Compiladores
Facultad de Ciencias UNAM
Análisis Léxico

Lourdes Del Carmen González Huesca [*]

15 de febrero de 2024

El análisis léxico es la primer fase de un compilador, incluye un escaneo del archivo o código fuente para identificar cadenas sintácticamente correctas respecto a la definición del lenguaje de programación en cuestión. Este análisis se realiza mediante la clasificación de **tokens** para generar una secuencia de símbolos para ser procesada por la siguiente fase. Los tokens reconocidos serán almacenados en la Tabla de Símbolos. El análisis léxico favorece el reconocimiento de palabras significativas (tokens) eliminando detalles irrelevantes como espacios en blanco o comentarios en el archivo, reduciendo el tamaño del código hasta en un 80 %. Además se pueden detectar posibles errores de escritura o errores léxicos que son los errores de escritura en identificadores, palabras reservadas, operadores, etc.

Un token es una palabra o secuencia significativa de símbolos que consta de dos partes, el nombre y el valor (opcional) del atributo: la primera es la representación del tipo de unidad léxica y la segunda es el valor de dicha unidad. Cada token identificado tiene a lo más un atributo. El patrón de un token es la descripción de la forma del token y el **lexema** son los caracteres del programa fuente que son instancia del token. Puede suceder que el lexema y atributo/valor de un token coincidan o no, por ejemplo: una llave izquierda será un token `LBRAC` con atributo vacío pero el lexema es { y 37 es un entero que será un token `INT` con atributo y lexema 37.

Ejemplo 1 (Tokens)**.**

token	descripción informal	lexema	atributo
if	caracteres i f	`if`	–
else	caracteres e l s e	`else`	–
id	una letra seguida de letras o dígitos	`myvariable`	entrada en la tabla
num	cualquier valor numérico	`4.235`	entrada en la tabla
relop	cualquier operador de comparación	`<=`	`LE`
literal	cualesquiera símbolos que están delimitados por comillas dobles	`" ... "`	entrada en la tabla

Los espacios en blanco, los saltos de línea y demás espacios para sangría (en inglés *indentation*) ayudan a separar los lexemas pero no serán conservados para las siguientes fases del compilador. Además estos caracteres son convenciones de alto nivel, particulares a cada lenguaje de programación que también pueden ayudar a identificar tokens, estructuras de control y palabras reservadas en esta primera fase del compilador. Los comentarios son tratados como espacios en blanco aunque algunos son especiales para la generación de documentación del código fuente.

Para obtener un mejor diseño en el analizador, opcionalemente se puede dividir en dos procesos:

[*]Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

1. ***Scanner***: proceso que elimina los comentarios y los espacios en blanco.

2. ***Lexer***: proceso para producir la secuencia de tokens.

La secuencia de tokens será la entrada del analizador sintáctico, que es la siguiente fase, por lo tanto es necesario que cada token conserve su nombre y atributo. Durante el análisis léxico existe una interacción con la Tabla de Símbolos para almacenar los valores de los tokens, así la siguiente fase recibirá una cadena de tokens o pares (nombre y valor) donde el valor apunta a una entrada en la tabla.

Un analizador léxico debe procesar una cadena de símbolos y devolver los tokens reconocidos, veamos un ejemplo antes de mencionar la forma de especificación y de describir las características de su implementación.

Ejemplo 2 (Secuencia de caracteres a tokens)**.**

if	(	b	==	0	)	{	a	=	0	;	}

```
IF; LPAREN; Ident("b"); RELOP(EQEQ); Int(0); RPAREN; LBRAC; Ident("a"); EQ; Int(0); SEMI;
                                                                                RBRACE
```

Especificación de un analizador léxico

Para analizar el archivo de entrada, caracter por caracter es necesario definir las formas posibles de composición de éstos para reconocer palabras permitidas por el lenguaje de programación. Este proceso se formaliza a través de las gramáticas para definir lenguajes y de los autómatas finitos para reconocer palabras. La especificación establece el tipo de cadenas de entrada que se aceptarán y el tipo de tokens que se le asociarán respectivamente, por lo tanto, la representación ideal para este fin son las expresiones regulares.

Consideración: en esta nota y en el curso en general, se supone conocimiento previo como Teoría de cadenas, lenguajes, expresiones regulares, autómatas finitos, sus tipos y transformaciones entre ellos.

