Teoría de Autómatas y Lenguajes Formales

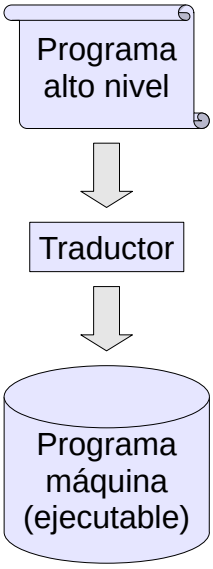
Prácticas:

Desarrollo de un mini-compilador

Contenido

1. Introducción

Cuando se desarrolló el primer ordenador digital, la única manera de comunicarse con él consistía en hablar su mismo idioma, su lenguaje máquina, compuesto por 1's y 0's.



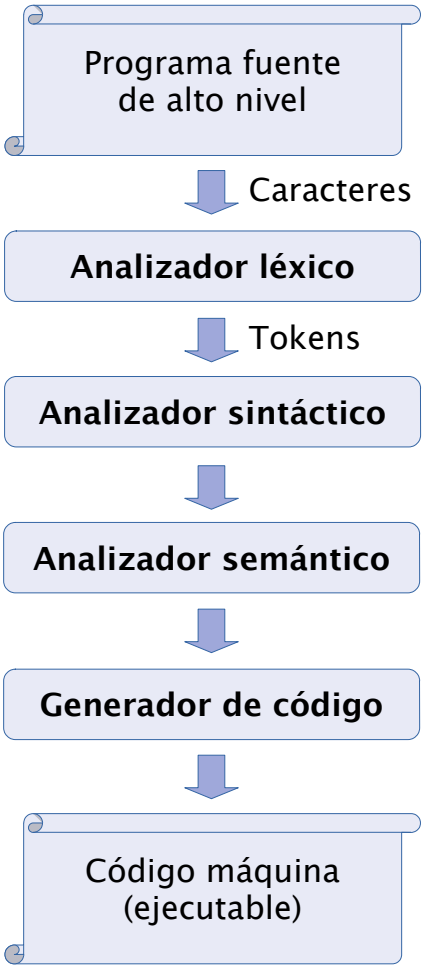
Esta aproximación resultaba realmente complicada, especialmente con programas de mediana complejidad. Por ello, comenzaron a aparecer lenguajes de más alto nivel, es decir, más próximos al ser humano.

El primer lenguaje fue el ensamblador, que simplemente asociaba etiquetas a las instrucciones máquina. Posteriormente surgieron los lenguajes de alto nivel (C, Pascal, ...).

Sin embargo, a pesar de que se elaboran programas con lenguajes de programación de alto nivel, el idioma de la máquina no ha cambiado. Es necesario, por tanto, contar con traductores que se encarguen de traducir los programas de alto nivel al idioma del computador, para que dichos programas sean ejecutables.

2. Estructura de un compilador

Un compilador se puede dividir en los módulos que aparecen en la siguiente figura:



2.1. Analizador léxico

Un analizador léxico lee los caracteres del texto a analizar y a partir de él obtiene tokens, que son agrupaciones de caracteres con un significado especial.

El siguiente ejemplo es una sentencia SQL que permite obtener una lista de los clientes almacenados en la tabla `CLIENTES`, ordenados por su nombre:

```
SELECT CODIGO,NOMBRE FROM CLIENTES ORDER BY NOMBRE
```

El analizador léxico de un compilador de SQL podría devolver los siguientes tokens (en negrita) a partir de la sentencia anterior:

- **`SELECT`**
- **`ID`** `: CODIGO`
- **`COMA`**
- **`ID`** `: NOMBRE`
- **`FROM`**
- **`ID`** `: CLIENTES`
- **`ORDER`**
- **`BY`**
- **`ID`** `: NOMBRE`

Como se puede observar, cada token `ID` puede tener asociado un valor distinto, mientras que los tokens `SELECT`, `COMA`, `FROM`, `ORDER` y `BY` no.

Hay que tener en cuenta que un compilador no necesita guardar los tokens por su nombre completo, sino que asigna a cada uno de ellos un número, aunque en el caso de `ID`, del ejemplo anterior, habría que guardar también el valor asociado a cada uno.

