UNIVERSIDAD NACIONAL DE TRUJILLO
ESCUELA DE INFORMATICA

COMPILADORES

Conociendo las Expresiones Regulares

ALVAREZ ALVAREZ, GUSTAVO ALEXANDER

1. INTRODUCCION

Las expresiones regulares representan patrones de cadenas de caracteres. Una expresión regular r se encuentra completamente definida mediante el conjunto de cadenas con las que concuerda. Este conjunto se denomina lenguaje generado por la expresión regular y se escribe como $L(r)$. Aquí la palabra lenguaje se utiliza sólo para definir "conjunto de cadenas" y no tiene (por lo menos en esta etapa) una relación específica con un lenguaje de programación. Este lenguaje depende, en primer lugar, del conjunto de caracteres que se encuentra disponible. En general, estaremos hablando del conjunto de caracteres ASCII o de algún subconjunto del mismo. En ocasiones el conjunto será más general que el conjunto de caracteres ASCII, en cuyo caso los elementos del conjunto se describirán como símbolos. Este conjunto de símbolos legales se conoce como alfabeto y por lo general se representa mediante el símbolo griego $\sum$ (sigma).

Una expresión regular r también contendrá caracteres del alfabeto, pero esos caracteres tendrán un significado diferente: en una expresión regular todos los símbolos indican patrones. En este documento distinguiremos el uso de un carácter como patrón escribiendo todo los patrones en negritas. De este modo, a es el carácter a usado como patrón.

Por último, una expresión regular r puede contener caracteres que tengan significados especiales. Este tipo de caracteres se llaman metacaracteres o metasímbolos, y por lo general no pueden ser caracteres legales en el alfabeto, porque no podríamos distinguir su uso como metacaracteres de su uso como miembros del alfabeto. Sin embargo, a menudo no es posible requerir tal exclusión, por lo que se debe utilizar una convención para diferenciar los dos usos posibles de un metacaracter. En muchas situaciones esto se realiza mediante el uso de un carácter de escape que "desactiva" el significado especial de un metacarácter. Unos caracteres de escape comunes son la diagonal inversa y las comillas. Advierta que los caracteres de escape, si también son caracteres legales en el alfabeto, son por sí mismos metacaracteres.

2. DEFINICIÓN

Ahora estamos en posición de describir el significado de las expresiones regulares al establecer cuales lenguajes genera cada patrón. Haremos esto en varias etapas. Comenzaremos por describir el conjunto de expresiones regulares básicas, las cuales se componen de símbolos individuales. Continuaremos con la descripción de las operaciones que generan nuevas expresiones regulares a partir de las ya existentes. Esto es similar a la manera en que se construyen las expresiones aritméticas: las expresiones aritméticas básicas son Los números, tales como 43 y 2.5. Entonces las operaciones aritméticas, como la suma y la multiplicación, se pueden utilizar para formar nuevas expresiones a partir de las existentes, como en el caso de $43*2.5$ y $43*25.+1.4$

El grupo de expresiones regulares que describiremos aquí es mínimo, ya que sólo contiene los metasímbolos y las operaciones esenciales. Después consideraremos extensiones a este conjunto mínimo.

Expresiones Regulares Básicas. Éstas son precisamente los caracteres simples del alfabeto, los cuales se corresponden a sí mismos. Dado cualquier carácter a del alfabeto $\sum$, indicamos que la expresión regular a corresponde al carácter **a** escribiendo $L(a) = \{\text{a}\}$. Existen otros dos símbolos que necesitaremos en situaciones especiales. Necesitamos poder indicar una concordancia con la cadena vacía, es decir, la cadena que no contiene ningún carácter. Utilizaremos el símbolo ε (épsilon) para denotar la cadena vacía, y definiremos el metasímbolo ε (ε en negritas) estableciendo que $L(\varepsilon) = \{\varepsilon\}$. También necesitaremos ocasionalmente ser capaces de describir un símbolo que corresponda a la ausencia de cadenas, es decir, cuyo lenguaje sea el conjunto vacío, el cual escribiremos como { }.

Emplearemos para esto el símbolo ϕ y escribiremos $L(\phi) = \{\}$. Observe la diferencia entre { } y $\{\varepsilon\}$: el conjunto { } no contiene ninguna cadena, mientras que el conjunto $\{\varepsilon\}$ contiene la cadena simple que no se compone de ningún carácter.