Ejemplo 3 (Reconocimiento de números reales)**.** La expresión regular que define números reales considerando a los dígitos d de cero a nueve es:

$$dd^\star\left(.d^\star + (\varepsilon + .d^\star)\left((e + \bar{+} + \bar{-})\, dd^\star\right)\right.$$

donde $\bar{+}$ y $\bar{-}$ representan los símbolos para positivo y negativo. Se deja al lector el diseño de un autómata que reconozca el mismo lenguaje que esta expresión regular.

Implementación de un analizador léxico

La tarea de un lexer es buscar los lexemas (instancias de tokens) del programa y hacerlo de una forma *eficiente*. Existen varias formas de realizar esta tarea:

1. ***Ad-hoc lexers***: Son escritos a mano y están basados en la técnica *look-ahead character* para determinar si se trata de un espacio o de un token, y en este último caso determinar el tipo de token. Una desventaja de estos lexers es que su diseño no está estandarizado y es difícil darle mantenimiento.

2. ***Generadores automáticos***: Automatizan la obtención de un analizador léxico a partir de un conjunto de expresiones regulares a través de construir autómatas finitos deterministas partiendo de la especificación dada.

Actualmente los lexers suelen ser creados con un generador automatizado. Los patrones que reconocerá serán las palabras reservadas, nombres de variables, símbolos extras, paréntesis, operadores, números, etc.

Para poder identificar diferentes patrones se usa la técnica de coincidencia más extensa, la cual permite leer símbolos hacia adelante para reconocer la secuencia más larga que es un token válido.

La regla de coincidencia más extensa posible permite identificar palabras reservadas e identificadores al procesar la cadena más larga posible mediante las transiciones de los autómatas que definen el lenguaje. Por ejemplo, la cadena `ifx = 0` debe ser reconocida como un token ID con valor `ifx`, seguido de un token RELOP con valor `EQ` y al final un token NUM con valor `0`. En cambio la cadena `if x = 0` debe ser reconocida como un token IF, seguido de un token RELOP con valor `EQ` y al final un token NUM con valor `0`. Obsérvese que en este punto no se está analizando la correctud de las cadenas ni el significado de ellas, mucho menos el resultado del cómputo que pudiera generar. Por lo tanto, ambas cadenas son reconocidas sin enviar error léxico.

La implementación del analizador es usando un autómata finito determinista, veamos *grosso modo* los pasos para obtener un analizador léxico:

1. especificar los tipos de tokens a ser reconocidos usando expresiones regulares para cada uno;
2. convertir las expresiones regulares en autómatas finitos, éstos pueden ser no deterministas incluso con transiciones épsilon pero es deseable que sean deterministas;
3. unir los autómatas en uno más grande que tenga la opción de escoger entre los diferentes autómatas de las expresiones regulares;
4. transformar el autómata anterior en uno determinista y opcionalmente minimizarlo;
5. implementar el autómata respetando la regla de coincidencia más extensa posible al almacenar el índice del último token reconocido y el índice de mayor alcance en la cadena de entrada, regresando error si no es posible reconocer alguna palabra.

Cabe mencionar que uno de los retos para el lexer es implementar de forma eficiente la Tabla de Símbolos, ya que en ella se guarda toda la información del stream de tokens del código fuente.

Generadores de analizadores léxicos

Existen herramientas que automatizan la obtención de un analizador léxico llamados generadores de analizadores léxicos o *Lexer Generators* en inglés, a partir de una especificación de expresiones regulares. Reciben una lista de expresiones regulares $\mathsf{R}_1, \mathsf{R}_2, \ldots, \mathsf{R}_n$ que definen cada token del lenguaje fuente y una acción correspondiente al token, es decir un pedazo de código para ser ejecutado cuando la expresión regular coincide. Al final genera un código escaneado donde se decide si una cadena de entrada es de la forma $(\mathsf{R}_1 \mid \mathsf{R}_2 \mid \cdots \mid \mathsf{R}_n)^\star$ y cada vez que se reconoce el token o (cadena más larga de algún token) se ejecuta la acción asociada.