El analizador léxico tampoco tiene por qué devolver tokens para todo lo que lea. Por ejemplo, en casi todos los lenguajes de programación se pueden escribir **comentarios**, es decir, texto que no afecta a la lógica del programa y que son útiles para el programador cuando tiene que revisar un código fuente con el que hace tiempo que no trabaja. Pues bien, los comentarios podrían ser ignorados completamente por el analizador léxico, aunque esto no es del todo cierto.

Por ejemplo, en Java, el compilador 'javac' ignora los comentarios del código fuente, pero la herramienta de documentación 'javadoc' sí que analiza los comentarios para construir a partir de ellos la documentación del código.

Realmente, un analizador léxico realiza una traducción desde un lenguaje de alto nivel a otro lenguaje más sencillo que será la entrada para el siguiente módulo del compilador, el **analizador sintáctico**.

Aunque los analizadores léxicos se pueden generar usando herramientas como **Lex**, en esta práctica se van a usar con fines didácticos los **autómatas finitos deterministas**, que se verán en el tema 2 de teoría.

Los autómatas finitos se pueden definir como grafos dirigidos. Más adelante se verá cómo traducir un autómata finito determinista en código fuente para que funcione como un analizador léxico.

2.2. Analizador sintáctico

El analizador sintáctico toma como entrada la secuencia de tokens que devuelve el analizador léxico, comprueba si éstos están en el orden correcto y extrae la información necesaria para los siguientes módulos del compilador, el analizador semántico y el generador de código.

En la sentencia del ejemplo anterior

```
SELECT CODIGO,NOMBRE FROM CLIENTES ORDER BY NOMBRE
```

el analizador sintáctico podría estructurar la información de la siguiente forma:

- Operación: `SELECT`
- Array de columnas:
  - `ID : CODIGO`
  - `ID : NOMBRE`
- Tabla: `CLIENTES`
- Ordenación: `NOMBRE`

Aunque los analizadores sintácticos se pueden generar usando herramientas como **Yacc**, en esta práctica se van a usar con fines didácticos las **gramáticas independientes del contexto**, que se verán en el tema 4 de teoría.

Una gramática extremadamente simplificada que permitiría analizar la sentencia anterior, podría ser la siguiente:

```
select    –▶ SELECT listaIds FROM listaIds where orderBy
listaIds  –▶ ID comaIds
comaIds   –▶ COMA ID comaIds | λ
where     –▶ WHERE expresión | λ
orderBy   –▶ ORDER BY ID ids | λ
expresión –▶ ...
```

Como se puede observar, una gramática no es más que un conjunto de reglas de producción, donde la parte izquierda de cada regla produce lo de la parte derecha. En el ejemplo anterior las palabras en mayúsculas son los símbolos terminales, que se corresponden a los tokens que provienen del análisis léxico; y las otras palabras, que contienen minúsculas, son los símbolos no terminales.

Como ya se ha visto (o se verá) en clase de teoría, el símbolo λ se corresponde con la cadena vacía. Por ejemplo, en la gramática anterior el símbolo '`where`' puede producir la cadena "`WHERE expresión`" o nada (la cadena vacía).

Para implementar un analizador sintáctico a partir de una gramática independiente del contexto se usará un analizador descendente recursivo, que se explicará más adelante.

2.3. Analizador semántico

El analizador semántico comprueba si es correcta la información que le proporciona el analizador sintáctico. Por ejemplo:

- Operación: `SELECT`
- Array de expresiones:
  - `ID : CODIGO`
  - `ID : NOMBRE`
- Tabla: `CLIENTES`
- Ordenación: `NOMBRE`

En el ejemplo anterior, el analizador semántico comprobaría que la operación `SELECT` es válida, que la tabla `CLIENTES` existe, que las columnas `CODIGO` y `NOMBRE` pertenecen a dicha tabla y que las columnas de ordenación, también pertenecen a la tabla.