Operaciones de expresiones regulares. Existen tres operaciones básicas en las expresiones regulares:

1) Selección entre alternativas, la cual se indica mediante el metacarácter | (barra vertical)
2) Concatenación, que se indica mediante yuxtaposición (sin un metacarácter)
3) Repetición o "cerradura", la cual se indica mediante el metacarácter *. Analizaremos cada una por turno proporcionando la construcción del conjunto correspondiente para los lenguajes de cadenas concordantes

Selección entre alternativas. Si r y s son expresiones regulares, entonces r|s es una expresión regular que define cualquier cadena que concuerda con r o con .S. En términos de lenguajes el lenguaje de $r \mid s$ es la unión de los lenguajes de r y s, o $L(r \mid s) = L(r) \cup L(s)$.

Corno un ejemplo simple, considere la expresión regular $a \mid b$: ésta corresponde tanto al carácter a como al carácter b, es decir, $L(a \mid b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$

Corno segundo ejemplo, la expresión regular $a \mid \varepsilon$ corresponde tanto al carácter simple a como a la cadena vacía (que no está compuesta por ningún carácter). En otras palabras, $L(a \mid \varepsilon) = \{a, \varepsilon\}$

La selección se puede extender a más de una alternativa, de manera que, por ejemplo, $L(a \mid b \mid c \mid d) = \{a \mid b \mid c \mid d\}$ En ocasiones también escribiremos largas secuencias de selecciones con puntos, como en $a \mid b \mid c \mid ... \mid z$, que corresponde a cualquiera de las letras minúsculas de la a a la z.

Concatenación. La concatenación de dos expresiones regulares r y s se escribe como rs, y corresponde a cualquier cadena que sea la concatenación de dos cadenas, con la primera de ellas correspondiendo a r y la segunda correspondiendo a s. Por ejemplo, la expresión regular ab corresponde sólo a la cadena ab, mientras que la expresión regular (a|b) c corresponde a las cadenas ac y bc. (El uso de los paréntesis como metacaracteres en esta expresión regular se explicará en breve.)

Podemos describir el efecto de la concatenación en términos de lenguajes generados al definir la concatenación de dos conjuntos de cadenas. Dados dos conjuntos de cadenas S, y S2, el conjunto concatenado de cadenas S_1S_2 es el conjunto de cadenas de S_1 complementado con todas las cadenas de S_2. Por ejemplo, si $S_1 = \{aa, b\}$ y $S_2 = \{a, bb\}$, entonces $S_1S_2 = \{aaa, aab, ba, bbb\}$. Ahora la operación de concatenación para expresiones regulares se puede definir como sigue: $L(rs) = L(r)L(s)$ De esta manera (utilizando nuestro ejemplo anterior), $L((a \mid b)c) = L(a \mid b)L(c) = \{ab\}\{c\} = \{ac, bc\}$

La concatenación también se puede extender a más de dos expresiones regulares: $L(r_1r_2 \ldots r_n) = L(r_1)L(r_2) \ldots L(r_n) =$ el conjunto de cadenas formado al concatenar todas Las cadenas de cada una de las $L(r_1), L(r_2) \ldots L(r_n)$

Repetición. La operación de repetición de una expresión reguIar, denominaúa también en ocasiones cerradura (de Kleene), se escribe r*, donde res una expresión regular. La expresión regular r* corresponde a cualquier concatenación finita de cadenas, cada una de las cuales corresponde a r. Por ejemplo, a* corresponde a las cadenas E, a, aa, aaa, ...

(Concuerda con E porque E es la concatenación de ninguna cadena concordante con a.)

Podemos definir la operación de repetición en términos de lenguajes generados definiendo, a

su vez, una operación similar * para conjuntos de cadenas. Dado un conjunto S de cadenas, sea

$$S^* = \{\varepsilon\} \cup S \cup SS \cup SSS \cup \ldots$$

Ésta es una unión de conjuntos infinita, pero cada uno de sus elementos es una concatenación finita de cadenas de S. En ocasiones el conjunto S* se escribe como sigue:

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

donde $S^n = S...S$ es la concatenación de S por n veces. ($S^0 = \{\varepsilon\}$.)
Ahora podemos definir la operación de repetición para expresiones regulares como sigue:

$$L(r^*) = L(r)^*$$

Considere como ejemplo la expresión regulaf $(a \mid bb)^*$. (De nueva cuenta, la razón de tener paréntesis como metacaracteres se explicará más adelante.) Esta expresión regular corresponde a cualquiera de las cadenas siguientes: $\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb$ y así sucesivamente. En términos de lenguajes, $L((a \mid bb)^*) = L(a \mid bb)^* = \{a, bb\}^*$ $= \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, \ldots\}$

Precedencia de Operaciones y el uso de los paréntesis. La descripción precedente no toma en cuenta la cuestión de la precedencia de las operaciones de elección, concatenación y repetición.
Por ejemplo, dada la expresión regular $a \mid b^*$, ¿deberíamos interpretar esto como $(a \mid b)^*$ o como $a \mid (b^*)$? (Existe una diferencia importante, puesto que $L((a \mid b)^*) = \{\varepsilon, a, b, bb, bbb, \ldots\}$, mientras que $L(a \mid (b^*)) = \{\varepsilon, a, b, bb, bbb, \ldots\}$) La convención estándar es que la repetición debería tener mayor precedencia, por lo tanto, la segunda interpretación es la correcta. En realidad, entre las tres operaciones, se Le da al * la precedencia mas alta, a la concatenación se le da la precedencia que sigue y a la | se le otorga la precedencia más baja. De este modo, por ejemplo, $a \mid bc^*$ se interpreta como $a \mid (b(c^*))$, mientras que $ab \mid c^*d$ se interpreta como $(ab) \mid ((c^*)d)$.
Cuando deseemos indicar una precedencia diferente, debemos usar paréntesis para hacerlo. Ésta es la razón por la que tuvimos que escribir $(a \mid b)c$ para indicar que la operación de elección debería tener mayor precedencia que la concatenación, ya que de otro modo $a \mid bc$ se interpretaría como si correspondiera tanto a a como a bc. De manera similar, $(a \mid bb)^*$ se interpretaría sin los paréntesis como $a \mid bb^*$, lo que corresponde a $a, b, bb, bbb, \ldots$ Los paréntesis aquí se usan igual que en aritmética, donde $(3+4)*5 = 35$, pero $3+4*5 = 23$, ya que se supone que * tiene precedencia más alta que +.

Nombres para expresiones regulares. A menudo es útil como una forma de simplificar la notación proporcionar un nombre para una expresión regular larga, de modo que no tengamos que escribir la expresión misma cada vez que deseemos utilizarla. Por ejemplo, si deseáramos desarrollar una expresión regular para una secuencia de uno o más dígitos numéricos, entonces escribiríamos $(1 \mid 2 \mid \ldots \mid 9)(1 \mid 2 \mid \ldots \mid 9)$

o podríamos escribir $digito \;\; digito^*$

donde $digito = 1 \mid 2 \mid 3 \mid \ldots \mid 9$ es una expresión regular del nombre $digito$.

El uso de una definición regular es muy conveniente, pero introduce la complicación agregada de que el nombre mismo se convierta en un metasímbolo y se deba encontrar un significado para distinguirlo de la concatenación de sus caracteres. En nuestro caso hicimos esa distinción al utilizar letra cursiva para el nombre. Advierta que no se debe emplear el nombre del término en su propia definición (es decir, de manera recursiva): debemos poder eliminar nombres reemplazándolos sucesivamente con las expresiones regulares para las que se establecieron.

Antes de considerar una serie de ejemplos para elaborar nuestra definición de expresiones regulares, reuniremos todas las piezas de la definición de una expresión regular.

Una expresión regular es una de las siguientes:

1. Una expresión regular básica constituida por un solo carácter a, donde a proviene de un alfabeto $\sum$ de caracteres legales; el metacarácter ε; o el metacarácter ϕ. En el primer caso, $L(a) = \{a\}$; en el segundo, $L(\varepsilon) = \{\varepsilon\}$; en el tercero, $L(\phi) = \{\}$.
2. Una expresión de la forma r $r \mid s$, donde r y s son expresiones regulares. En este caso, $L(r \mid s) = L(r) \cup L(s)$.
3. Una expresión de la forma rs, donde r y s son expresiones regulares. En este caso, $L(rs) = L(r)L(s)$.
4. Una expresión de la forma $r*$, donde r es una expresión regular. En este caso, $L(r)* = L(r*)$.
5. Una expresión de la forma (r), donde r es una expresión regular. En este caso, $L((r)) = L(r)$. De este modo, los paréntesis no cambian el lenguaje, sólo se utilizan para ajustar la precedencia de las operaciones.