Los generadores realizan los cómputos necesarios siguiendo la implementación descrita en la sección anterior:

1. considerar cada expresión regular R_i y su acción asociada A_i;
2. calcular un autómata finito no deterministas para $(\mathsf{R}_1 \mid \mathsf{R}_2 \mid \cdots \mid \mathsf{R}_n)$
3. transformar el autómata anterior en uno determinista y minimizarlo;
4. producir la tabla de transiciones que define al autómata.

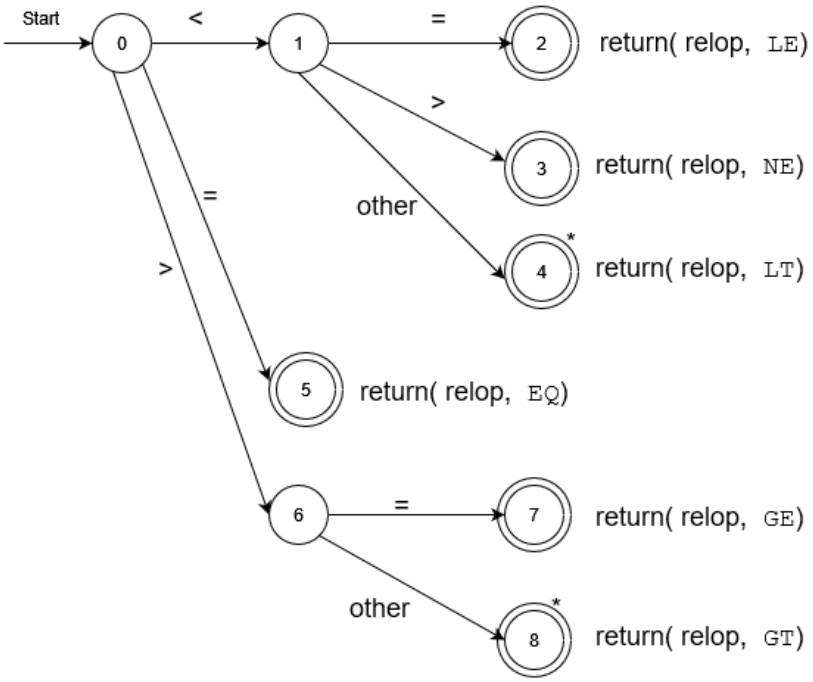
Además el autómata resultante respeta la regla de coincidencia más extensa posible al dar prioridad a las transiciones posibles y si no se llega a un estado final se reporta error léxico.

Algunas referencias de generadores léxicos son:

- `Lex`: generador léxico para Unix que obtiene un lexer en C.
  `https://en.wikipedia.org/wiki/Lex_(software)`
  `http://dinosaur.compilertools.net/lex/index.html`
- `Flex`: *Fast LEXical analyzer generator*
  `https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html#SEC1`
- `ocamllex`: generador léxico en Ocaml.
  `https://courses.softlab.ntua.gr/compilers/2015a/ocamllex-tutorial.pdf`
  `https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`
- Python Lex-Yacc
  `https://www.dabeaz.com/ply/`

Lexer para operadores de comparación

Veamos un ejemplo. Las estructuras a reconocer serán los operadores de comparación.
Éstas son definidas de la siguiente manera:
relop →< | > | <= | >= | = | <>
El autómata que puede reconocerlas es el siguiente:



Veamos que para cada elemento de *relop* existe un conjunto ordenado de transiciones que llegan al estado final que lo reconoce. Por ejemplo, para el elemento <> basta con ir del estado cero, luego al estado 1, luego al estado 3 y se llega al estado terminal que lo reconoce.

En el diagrama podemos ver que para cada estado final existe un *return* que es la acción de creación del token.

También es importante recalcar que el autómata anterior respeta la regla de coincidencia más larga. Esto se logra con el estado 4 y 8 marcados con un *****. Ya que aunque en el estado 1 y 6 sería posible ya reconocer a los tokens < y > respectivamente, no se marcan como estados terminales ya que aún es posible derivar otro token de esos estados.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

[3] Frank Pfenning. Notas del curso (15-411) Compiler Design. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`, 2014.

[4] François Pottier. Presentaciones del curso Compilation (inf564) École Polytechnique, Palaiseau, Francia. `http://gallium.inria.fr/~fpottier/X/INF564/`, 2016. Material en francés.

[5] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[6] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos. `https://www.cis.upenn.edu/~cis341/current/`, 2018.