En otros lenguajes de programación, como Java, el analizador semántico es el que permite definir variables, comprueba los tipos de datos, comprueba los nombres de clases y funciones,... Para todo ello, el analizador semántico debe disponer de varias tablas de símbolos.

2.4. Generador de código

El generador de código decide qué pasos hay que realizar para ejecutar el programa a partir de la información que recibe del analizador sintáctico y del semántico.

En algunos lenguajes, como C, genera código ejecutable directamente por el ordenador. En otros lenguajes, como Java y C#, genera un código intermedio que es ejecutable por una máquina virtual.

Sin embargo, en el caso de una base de datos, como en el ejemplo anterior de SQL, la generación de código se limita a decidir de qué tablas hay que leer ciertas columnas, con qué ordenación, qué condiciones hay que evaluar,...

Se podría decir que en una base de datos la generación de código realiza una planificación de ejecución en lugar de una generación de código ejecutable como tal.

3. El mini-compilador

El lenguaje del mini–compilador que se va a desarrollar es muy sencillo, pues no permitirá definir funciones ni estructuras de datos, pero sí que dispondrá de sentencias de control (condiciones y bucles) y tipos de datos sencillos (enteros y cadenas de caracteres) con los que se podrá hacer cálculos numéricos enteros, expresiones booleanas y concatenación de cadenas.

El lenguaje admitirá los comentarios al estilo de Java, es decir, comentarios de línea (`//`) y de bloque (`/*...*/`). Estos comentarios deberán ser ignorados por el analizador léxico.

3.1. Tokens del lenguaje

Los tokens del lenguaje serán obtenidos mediante el analizador léxico, el cual se debe diseñar como un gran autómata finito determinista implementado en la clase '**AFD.java**'. Esta clase heredará de '**Alex.java**' y utilizará la clase '**Token.java**'.

Token	Texto / descripción
ENTERO	`entero` (palabra reservada)
CADENA	`cadena` "
SI	`si` "
SINO	`sino` "
MIENTRAS	`mientras` "
FIN	`fin` "
IMPRIMIR	`imprimir` "
ID	Identificadores (nombres de variables).
INTVAL	Cualquier valor numérico entero.
STRVAL	Cadena de caracteres entre dobles comillas.
ASIGN	`:=` (operador de asignación)
SUM	`+` `-`
MUL	`*` `/`
REL	`=` `<>` `<` `<=` `>` `>=`
NEG	`!`
OR	`||`
AND	`&&`
IPAR	`(`
DPAR	`)`

3.2. Gramática del lenguaje

La sintaxis del lenguaje se puede definir mediante la siguiente gramática, en la que los símbolos terminales (tokens) están escritos en mayúsculas al estilo de las constantes del lenguaje Java, y los símbolos no terminales en minúsculas, al estilo de los nombres de las funciones, también del lenguaje Java.

Gramática del lenguaje

```
programa —► declaración bloque EOF

declaración —► CADENA ID declaración
             | ENTERO ID declaración
             | λ

bloque —► asignación bloque
        | impresión  bloque
        | condición  bloque
        | iteración  bloque
        | λ

asignación —► ID ASIGN expresión
impresión  —► IMPRIMIR expresión
condición  —► SI expresión bloque sino FIN

sino —► SINO bloque
      | λ

iteración —► MIENTRAS expresión bloque FIN
expresión —► vor vor1


vor   —► vand vand1                    vsum  —► vmul vmul1

vor1  —► OR vor vor1                   vsum1 —► SUM vsum vsum1
       | λ                                      | λ

vand  —► vrel vrel1                    vmul  —► IPAR expresión DPAR
       | NEG vrel vrel1                         | valor

vand1 —► AND vand vand1                vmul1 —► MUL vmul vmul1
       | λ                                      | λ

vrel  —► vsum vsum1                    valor —► ID
       | SUM vsum vsum1                         | INTVAL
                                                | STRVAL
vrel1 —► REL vrel
       | λ
```

La gramática se debe implementar con el método **descendente-recursivo** en la clase '**ADR.java**', que es una subclase de '**ASin.java**'. Para ello se debe seguir la gramática como guía casi "al pie de la letra".

