    es guardada en la variable ParsedExpression. De otra forma,
    el punto en el cual el parseo falla se reporta en
    las variables FailureLineNo y FailureCharNo, junto
    con el mensaje de error apropiado en FailureMessage. *)

FUNCTION ParsesAsExpr(tokens : TokenStreamPtr) :
                                                    BOOLEAN;

VAR ParsedExpression : RegExprType;

IMPLEMENTATION

(* La implementacion del parser sigue el patron
    del parseo recursivo descendente.

    Manejo de errores :  Cuando se intenta  parsear
    una expresion, ParseHasFailed es inicializada a FALSE.
    Cuando se encuentra un error de parseo,
    esta variable adquiere el valor de TRUE y ,
    al mismo tiempo, se reporta el error.
    En este punto el proceso de parseo termina
    conceptualmente, regresando la indicacion de error
    de parseo a la rutina que lo inicio.
    Sin embargo, puesto que Turbo Pascal no posee
    mecanismos para
    transferencias de control no local,
    se permite que el parseo
    continue hasta que termine normalmente
    (ya que siempre terminara).
    Generalmente, se pueden hallar algunos errores
    adicionales, pero estos no se reportan
    ya que pueden ser producto del primer error.
    Cuando el control regresa finalmente
    a la rutina ParsesAsExpr, el espacio
    de heap ocupado por los objetos alojados por el
    parser se libera y se regresa una indicacion
    apropiada de error de parseo.                    *)

VAR theTokenStream : TokenStreamPtr;
    ParseHasFailed : BOOLEAN;


PROCEDURE ParsingError(expectedItem : STRING);

BEGIN
  IF NOT ParseHasFailed THEN
```

```
    BEGIN
      FailureLineNo := theTokenStream^.CurrentTokenLine;
      FailureCharNo :=
                  theTokenStream^.CurrentTokenPosition;
      IF theTokenStream^.CurrentTokenKind = BadToken THEN
        FailureMessage := theTokenStream^.BadTokenMessage
      ELSE
       FailureMessage := 'Se espera ' + expectedItem + '.';
      ParseHasFailed := TRUE;
      theTokenStream^.Terminate
    END
END;


PROCEDURE AcceptToken(theKind : TokenKind);

VAR ExpectedToken : STRING;

BEGIN
  IF theTokenStream^.CurrentTokenKind = theKind THEN
       theTokenStream^.Advance
  ELSE ParsingError(TokenName(theKind))
END;


FUNCTION ParseSimple : RegExprType; FORWARD;


FUNCTION ParseVariable : RegExprType;

BEGIN
  IF theTokenStream^.CurrentTokenKind = VariableSymbol THEN
    BEGIN
      ParseVariable :=
       MakeVariable(theTokenStream^.CurrentSymbolString);
      theTokenStream^.Advance
    END
  ELSE
    BEGIN
      ParsingError('<variable>');
      ParseVariable := TheUndefinedValue
    END
END;


FUNCTION ParseFactor : RegExprType;
```

```
VAR e1 : RegExprType;

BEGIN
  e1 := ParseSimple;
  IF theTokenStream^.CurrentTokenKind = ClosureOp THEN
    BEGIN
      thetokenStream^.Advance;
      ParseFactor := MakeClosure(e1)
    END
  ELSE ParseFactor := e1
END;


FUNCTION ParseTerm : RegExprType;

VAR e1, e2 : RegExprType;

BEGIN
  e1 := ParseFactor;
  WHILE theTokenStream^.CurrentTokenKind IN
                                    [ConcatenationOp] DO
  BEGIN
    theTokenStream^.Advance;
    e2 := ParseFactor;
    e1 := MakeConcatenation(e1, e2)
  END;
  ParseTerm := e1
END;


FUNCTION ParseExpression : RegExprType;
VAR e1, e2 : RegExprType;
BEGIN
  e1 := ParseTerm;
  WHILE theTokenStream^.CurrentTokenKind IN [PlusOp] DO
  BEGIN
    theTokenStream^.Advance;
    e2 := ParseTerm;
    e1 := MakeSum(e1, e2)
  END;
  ParseExpression := e1
END;

