PLAN DE ESTUDIOS 2008

Agrego el Plan de Estudios 2008, tal como se determinó en Rectorado para todas las Regionales, para que se comprenda porqué he incluido en el libro los complejos capítulos 3 y 4 del Volumen 2.

Asignatura: SINTAXIS Y SEMÁNTICA DE LOS LENGUAJES

OBJETIVOS:

1. Conocer los elementos propios de la sintaxis y semántica de los lenguajes de programación.
2. Conocer los lenguajes formales y autómatas.
3. Comprender conceptos y procedimientos de las gramáticas independientes (libres) de contexto y gramáticas regulares para especificar la sintaxis de los lenguajes de programación.
4. Utilizar distintos tipos de autómatas y distintos tipos de notaciones gramaticales.
5. Comprender el procesamiento de lenguajes y, en particular, **el proceso de compilación**.

CONTENIDOS MÍNIMOS:

1. Gramática y Lenguajes Formales.
2. Jerarquía de Chomsky.
3. **Autómatas Finitos**. Expresiones Regulares y **su aplicación al Análisis Léxico**.
4. Gramáticas Independientes del Contexto.
5. **Autómatas Push-Down y su Aplicación al Análisis Sintáctico**.
6. **Otros tipos de Analizadores Sintácticos**.
7. Máquinas de Turing.
8. Introducción a las Semánticas.

CAP. 3 INTRODUCCIÓN AL PROCESO DE COMPILACIÓN ACLARACIONES Y AGREGADOS

. . . . . .

3.1 CONCEPTOS BÁSICOS [La Estructura del Compilador Micro]

1º El SCANNER lee un programa fuente desde un archivo de texto y produce un flujo de representaciones de tokens. El SCANNER se implementa como una función que produce la representación de un token cuando es llamado por el PARSER.

2º El PARSER procesa los tokens hasta que reconoce una estructura sintáctica que requiere un procesamiento semántico. Entonces, invoca a una RUTINA SEMÁNTICA. Algunas de estas rutinas semánticas usan la información de la representación de los tokens.

3º Las RUTINAS SEMÁNTICAS producen una salida en forma de instrucción para una MV.

4º La TABLA DE SÍMBOLOS es usada solo por las **rutinas semánticas** [Fischer, pág. 24].

3.2 UN COMPILADOR SIMPLE

3.2.4 UN ANALIZADOR LÉXICO PARA MICRO

Recuerde que el prototipo de la función que implementa al Scanner es:

```
TOKEN Scanner (void);
```

3.2.4.1 EL AFD PARA IMPLEMENTAR EL SCANNER

El programador debe desarrollar ciertas funciones auxiliares:

1º `void ErrorLexico(int);` que recibe el carácter que produce un error léxico, despliega un mensaje y guarda una "bandera" en una variable global para que el resto del compilador sepa que hay un error insalvable.

2º `void AgregarCaracter(int);` que añade un carácter al buffer, que es un vector de caracteres global utilizado para almacenar los caracteres de los identificadores y los dígitos de las constantes en la medida que son reconocidos por el AFD. Todas las rutinas que forman el Parser y las rutinas semánticas pueden acceder a este buffer.

3º `TOKEN EsReservada(void);` que, dado el identificador reconocido por el AFD y que está almacenado en el buffer, retorna el token que le corresponde (`ID` o el código de alguna de las cuatro palabras reservadas).

4º `void LimpiarBuffer(void);` que "vacía" el buffer para el próximo uso (simplemente, coloca el carácter nulo en la primera posición del vector).