Advertimos que, en esta definición, la precedencia de las operaciones en (2), (3) y (4) está en el orden inverso de su enumeración; es decir, | tiene precedencia más baja que la concatenación, y ésta tiene una precedencia más baja que el asterisco *. También advertimos que esta definición proporciona un significado de metacarácter a los seis símbolos φ, ε, |, *, (,).

3. EXTENSIONES PARA LAS EXPRESIONES REGULARES

Formulamos una definición para las expresiones regulares que emplean un conjunto mínimo de operaciones comunes a todas las aplicaciones, y podríamos limitarnos a utilizar sólo las tres operaciones básicas (junto con paréntesis) en todos nuestros ejemplos. Sin embargo, ya vimos en los ejemplos anteriores que escribir expresiones regulares utilizando solo estos operadores en ocasiones es poco manejable, ya que se crean expresiones regulares que son más complicadas de lo que serían si se dispusiera de un conjunto de operaciones más expresivo. Por ejemplo, sería útil tener una notación para una coincidencia de cualquier carácter (ahora tenemos que enumerar todos los caracteres en el alfabeto como una larga alternativa). Además, ayudaría tener una expresión regular para una gama de caracteres y una expresión regular para todos los caracteres excepto uno.

En los párrafos siguientes describiremos algunas extensiones a las expresiones regulares estándar ya analizadas, con los correspondientes metasíinbolos nuevos, que cubren éstas y situaciones comunes semejantes. En la mayoría de estos casos no existe una terminología común, de modo que utilizaremos una notación similar a la empleada por el generador de

analizadores léxicos Lex, el cual se describe más adelante en este capítulo. En realidad, muchas de las situaciones que describiremos aparecerán de nuevo en nuestra descripción de Lex. No obstante, no todas las aplicaciones que utilizan expresiones regulares incluirán estas operaciones, e incluso aunque lo hicieran, se puede emplear una notación diferente.

Ahora pasaremos a nuestra lista de nuevas operaciones

Uno o más repeticiones

Dada una expresión regular r, la repetición de r se describe utilizando la operación de cerradura estándar, que se escribe $r*$. Esto permite que r se repita 0 o más veces. Una situación típica que surge es la necesidad de una o más repeticiones en lugar de ninguna, lo que garantiza que aparece por lo tenemos una cadena correspondiente a r, y no permite la cadena vacía ε. Un ejemplo es el de un número natural, donde queremos una secuencia de dígitos, pero deseamos que por lo menos aparezca uno. Por ejemplo: si deseamos definir números binarios, podríamos escribir $(0|1)*$, pero esto también coincidirá con la cadena vacía, la cual no es un número. Por supuesto, podríamos escribir:

$$(0|1)(0|1)*$$

Pero esta situación se presenta con tanta frecuencia que se desarrolló para ella una notación relativamente estándar en la que se utiliza + en lugar de *: $r+$, que indica una o más repeticiones de r. De este modo, nuestra expresión regular anterior para números binarios puede escribirse ahora como:

$$(0|1)+$$

Cualquier Carácter

Una situación común es la necesidad de generar cualquier carácter en el alfabeto. Sin una operación especial esto requiere que todo carácter en el alfabeto sea enumerado en una alternativa. Un metacarácter típico que se utiliza para expresar una concordancia de cualquier carácter es el punto ".", el cual no requiere que el alfabeto se escriba realmente en forma extendida. Con este metacarácter podemos escribir una expresión regular para todas las cadenas que contengan al menos una h como se muestra a continuación:

$$.b.*$$

Un Intervalo de Caracteres

A menudo necesitamos escribir un intervalo de caracteres, como el de todas las letras minúsculas o el de todos los dígitos. Hasta ahora hemos hecho esto utilizando la notación $a|b|\ldots|z$ para las letras minúsculas o $0|1|\ldots|9$ para los dígitos. Una alternativa es tener una notación especial para esta situación, y una que es común es la de emplear corchetes y un guión, como en $[a-z]$ para las letras minúsculas y $[0-9]$ para los dígitos. Esto también se puede emplear para alternativas individuales, de modo que $a|b|c$ puede escribirse como $[a|b|c]$. También se pueden incluir los intervalos múltiples, de manera que $[a-zA-Z]$ representa todas las letras minúsculas y mayúsculas.