FUNCTION ParseParenthesis : RegExprType;
```

```
VAR arg : RegExprType;

BEGIN
  AcceptToken(LeftParen);
  arg := ParseExpression;
  AcceptToken(RightParen);
  ParseParenthesis := arg
END;


FUNCTION ParseSimple : RegExprType;

BEGIN
  CASE theTokenStream^.CurrentTokenKind OF
    VariableSymbol : ParseSimple := ParseVariable;
    LeftParen: ParseSimple := ParseParenthesis
   ELSE ParsingError('<simple>')
  END
END;


FUNCTION ParsesAsExpr(tokens : TokenStreamPtr) :
                                                    BOOLEAN;

VAR HeapSpaceMark : Pointer;

BEGIN
  theTokenStream := tokens;
  ParseHasFailed := FALSE;
  Mark(HeapSpaceMark);
  ParsedExpression := ParseExpression;
  IF theTokenStream^.CurrentTokenKind <> EndOfStream THEN
    ParsingError('<fin-de-entrada>');
  IF ParseHasFailed THEN Release(HeapSpaceMark);
  ParsesAsExpr := NOT ParseHasFailed
END;

END .
```

B.2.5 Módulo Simplif

```
UNIT Simplif;

(* En este modulo se ha implementado el
algoritmo de simplificacion y sus funciones asociadas   *)
```

```
INTERFACE

USES rExpressions;

FUNCTION Simplify( expr : RegExprType) : RegExprType;

IMPLEMENTATION

FUNCTION CanonicalSum(expr1, expr2 : RegExprType) :
                                    RegExprType; FORWARD;

FUNCTION SumandP(e1, e2 : RegExprType) : BOOLEAN;
BEGIN
 IF VariableP(e2) THEN
    BEGIN
      IF VariableP(e1) THEN
         SumandP := Equal(e1, e2)
      ELSE IF VariableP(Operand2(e1)) THEN
            BEGIN
               IF Equal(e2, Operand2(e1)) THEN
                 SumandP := TRUE
               ELSE
                 BEGIN
                   IF SumP(e1) THEN
                     SumandP := SumandP(Operand1(e1), e2)
                   ELSE
                     SumandP := FALSE
                 END
             END
      ELSE
         BEGIN
           IF SumP(e1) THEN
             SumandP := SumandP(Operand1(e1), e2)
           ELSE
             SumandP := FALSE
         END
    END
 ELSE IF ClosureP(e2) THEN
    BEGIN
       IF ClosureP(e1) THEN
         SumandP := Equal(e1, e2)
      ELSE IF ClosureP(Operand2(e1)) THEN
            BEGIN
               IF Equal(e2, Operand2(e1)) THEN
                 SumandP := TRUE
```

```
                        ELSE
                          BEGIN
                            IF SumP(e1) THEN
                              SumandP := SumandP(Operand1(e1), e2)
                            ELSE
                              SumandP := FALSE
                          END
                      END
                ELSE
                   BEGIN
                     IF SumP(e1) THEN
                       SumandP := SumandP(Operand1(e1), e2)
                     ELSE
                       SumandP := FALSE
                   END
            END
          ELSE IF ConcatenationP(e2) THEN
            BEGIN
              IF ConcatenationP(e1) THEN
                 SumandP := Equal(e1, e2)
              ELSE IF ConcatenationP(Operand2(e1)) THEN
                    BEGIN
                       IF Equal(e2, Operand2(e1)) THEN
                         SumandP := TRUE
                       ELSE
                         BEGIN
                           IF SumP(e1) THEN
                             SumandP := SumandP(Operand1(e1), e2)
                           ELSE
                             SumandP := FALSE
                         END
                    END
              ELSE
                 BEGIN
                   IF SumP(e1) THEN
                     SumandP := SumandP(Operand1(e1), e2)
                   ELSE
                     SumandP := FALSE
                 END
            END
          ELSE SumandP := FALSE
        END;

        FUNCTION MakeSimplifiedSum(expr1, expr2 : RegExprType) :
                                                    RegExprType;
```

