Análisis Léxico
Compiladores 2020-1, Nota de clase 2

Lourdes Del Carmen González Huesca

17 de agosto de 2019
Facultad de Ciencias UNAM

El análisis léxico es la primer fase de un compilador, incluye un escaneo del archivo o código fuente para identificar cadenas sintácticamente correctas mediante la clasificación de **tokens** generando una secuencia de símbolos para ser procesada por la siguiente fase. Este análisis favorece el reconocimiento de palabras significativas (tokens) eliminando detalles irrelevantes como espacios en blanco o comentarios en el archivo, reduciendo el tamaño del código 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 significativa que consta de dos partes, el nombre y su atributo: la primera es la representación del tipo de unidad léxica y la segunda es el valor de dicha unidad. Cada token 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.

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 o estructuras de control. Los comentarios son tratados como espacios en blanco aunque algunos son especiales para la generación de documentación del código fuente. Una secuencia de tokens serán la entrada del analizador sintáctico, por lo tanto es necesario que cada uno tenga su nombre y atributo.

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

1. ***Scanner***: proceso que elimina los comentarios y los espacios en blanco.
2. ***Lexer***: proceso para producir la secuencia de tokens.

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.

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`

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 (Procesamiento de una cadena de símbolos)**.** Dada la cadena `if (b==0) { a=0 }` se identifican los siguientes tokens con sus atributos correspondientes:
`IF; LPAREN; ID("b"); EQEQ; NUM(0); RPAREN; LBRACE; ID("a"); EQ; NUM(0); 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 objeto. 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:

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

donde $\bar{+}$ y $\bar{-}$ representan los símbolos para postivo 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 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 autómtas 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.

Generadores de analizadores léxicos

Existen herramientas que automatizan la obtención de un analizar léxico llamados generadores de analizaroes 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`
- `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`

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, 2014. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`.

[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, 2018. `https://www.cis.upenn.edu/~cis341/current/`.