Esta notación general se conoce como clases de caracteres. Advierta que esta notación puede depender del orden subyacente del conjunto de caracteres. Por ejemplo, al escribir [A-Z] se supone que los caracteres B, C, y los demás vienen entre los caracteres A y Z (una suposición razonable) y que sólo 109 caracteres en mayúsculas están entre A y Z (algo que es verdadero para el conjunto de caracteres ASCII). Sin embargo, al escribir [A-z] no se definirán los mismos caracteres que para [A-Za-z], incluso en el caso del conjunto de caracteres ASCII.

Cualquier Carácter que no este en un Conjunto dado

Como hemos visto, a menudo es de utilidad poder excluir un carácter simple del conjunto de, caracteres por generar. Esto se puede conseguir al diseñar un metacarácter para indicar la operación de negación ("not") o complementaria sobre un conjunto de alternativas, Por ejemplo, un carácter estándar que representa la negación en lógica es la "tilde" ~, y podríamos escribir una expresión regular para un carácter en el alfabeto que no sea a como $\sim a$ y un carácter que no sea a, ni b, ni c, como

$$\sim (a \mid b \mid c)$$

Una alternativa para esta notación se emplea en Lex, donde el carácter "carat" ^ se utiliza en conjunto con las clases de caracteres que acabamos de describir para la formación de complementos. Por ejemplo: cualquier carácter que no sea a se escribe corno $[\hat{}a]$, mientras que cualquier carácter que no sea a, ni b ni c se escribe como:

$$[\hat{}abc]$$

Subexpresiones Opcionales

Por último, un suceso que se presenta comúnmente es el de cadenas que contienen partes opcionales que pueden o no aparecer en cualquier cadena en particular. Por ejemplo, un número puede o no tener un signo inicial, tal como + o -. Podemos emplear alternativas para expresar esto como en las definiciones regulares:

$$natural = [0-9]+$$

$$naturalconSigno = natural + natural + natural$$

Esto se puede convertir rápidamente en algo voluminoso, e introduciremos e1 metacarácter de signo de interrogación $r?$ para indicar que las cadenas que coincidan con r son opcionales (o que están presentes 0 o 1 copias de r). De este modo, el ejemplo del signo inicial se convierte en

$$natural = [0-9]+$$

$$naturalconSigno = (+ \mid -)?\, natural$$

4. Expresiones Regulares para Tokens de Lenguajes de Programación

Los tokens de lenguajes de programación tienden a caer dentro de varias categorías limitadas que son basta~ite estandarizadas a través de muchos lenguajes de programaciún diferentes. Una categoría es la de las palabras reservadas, en ocasiones también conocidas como palabras clave, que son cadenas fijas de caracteres alfabéticos que tienen un significado especial en el lenguaje. Los ejemplos incluyen if, while y do en lenguajes como Pascal, C y Ada. Otra categoría se compone de los símbolos especiales, que incluyen operadores aritméticos, de asignación y de igualdad. Éstos pueden ser un carácter simple, tal como =, o múltiples caracteres o compuestos. tales como : = o ++. Una tercera categoría se compone de los identificadores, que por lo común se definen como secuencias de letras y dígitos que comienzan con una letra. Una categoría final se compone de literales o constantes, que incluyen constantes numéricas tales como 42 y 3.14159, literales de cadena como "hola, mundo" y caracteres tales como "a" y "b. Aquí describiremos algunas de estas expresiones regulares típicas y analizaremos algunas otras cuestiones relacionadas con el reconocimiento de tokens.

Números. Los números pueden ser solo secuencias de dígitos (números naturales), o números decimales, o números con un exponente (indicado mediante una "e" o "E). Por ejemplo, 2.71E-2 representa el número .0271. Podemos escribir definiciones regulares para esos números como se ve a continuación

$$nat = [0-9]+$$

$$natconSigno = (+|-)?\ nat$$

$$número = natconSigno("."\ nat)?(E\ natconsigno)?$$

Aquí escribimos el punto decimal entre comillas para enfatizar que debería ser generado directamente y no interpretado como un metacarácter.