```
BEGIN
  IF VoidP (expr2) THEN MakeSimplifiedSum :=
                            CanonicalSum(expr2, expr1)
  ELSE IF EmptyP(expr2) THEN
      MakeSimplifiedSum := CanonicalSum(expr2, expr1)
  ELSE IF SumP(expr1) THEN
         BEGIN
            IF SumandP(expr1, expr2) THEN
              BEGIN
               expr2 := TheVoidSet;
               MakeSimplifiedSum :=
                            CanonicalSum(expr2, expr1)
              END
            ELSE MakeSimplifiedSum :=
                    CanonicalSum(expr1, expr2)
         END
  ELSE MakeSimplifiedSum := CanonicalSum(expr1, expr2)
END;

FUNCTION MakeSimplifiedConcatenation
    (expr1, expr2: RegExprType) : RegExprType; FORWARD;

FUNCTION MakeSimplifiedClosure
          (expr : RegExprType) : RegExprType; FORWARD;

FUNCTION CanonicalSum(expr1, expr2 : RegExprType) :
                                              RegExprType;
BEGIN
  IF Equal (expr1, expr2) THEN CanonicalSum := expr1
  ELSE IF VoidP(expr1) THEN CanonicalSum := expr2
  ELSE IF VoidP(expr2) THEN CanonicalSum := expr1
  ELSE IF EmptyP(expr1) THEN
        BEGIN
         IF ConcatenationP(expr2) THEN
          BEGIN
           IF (ClosureP(Operand2(expr2))) AND
              (Equal(Operand1(expr2),
                        Operand(Operand2(expr2)))) THEN
                 CanonicalSum :=
                  MakeSimplifiedClosure(Operand1(expr2))
           ELSE CanonicalSum := MakeSum(expr1, expr2)
          END
         ELSE CanonicalSum := MakeSum(expr1, expr2)
        END
  ELSE IF (ConcatenationP(expr1)) AND
                          (ConcatenationP(expr2)) THEN
```

```
            BEGIN
             IF Equal(Operand1(expr1), Operand1(expr2)) THEN
              CanonicalSum :=
               MakeSimplifiedConcatenation(Operand1(expr1),
                 MakeSimplifiedSum(Operand2(expr1),
                     Operand2(expr2)))
              ELSE
               BEGIN
                IF Equal(Operand2(expr1), Operand2(expr2)) THEN
                 CanonicalSum :=
                  MakeSimplifiedConcatenation(MakeSimplifiedSum
                    (Operand1(expr1), Operand1(expr2)),
                                        Operand2(expr1))
                 ELSE CanonicalSum := MakeSum(expr1, expr2)
               END
            END
  ELSE CanonicalSum := MakeSum(expr1, expr2)
END;

FUNCTION MakeSimplifiedClosure (expr : RegExprType) :
                                              RegExprType;
BEGIN
  IF EmptyP (expr) THEN MakeSimplifiedClosure := expr
  ELSE IF VoidP(expr) THEN
              MakeSimplifiedClosure := TheEmptyString
  ELSE IF ClosureP(expr) THEN
        MakeSimplifiedClosure :=
                  MakeSimplifiedClosure(Operand(expr))
  ELSE IF ConcatenationP(expr) THEN
        BEGIN
         IF (ClosureP(Operand1(expr))) AND
              (ClosureP(Operand2(expr))) THEN
               MakeSimplifiedClosure:=
                MakesimplifiedClosure(MakeSimplifiedSum
                          (Operand(Operand1(expr)),
                            Operand(Operand2(expr))))
         ELSE
          MakeSimplifiedClosure := MakeClosure(expr)
        END
  ELSE IF SumP(expr) THEN
        BEGIN
         IF (ClosureP(Operand1(expr))) AND
                      (ClosureP(Operand2(expr))) THEN
          MakeSimplifiedClosure:=
             MakeSimplifiedClosure(MakeSimplifiedSum
                 (Operand(Operand1(expr)),
```

```
                                    Operand(Operand2(expr))))
            ELSE IF EmptyP(Operand1(expr)) THEN
              MakeSimplifiedClosure:=
                    MakeSimplifiedClosure(Operand2(expr))
            ELSE IF EmptyP(Operand2(expr)) THEN
                  MakeSimplifiedClosure:=
                    MakeSimplifiedClosure(Operand1(expr))
            ELSE MakeSimplifiedClosure:=MakeClosure(expr)
           END
  ELSE MakeSimplifiedClosure := MakeClosure(expr)
END;