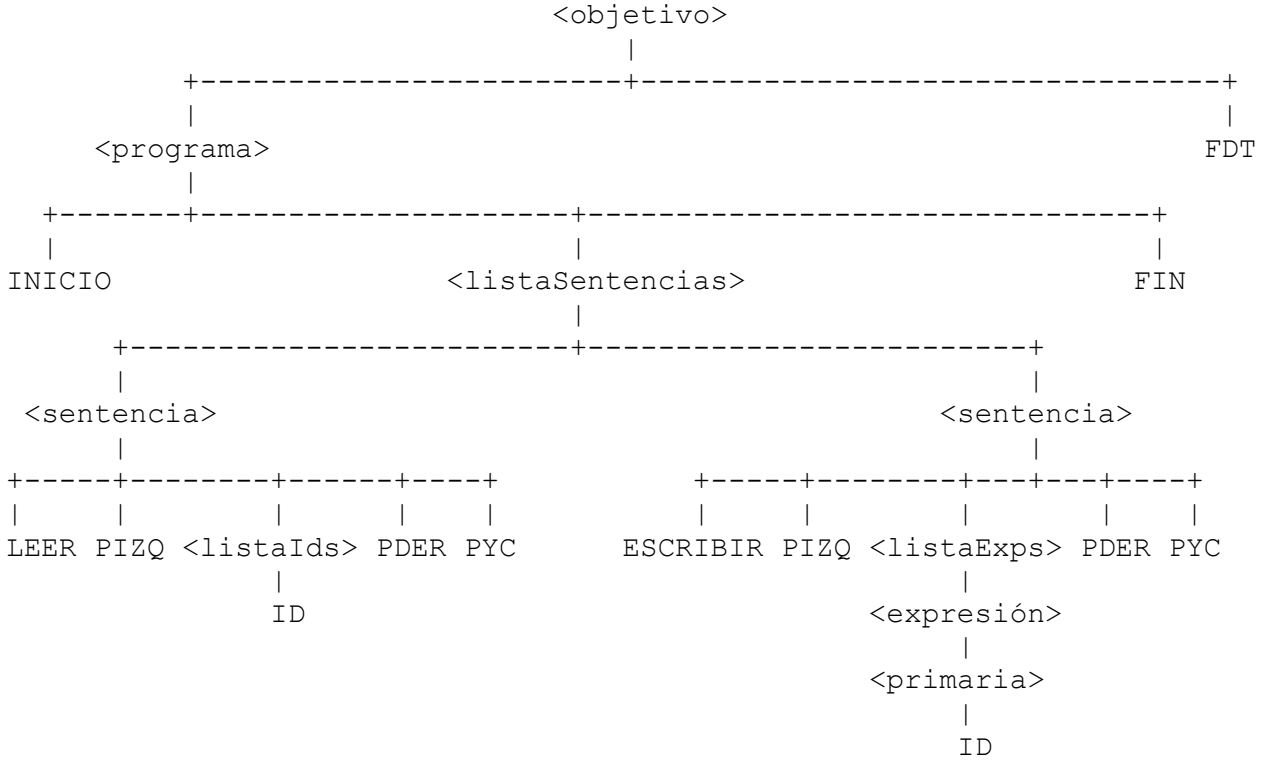
3.2.5 UN PARSER PARA MICRO

Los terminales de la GIC utilizada en el Análisis Sintáctico son los tokens que retorna el Scanner y no los caracteres o las secuencias de caracteres – según corresponda – que se encuentran en el Programa Fuente.

Un Árbol de Análisis Sintáctico parte del axioma de una GIC y representa la derivación de una construcción (declaración, sentencia, expresión, bloque y hasta el programa completo). Un AAS tiene las siguientes propiedades:

– La raíz del AAS está etiquetada con el axioma de la GIC.
– Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.
– Cada nodo interior está etiquetado con un noterminal.

Ejemplo:
Sea el programa `inicio leer(a); escribir(a); fin`
El AAS será:



Si para cada terminal (token) se sigue un paso más, aparece el programa fuente seguido de ese token especial llamado fdt.

Como ya se ha visto en diferentes PAS, existen dos funciones que vinculan al Parser con el Scanner:

1º Match, con prototipo `void Match(TOKEN);`

2º ProximoToken, con prototipo `TOKEN ProximoToken(void);`

El procedimiento Match con argumento t, es decir, Match(t), invoca al Scanner para obtener el próximo token. Si el token obtenido por el Scanner es t, es decir, coincide con el argumento con el cual se invoca a Match, entonces todo es correcto porque coinciden y el token es guardado en una variable global llamada tokenActual.

En cambio, si el token obtenido por el Scanner no coincide con el argumento t, entonces se ha producido un Error Sintáctico; se debe emitir un mensaje de error y tener en cuenta esta situación porque el proceso de compilación ya no puede ser correcto.

Por otro lado, ProximoToken obtiene el próximo token, tal como su nombre lo indica.

Como en ciertos casos Match y ProximoToken son llamadas una a continuación de la otra (vea el PAS Sentencia), debe haber una variable global con una "bandera" que indique si el token ya fue obtenido. [El libro de Fischer no da información alguna al respecto.]

EJERCICIO 1: ¿Se da cuenta por qué?

Veamos otro PAS en el que se da esta situación. La sentencia de tipo:

```
LEER ( <listaIdentificadores> );
```

utiliza el PAS ListaIdentificadores al ser analizada sintácticamente (vea el PAS Sentencia). Lo construimos así:

```
void ListaIdentificadores (void) {
   /* <listaIdentificadores> -> <identificador> {, <identificador>} */
   Match(ID);
   while (ProximoToken() == COMA) {
      Match(COMA);  Match(ID);
   }
}
```

EJERCICIO 2: ¿Qué diferencia fundamental encuentra en las actividades que realizan estas dos funciones?
EJERCICIO 3: Describa cómo trabaja el PAS ListaIdentificadores.
EJERCICIO 4: Explique qué sucede cuando `ProximoToken()!=COMA`.