Identificadores y Palabras Reservadas. Las palabras reservadas son las más simples de escribir como expresiones regulares: están representadas por sus secuencias fijas de caracteres. Si quisiéramos recolectar todas las palabras reservadas en una definición, escribiríamos algo como:

$$reservada = if\ |\ while\ |\ do\ |\ \ldots$$

Los identificadores, por otra parte, son cadenas de caracteres que no son fijas. Un identificador debe comenzar, por lo común, con una letra y contener sólo letras y dígitos. Podemos expresar esto en términos de definiciones regulares como sigue:

$$letra = [a-zA-Z]$$

$$digito = [0-9]$$

$$identificador = letra(letra\ |\ digito)*$$

Comentarios. Los comentarios por lo regular se ignoran durante el proceso del análisis Léxico. No obstante, un analizador léxico debe reconocer los comentarios y descartarlos. Por consiguiente, necesitaremos escribir expresiones regulares para comentarios, aun cuando un analizador Léxico pueda no tener un token constante explícito (podríamos llamar a estos pseudotokens). Los comentarios pueden tener varias formas diferentes. Suelen ser de formato libre o estar rodeados de delimitadores tales como

{este es un comentario en Pascal}

/*este es un comentario de C*/

o comenzar con un carácter o caracteres especificados y continuar hasta el final de la línea, como en

; este es un comentario de Sheme

-- este es un comentario de Ada

No es difícil escribir una expresión regular para comentarios que tenga delimitadores de carácter simple, tal como el comentario de Pascal, o para aquellos que partan de algún(os) caracteres) especificado(s) hasta el final de la línea. Por ejemplo, el caso de1 comentario de Pascal puede escribirse como

{(~}*}

donde escribimos ~} para indicar "not }" y donde partimos del supuesto de que el carácter } no tiene significado como un metacarácter. (Una expresión diferente debe escribirse para Lex, la cual analizaremos más adelante en este capítulo.) De manera similar, un comentario en Ada se puede hacer coincidir mediante la expresión regular

$$--(\sim nuevalínea)*$$

en la cual suponemos que nuevalínea corresponde al final de una línea (lo que se escribe como \n en muchos sistemas), que el carácter "-" no tiene significado como un metacarácter, y que el final de la línea no está incluido en el comentario mismo.

Es mucho más difícil escribir una expresión regular para el caso de los delimitadores que tienen más de un carácter de longitud, tal como los comentarios de C. Para ver esto considere el conjunto de cadenas $ba\ldots$ (ausencia de apariciones de ab)$\ldots ab$ (utilizamos $ba\ldots ab$ en lugar de los delimitadores de C /*…*/ ya que el asterisco, y en ocasiones la diagonal, es un metacarácter que requiere de un manejo especial). No podemos escribir simplemente

$$ba(\sim(ab))^*ab$$

porque el operador "not" por lo regular está restringido a caracteres simples en lugar de cadenas de caracteres. Podemos intentar escribir una definición para $\sim(ab)$ utilizando $\sim a$, $\sim b$ y $\sim(a|b)$, peco esto es no trivial. Una solución es

$$b^*(a^*\sim(a|b)^*)^*a^*$$

pero es difícil de leer (y de demostrar correctamente). De este modo, una expresión regular para los comentarios de C es tan complicada que casi nunca se escribe en la práctica. De hecho, este caso se suele manejar mediante métodos ex profeso en los analizadores léxicos reales.

Por último, otra complicación en el reconocimiento de los comentarios es que, en algunos lenguajes de programación, los comentarios pueden estar anidados. Por ejemplo, Modula-2 permite comentarios de la forma

(* esto es (* un comentario de *)en Modula-2 *)

Los delimitadores de comentario deben estar exactamente pareados en dichos comentarios anidados, de manera que lo que sigue no es un comentario legal de Modula-2:

(* esto es (* ilegal en Modula-2 *))

La anidación de los comentarios requiere que el analizador léxico cuente el número de los delimitadores.

Ambigüedad, Espacios en Blanco y Búsqueda hacia delante. A menudo en la descripción de los tokens de lenguajes de programación utilizando expresiones regulares, algunas cadenas se pueden definir mediante varias expresiones regulares diferentes. Por ejemplo, cadenas tales como if y while podrían ser identificadores o palabras clave. De manera semejante, la cadena <> se podría interpretar como la representación de dos tokens ("menor que" y "mayor que") o como un token simple ("no es igual a"). Una definición de lenguaje de programación debe establecer cuál interpretación se observará, y las expresiones regulares por sí mismas no pueden hacer esto. En realidad, una definición de lenguaje debe proporcionar reglas de no ambigüedad que implicarán cuál significado es el conveniente para cada uno de tales casos.