FUNCTION CanonicalConcatenation
           (expr1, expr2 : RegExprType) : RegExprType;

BEGIN
  IF VoidP(expr1) THEN CanonicalConcatenation := expr1
  ELSE IF EmptyP(expr1) THEN
                        CanonicalConcatenation := expr2
  ELSE IF (ClosureP(expr1)) AND (ClosureP(expr2)) THEN
        BEGIN
         IF Equal(Operand(expr1),Operand(expr2)) THEN
          CanonicalConcatenation :=
                MakeSimplifiedClosure(Operand(expr1))
         ELSE IF ConcatenationP(Operand(expr2)) THEN
               BEGIN
                IF
                Equal(Operand2(operand(expr2)), expr1)
                 THEN
                   CanonicalConcatenation :=
                    MakeSimplifiedClosure
                     (MakeSimplifiedSum(Operand(expr1),
                            Operand1(Operand(expr2))))
                 ELSE CanonicalConcatenation :=
                        MakeConcatenation(expr1, expr2)
               END
         ELSE IF ConcatenationP(Operand(expr1)) THEN
               BEGIN
                IF Equal(Operand1(Operand(expr1)), expr2) THEN
                   CanonicalConcatenation:=
                    MakeSimplifiedClosure
                    (MakeSimplifiedSum(Operand(expr2),
                           Operand2(Operand(expr1))))
                 ELSE CanonicalConcatenation :=
                       MakeConcatenation(expr1, expr2)
               END
```

```
        ELSE CanonicalConcatenation :=
                        MakeConcatenation(expr1, expr2)
      END
  ELSE CanonicalConcatenation :=
                        MakeConcatenation(expr1, expr2)
END;

FUNCTION MakeSimplifiedConcatenation
          (expr1, expr2:RegExprType) : RegExprType;

BEGIN
  IF VoidP(expr2) THEN
     MakeSimplifiedConcatenation :=
                    CanonicalConcatenation(expr2, expr1)
  ELSE IF EmptyP(expr2) THEN
     MakeSimplifiedConcatenation :=
                    CanonicalConcatenation(expr2, expr1)
  ELSE IF (ClosureP(expr1)) AND NOT
                                (ClosureP(expr2)) THEN
       BEGIN
        IF Equal(Operand(expr1), expr2) THEN
         MakeSimplifiedConcatenation :=
               CanonicalConcatenation(expr2, expr1)
        ELSE MakeSimplifiedConcatenation :=
              CanonicalConcatenation(expr1, expr2)
       END
  ELSE MakeSimplifiedConcatenation :=
              Canonicalconcatenation(expr1, expr2)
END;

FUNCTION Simplify( expr : RegExprType ) : RegExprType;

BEGIN
  IF VariableP(expr) THEN Simplify := expr
  ELSE IF ClosureP(expr) THEN
          Simplify :=
           MakeSimplifiedClosure(Simplify(Operand(expr)))
  ELSE IF ConcatenationP(expr) THEN
   Simplify := MakeSimplifiedConcatenation
     (Simplify(Operand1(expr)), Simplify(Operand2(expr)))
  ELSE IF SumP(expr) THEN
    Simplify := MakeSimplifiedSum
     (Simplify(Operand1(expr)), Simplify(Operand2(expr)))
END;
END.
```

B.2.6 Módulo OutputStreams

```
UNIT OutputStream;

(* Este modulo proporciona la abstraccion
   de flujo de salida usada por
   el pretty printer.                    *)

INTERFACE

VAR LineWidth : INTEGER;

PROCEDURE InitOutputStream(VAR f : TEXT);

PROCEDURE Emit(s : STRING);

PROCEDURE SuppressBlank;

PROCEDURE NewBlock;

PROCEDURE EndBlock;

PROCEDURE Indent(relativeAmount : INTEGER);

PROCEDURE LineBreak(blankLines : INTEGER);

FUNCTION AvailableSpace : INTEGER;

FUNCTION SpaceGainedByLineBreak : INTEGER;

IMPLEMENTATION

CONST MaxMarginStackDepth = 40;