En la implementación, cada símbolo no terminal será una función en cuyo cuerpo se implementará aquello que pueda producir dicho símbolo. El cuerpo consistirá en llamadas a otros símbolos no terminales, en llamadas a la función **`tokenRead`** para los tokens y en condiciones para decidir qué regla aplicar cuando haya más de una opción disponible.

3.3. Análisis semántico y generación de código

El análisis semántico y la generación de código se realizará durante el análisis descendente–recursivo. Por tanto, habrá que insertar llamadas a las funciones **`code*`** en la clase **`ADR`**.

Por ejemplo, tras la declaración de una variable, el analizador sintáctico deberá llamar a **`codeVariableInteger`** o a **`codeVariableString`**, dependiendo del tipo de declaración, para que el analizador tome nota de las nuevas variables. Cuando se encuentre un identificador en una expresión, el analizador deberá llamar a **`codeVariableExpression`** para que se compruebe que la variable existe y se genere el código ejecutable necesario.

La gramática del lenguaje está diseñada de forma que implementa la precedencia entre operadores y facilita la transformación de las expresiones desde notación **infija**, tal como aparecen en el código fuente, a notación **postfija**, donde en una expresión se escriben primero los operandos y al final el operador.

Es necesario generar el código usando notación postfija porque la máquina virtual que ejecutará el código generado por el compilador utiliza una pila para evaluar las expresiones.

En la página siguiente se muestra un ejemplo de código fuente que calcula los números primos hasta el 100 y el código ejecutable que debe generar el compilador. En el código ejecutable se puede ver claramente el uso de la notación postfija.

El único operador que en la generación de código será diferente del texto del token, es la **negación** de un número entero. Este operador se escribe como "**`-1`**". Se pueden ver todos los operadores en la función **`codeOperator`**.

Código fuente

```
entero n
entero d
n := 2

mientras n < 100
    d := 2

    mientras d * d <= n && n / d * d <> n
        d := d + 1
    fin

    si d * d > n
        imprimir n +" primo!"
    sino
        imprimir n +" no"
    fin

    n := n + 1
fin
```

Código ejecutable

```
 0:  int  decl  n
 1:  int  decl  d
 2:  int  var   n
 3:  int  cte   2
 4:       :=
 5:       while
 6:  int  var   n
 7:  int  cte   100
 8:       <
 9:       if    58
10:  int  var   d
11:  int  cte   2
12:       :=
13:       while
14:  int  var   d
15:  int  var   d
16:       *
17:  int  var   n
18:       <=
19:  int  var   n
20:  int  var   d
21:       /
22:  int  var   d
23:       *
24:  int  var   n
25:       <>
26:       &&
27:       if    34
28:  int  var   d
29:  int  var   d
30:  int  cte   1
31:       +
32:       :=
33:       goto  13
34:       end
35:  int  var   d
36:  int  var   d
37:       *
38:  int  var   n
39:       >
40:       if    46
41:  int  var   n
42:  str  cte   " primo!"
43:       +
44:       print
45:       goto  51
46:       else
47:  int  var   n
48:  str  cte   " no"
49:       +
50:       print
51:       end
52:  int  var   n
53:  int  var   n
54:  int  cte   1
55:       +
56:       :=
57:       goto  5
58:       end
```

4. Planificación del desarrollo de la práctica

- 1ª sesión: presentación del mini–compilador
- 2ª sesión
- 3ª sesión: análisis léxico
- 4ª sesión
- 5ª sesión: análisis sintáctico
- 6ª sesión: análisis semántico y generación de código
- 7ª sesión
- 8ª sesión (1 hora): finalización del mini–compilador

5. Entrega de la práctica

La fecha límite para entregar la práctica es el 16 de Enero, y se debe realizar mediante una tarea de PoliformaT.

Si la práctica se ha realizado en pareja, debe entregarla sólo uno de los dos componentes, y además, hay que indicar con quién ha realizado la práctica en el campo de texto de la tarea de PoliformaT.

Se comprobará la compilación y ejecución de los ejemplos del fichero '**test.zip**':

- factorial
- negar
- primos
- xor