Dos reglas típicas que manejan los ejemplos que se acaban de dar son las siguientes. La primera establece que, cuando una cadena puede ser un identificador o una palabra clave, se prefiere por lo general la interpretación corno palabra clave. Esto se da a entender mediante el uso del término **palabra reservada**, lo que quiere decir que es simplemente una palabra clave que no puede ser también un identificador. La segunda establece que, cuando una cadena puede ser un token simple o una secuencia de varios tokens, por lo común se prefiere la interpretación del token simple. Esta preferencia se conoce a menudo como el principio de la subcadena más larga: la cadena más larga de caracteres que podrían constituir un token simple en cualquier punto se supone que representa el siguiente token:

Una cuestión que surge con el uso del principio de la subcadena más larga es la cuestión de los delimitadores de token, o caracteres que implican que una cadena más larga en el punto donde aparecen no puede representar un token. Los caracteres que son parte no ambigua de otros tokens son delimitadores. Por ejemplo, en la cadena $xtemp = ytemp$, el signo de igualdad delimita el identificador $xtemp$, porque = no puede aparecer como parte de un identificador. Los espacios en blanco, los retornos de línea y los caracteres de tabulación generalmente también se asumen como delimitadores de token: $while\ x\ \ldots$ se interpreta entonces como compuesto de dos tokens que representan la palabra reservada $while$ y el identificador de nombre x, puesto que un espacio en blanco separa las dos cadenas de caracteres. En esta situación a menudo es útil definir un pseudotoken de espacio en blanco, similar al pseudotoken de comentario, que sólo sirve al analizador léxico de manera interna para distinguir otros tokens. En realidad, los comentarios mismos por lo regular sirven como delimitadores, de manera que, por ejemplo, el fragmento de código en lenguaje C

$do\ /**/\ if$

representa las dos palabras reservadas do e if más que el identificador $doif$.
Una definición típica del pseudotoken de espacio en blanco en un lenguaje de programación es

$$espacioenblanco = (nuevalinea \mid blanco \mid tabulacion \mid comentario)+$$

donde los identificadores a la derecha representan los caracteres o cadenas apropiados. Advierta que, además de actuar como un delimitador de token, el espacio en blanco por lo regular no se toma en cuenta. Los lenguajes que especifican este comportamiento se denominan de formato libre. Las alternativas al formato libre incluyen el formato fijo de unos cuantos lenguajes como FORTRAN y diversos usos de la sangría en el texto, tal como la regla de fuera de lugar (véase la sección de notas y referencias). Un analizador Léxico para un lenguaje de formato libre debe descartar el espacio en blanco después de verificar cualquier efecto de delimitación del token.

Los delimitadores terminan las cadenas que forman el token pero no son pase del token mismo. De este modo, un analizador Léxico se debe ocupar del problema de la búsqueda hacia delante: cuando encuentra un delimitador debe arreglar que éste no se elimine del resto de la entrada, ya sea devolviéndolo a la cadena de entrada ("respaldándolo") o mirando hacia delante antes de eliminar el carácter de la entrada. En la mayoría de los casos sólo se necesita hacer esto para un carácter simple ("búsqueda hacia delante de carácter simple"). Por ejemplo, en la cadena $xtemp = ytemp$, el final del identificador $xtemp$ se encuentra cuando se halla el =, y el signo = debe permanecer en la entrada, ya que representa el siguiente token a reconocer.
Advierta también que es posible que no se necesite la búsqueda hacia delante para reconocer un token. Por ejemplo, el signo de igualdad puede ser el único token que comienza con el carácter =, en cuyo caso se puede reconocer de inmediato sin consultar el carácter siguiente.

En ocasiones un lenguaje puede requerir más que la búsqueda hacia delante de carácter simple, y el analizador Léxico debe estar preparado para respaldar posiblemente de manera arbitraria muchos caracteres. En ese caso, el alineamiento en memoria intermedia de los caracteres de entrada y la marca de lugares para un retroseguimiento se convierten en el diseño de un analizador Léxico.

5. BIBLIOGRAFÍA

[01] Kennet C. Louden, Construcción de Compiladores, Capitulo 2 - Thompson Paraninfo S.A. Edición 2004.