VAR CurrentLeftMargin, CursorColumn,
    BlanksNeeded, BlockDepth : INTEGER;
    OutputFilePtr : ^ TEXT;
    LeftMarginStack :
            ARRAY [1..MaxMarginStackDepth] OF INTEGER;


PROCEDURE InitOutputStream(VAR f : TEXT);

BEGIN
  OutputFilePtr := @f;
  BlanksNeeded := 0;
```

```
  CursorColumn := 0;
  BlockDepth := 0;
  CurrentLeftMargin := 0
END;


PROCEDURE Emit(s : STRING);

VAR Indent, i : INTEGER;

BEGIN
  IF Length(s) > AvailableSpace THEN
    BEGIN
      writeln(OutputFilePtr^);
      IF CurrentLeftMargin + Length(s) > LineWidth THEN
        Indent := LineWidth - Length(s)
      ELSE Indent := CurrentLeftMargin;
      FOR i := 1 TO Indent DO
          write(OutputFilePtr^, ' ');
      CursorColumn := Indent;
      BlanksNeeded := 0
    END;
  FOR i := 1 TO BlanksNeeded DO
     write(OutputFilePtr^, ' ');
  write(OutputFilePtr^, s);
  CursorColumn :=
                CursorColumn + Length(s) + BlanksNeeded;
  BlanksNeeded := 1
END;


PROCEDURE SuppressBlank;

BEGIN
  BlanksNeeded := 0
END;


PROCEDURE NewBlock;

BEGIN
  BlockDepth := BlockDepth + 1;
  IF BlockDepth <= MaxMarginStackDepth THEN
    BEGIN
      LeftMarginStack[BlockDepth] := CurrentLeftMargin;
      CurrentLeftMargin := CursorColumn + BlanksNeeded
```

```
    END
END;


PROCEDURE EndBlock;

BEGIN
  IF BlockDepth <= MaxMarginStackDepth THEN
    CurrentLeftMargin := LeftMarginStack[BlockDepth];
  BlockDepth := BlockDepth - 1
END;


PROCEDURE Indent(relativeAmount : INTEGER);

BEGIN
  CurrentLeftMargin := CurrentLeftMargin + relativeAmount
END;


FUNCTION AvailableSpace : INTEGER;

BEGIN
  AvailableSpace := LineWidth -
                          CursorColumn - BlanksNeeded
END;


FUNCTION SpaceGainedByLineBreak : INTEGER;

BEGIN
  SpaceGainedByLineBreak := CursorColumn +
                       BlanksNeeded - CurrentLeftMargin
END;


PROCEDURE LineBreak(blankLines : INTEGER);

VAR i : INTEGER;

BEGIN
  FOR i := 0 TO blankLines DO
     writeln(OutputFilePtr^);
  FOR i := 1 TO CurrentLeftMargin DO
     write(OutputFilePtr^, ' ');
  CursorColumn := CurrentLeftMargin;
```

```
  BlanksNeeded := 0
END;

BEGIN
  LineWidth := 75
END .
```

B.2.7 Módulo PrettyPrinter

```
UNIT PrettyPrinter;

(*  Este modulo proporciona el pretty printer
    para las expresiones regulares.            *)

INTERFACE
  USES rExpressions;

CONST SpaceGainNeededForLineBreak = 5;

PROCEDURE PrettyPrintRExpression
         (VAR outFile : TEXT; theExpr : RegExprType);


IMPLEMENTATION
USES OutputStream;



FUNCTION SpaceAfterVariable
       (x : RegExprType; initialSpace : INTEGER) :INTEGER;

BEGIN
  SpaceAfterVariable :=
                initialSpace - Length(SymbolString(x)^)
END;


FUNCTION SpaceAfterRexpr
    (x : RegExprType; initialSpace : INTEGER) :INTEGER;

BEGIN
  IF VariableP(x) THEN
    SpaceAfterRexpr := SpaceAfterVariable(x, initialSpace)
  ELSE IF ClosureP(x) THEN
    SpaceAfterRexpr :=
```

```
                SpaceAfterRexpr(Operand(x), initialSpace)
  ELSE IF (ConcatenationP(x)) OR (SumP(x)) THEN
        SpaceAfterRexpr :=
           SpaceAfterRexpr(Operand1(x), initialSpace) +
           SpaceAfterRexpr(Operand2(x), initialSpace) -4
END;


PROCEDURE PrettyVariable
            (x : RegExprType; MarginReserve : INTEGER);

VAR symString : STRING;

BEGIN
  symString := SymbolString(x)^;
  IF Length(symString) + MarginReserve > AvailableSpace
                                      THEN LineBreak(0);
  Emit(symString)
END;


PROCEDURE PrettyRexpr
    (x : RegExprType; MarginReserve : INTEGER); FORWARD;

PROCEDURE UnparseParen(x :RegExprType);
BEGIN
   Emit('(');
   SuppressBlank;
   PrettyRexpr(x, 0);
   SuppressBlank;
   Emit(')');
   SuppressBlank
END;
PROCEDURE UnparseTerm(x:RegExprType);FORWARD;
PROCEDURE UnparseFactor(x:RegExprType);FORWARD;
PROCEDURE UnparseSimple(x:RegExprType);FORWARD;
PROCEDURE PrettySum
           (x : RegExprType; MarginReserve : INTEGER);

BEGIN
  IF (SpaceGainedByLineBreak >= SpaceGainNeededForLineBreak)
   AND
   (SpaceAfterRexpr(x, AvailableSpace - MarginReserve) <= 0)
  THEN LineBreak(0);
  SuppressBlank;
```

```
  PrettyRexpr(Operand1(x), MarginReserve);
  SuppressBlank;
  Emit('+');
  SuppressBlank;
  UnparseTerm(Operand2(x));
  SuppressBlank;
END;
PROCEDURE PrettyConcatenation
            (x : RegExprType; MarginReserve : INTEGER);

BEGIN
  IF (SpaceGainedByLineBreak >= SpaceGainNeededForLineBreak)
   AND
    (SpaceAfterRexpr(x, AvailableSpace - MarginReserve) <= 0)
    THEN LineBreak(0);
  SuppressBlank;
  UnparseTerm(Operand1(x));
  SuppressBlank;
  Emit('.');
  SuppressBlank;
  UnparseFactor(Operand2(x));
  SuppressBlank;
END;

PROCEDURE PrettyClosure
            (x : RegExprType; MarginReserve : INTEGER);

BEGIN
  IF (SpaceGainedByLineBreak >= SpaceGainNeededForLineBreak)
  AND
   (SpaceAfterRexpr(x, AvailableSpace - MarginReserve) <= 0)
   THEN LineBreak(0);
  SuppressBlank;
  UnparseSimple(Operand(x));
  SuppressBlank;
  Emit('*');
  SuppressBlank;
END;

PROCEDURE UnparseSimple(x:RegExprType);

BEGIN
  IF VariableP(x) THEN
    PrettyVariable(x,0)
  ELSE
    UnparseParen(x)
```

```
END;

PROCEDURE UnparseFactor(x:RegExprType);

BEGIN
  IF ClosureP(x) THEN
    PrettyClosure(x, 0)
  ELSE
    UnparseSimple(x)
END;

PROCEDURE UnparseTerm(x:RegExprType);

BEGIN
  IF ConcatenationP(x) THEN
    PrettyConcatenation(x,0)
  ELSE
    UnparseFactor(x)
END;



PROCEDURE PrettyRexpr
           (x : RegExprType; MarginReserve : INTEGER);

BEGIN
  IF SumP(x) THEN PrettySum(x, MarginReserve)
  ELSE UnparseTerm(x)
END;


PROCEDURE PrettyPrintRexpression
           (VAR outFile : TEXT; theExpr : RegExprType);

BEGIN
  InitOutputStream(outFile);
  PrettyRexpr(theExpr, 0);
  writeln(outFile)
END;

END .
```

B.2.8 Módulo Auxiliar

```
UNIT Auxiliar;

(* En esta unidad se definen algunos procedimientos
   para leer de la entrada (RdLine) o escribir en la
   salida (WrChar, WrInt, WrText, WrLine y WrTextLine).
   La razon de ser de estos procedimientos es facilitar
   la implementacion de otras unidades, evitando
   la confusion. Tambien pueden tener utilidad en la
   lectura e interpretacion de los archivos fuentes  *)

INTERFACE
PROCEDURE RdLine(VAR textLine : STRING);


PROCEDURE WrChar(ch : CHAR);


PROCEDURE WrInt(x : INTEGER);


PROCEDURE WrText(theText : STRING);


PROCEDURE WrLine;


PROCEDURE WrTextLine(theText : STRING);


IMPLEMENTATION


PROCEDURE RdLine(VAR textLine : STRING);

BEGIN
  READLN(input, textLine);
END;


PROCEDURE WrChar(ch : CHAR);

BEGIN
  write(output, ch)
END;
```

```
PROCEDURE WrInt(x : INTEGER);

BEGIN
  write(output, x)
END;


PROCEDURE WrText(theText : STRING);

BEGIN
  write(output, theText)
END;


PROCEDURE WrLine;

BEGIN
  writeln(output)
END;


PROCEDURE WrTextLine(theText : STRING);

BEGIN
  writeln(output, theText)
END;

END .
```

B.2.9 Programa Principal

```
PROGRAM RegExpr;

(*  Este es el programa principal del sistema
    simplificador de expresiones regulares.   *)

{$M 65520,0,655360}

USES rExpressions, Auxiliar, InputStreams,
     LexicalAnalyzer, Parser,Simplif, OutputStream,
     PrettyPrinter;

CONST TabChar = #9;
```

```
VAR ExitSave : Pointer;

{$F+}


PROCEDURE RuntimeErrors;

BEGIN
  ExitProc := ExitSave;
  IF ErrorAddr <> NIL THEN
    BEGIN
     IF ExitCode = 202 THEN
       WrTextLine
       ('Error de overflow de Stack -
                    posible recursion infinita!!')
     ELSE IF ExitCode = 203 THEN
      WrTextLine
      ('Error de overflow de Heap -
            el programa usa demasiada memoria.')
      ELSE
        BEGIN
          WrText('Error de Run time #');
          WrInt(ExitCode);
          WrLine;
          IF ExitCode >= 200 THEN
           BEGIN
            WrTextLine
            ('Error desconocido en el
               Simplificador de Expresiones Regulares!');
           END
        END;
      ErrorAddr := NIL
    END;

END;

{$F-}


PROCEDURE WriteShortHelp;

BEGIN
  WrTextLine
    ('Comandos:  /ayuda, /info, /salir.');
END;
```

```
PROCEDURE WriteLongHelp;

BEGIN
  WrTextLine
    ('Introduzca expresiones regulares' +
     ' o uno de los siguientes comandos:');
  WrLine;
  WrTextLine('/ayuda          imprime esta informacion de ayuda.');
  WrTextLine('/info           imprime informacion del realizador.');
  WrTextLine('/salir          concluye la sesion.');
  WrLine;
  WrTextLine
   ('Las expresiones pueden abarcar varias lineas.');
  WrTextLine
   ('El operador de concatenacion es . (punto).');
  WrTextLine
   ('La cadena vacia y el conjunto vacio se representan por' +
                                  ' % y @, respectivamente.');
  WrTextLine
    ('La entrada puede abortarse
     forzando un error de sintaxis(p.e., con ^X [RETURN]).')
END;

PROCEDURE WriteInfo;

BEGIN
  WrTextLine
    ('Este programa fue realizado por
                Alejandro Augusto Rafael Trejo Ortiz');
  WrLine;
END;

TYPE
  InputLineClass = (BlankLine, CommandLine, ExpressionLine);


FUNCTION InputLineType(s : STRING) : InputLineClass;

VAR i : INTEGER;

BEGIN
  i := 1;
  WHILE (i <= Length(s)) AND
      ((s[i] = ' ') OR (s[i] = TabChar)) DO
          i := i + 1;
```

```
  IF i > Length(s) THEN InputLineType := BlankLine
  ELSE IF s[i] = '/' THEN InputLineType := CommandLine
  ELSE InputLineType := ExpressionLine
END;


PROCEDURE ParseCommandLine
                (inputLine : STRING; VAR Cmd : STRING);

VAR i : INTEGER;

BEGIN
  i := 1;
  WHILE inputLine[i] <> '/' DO
    i := i + 1;
  Cmd := '';
  i := i + 1;
  WHILE (i <= Length(inputLine)) AND
                  (inputLine[i] IN [' ', TabChar]) DO
     i := i + 1;
  WHILE (i <= Length(inputLine)) AND
        (inputLine[i] IN ['A' .. 'Z', 'a' .. 'z']) DO
    BEGIN
      Cmd := Cmd + UpCase(inputLine[i]);
      i := i + 1
    END;
END;


PROCEDURE SessionLoop;
(* El procedimiento SessionLoop ejecuta repetidamente
   sesiones interactivas hasta que
   se recibe un comando /salir.      *)

(* El procedimiento DoSession ejecuta una sola sesion.
   La sesion concluye con un
   comando /salir .          *)


PROCEDURE DoSession (VAR QuitSignalled : BOOLEAN);

VAR
    inputLine, CommandWord : STRING;
    thsInputStream : InputStreamPtr;
    theTokenStream : InteractiveTokenStreamPtr;
    result : RegExprType;
```

```
    i : INTEGER;
    EvalHeapMark : Pointer;

BEGIN
  NEW(theInputStream);
  NEW(theTokenStream);
  QuitSignalled := FALSE;


  REPEAT
    RdLine(inputLine);
    CASE InputLineType(inputLine) OF
      CommandLine:
        BEGIN
          ParseCommandLine(inputLine, CommandWord);
          IF CommandWord = 'AYUDA' THEN WriteLongHelp
          ELSE IF CommandWord = 'INFO' THEN WriteInfo
          ELSE IF CommandWord = 'SALIR'
                              THEN QuitSignalled := TRUE
          ELSE
            BEGIN
              WrTextLine('Comando desconocido.');
              WriteShortHelp
            END
        END;
      ExpressionLine:
        BEGIN
          theInputStream^.Init(inputLine);
          theTokenStream^.Init(theInputStream);
          Mark(EvalHeapMark);
          IF ParsesAsExpr(theTokenStream) THEN
            BEGIN
              result := Simplify(ParsedExpression);
              IF result <> TheUndefinedValue THEN
                BEGIN
                 PrettyPrintRexpression(output, result);
                END
            END
          ELSE
            BEGIN
              FOR i := 1 TO FailureCharNo - 1 DO
                 WrChar(' ');
              WrTextLine('^');
              WrTextLine('Error:  ' + FailureMessage)
            END;
          Release(EvalHeapMark)
```

```
        END;
      BlankLine:
        (* No hacer nada,solo obtener la linea siguiente. *)
    END
  UNTIL QuitSignalled
END;

VAR QuitSignalled : BOOLEAN;

BEGIN
  REPEAT
    DoSession(QuitSignalled);
  UNTIL QuitSignalled
END;

BEGIN
  IF ParamCount > 0 THEN writeln('Uso: regexpr.')
  ELSE
    BEGIN
      ExitSave := ExitProc;
      ExitProc := @RuntimeErrors;
      WriteShortHelp;
      SessionLoop
    END
END .
```

Bibliografía

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addisson-Wesley, Reading, Massachusetts, 1986.

[2] J. Brzozowski. *Derivatives of Regular Expressions.* *JACM* **11**(1964).

[3] B. Buchberger, G.E. Collins, R. Loos (Editors). *Computer Algebra, Symbolic and Algebraic Computation.* Springer-Verlag, Wien, Österreich, second edition, 1983.

[4] R. Cameron, A. Dixon. *Symbolic Computing with LISP.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[5] K.O. Geddes, S.R. Czapor, G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publisher, Boston, Massachusetts, 1992.

[6] A. I. Holub. *Compiler design in C.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[7] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[8] H. B. Hunt III, D.J. Rosenkrantz, T. G. Szymanski. *On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages. JCSS* **12** (1976).

[9] D.E. Knuth. *The Art of Computer Programming.* *Vol. 1 Fundamental Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1973.

[10] K. Korsvold. *An On-Line Program for Non-Numerical Algebra.* *CACM* **9** (1966).

[11] J. Sammet. *Survey of Formula Manipulation.* *CACM* **9** (1966).