

**INFORME**

**TP FINAL COMPILADORES**

**ALUMNOS:**

* **Albornoz, Carlos 66**
* **Alegre, Gustavo 66**
* **Moura, Damian Elias 66422 d.eliasmoura@hotmail.com**

**PROFESOR:**

* **Ing. Ruidias, Hector**

**AÑO: 2018**

**INTRODUCCIÓN**

Mediante este informe se busca describir los pasos llevados a cabo para el desarrollo de un compilador (análisis léxico y sintáctico), que traduce un lenguaje fuente escrita en pseudocódigo a un lenguaje comprensible por la máquina, Lenguaje C.

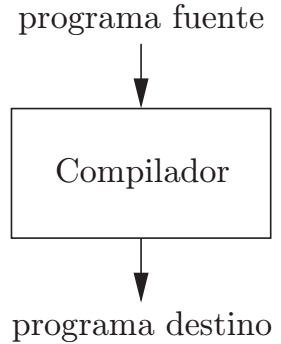
Con tal fin, se comenzara desarrollando brevemente los temas teóricos necesarios para comprender cómo se lleva a cabo proceso de traducción del lenguaje fuente al lenguaje destino realizador por el compilador. En particular se hablara con respecto a las fases de análisis de un compilador en particular, (A. Léxico, A. Sintáctico y A. Semántico).

Luego, se describirán las herramientas utilizadas para generar los distintos tipos de analizadores, y cómo se llevó a cabo su utilización.

Una vez desarrollados estos conceptos teóricos, se describe la estructura de la gramática del lenguaje que se desea traducir, definiendo sus reglas y componentes léxicos.

**INTRODUCCIÓN TEÓRICA**

**COMPILADOR**

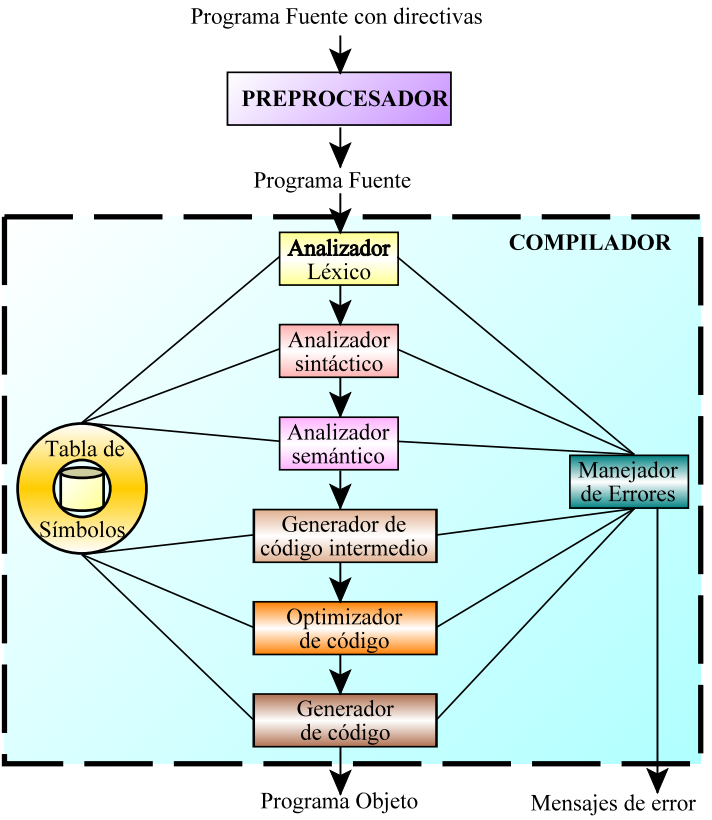
Un compilador es un programa que puede leer un programa que esta escrito en un lenguaje (programa fuente escrito en un lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (programa destino escrito en un lenguaje destino). Normalmente esta traducción es a un lenguaje maquina (C, C#, Java, etc), aunque también puede traducirlo a un código intermedio (Bytecode).

**Fig. 1 Compilador**

**ESTRUCTURA DE UN COMPILADOR**

Un compilador esta compuesto por dos etapas:

* Análisis (front-end):
  + se determina la estructura y el significado del lenguaje fuente.
  + divide el programa fuente en componentes e impone una estructura gramatical sobre ellas, la cual es usada para crear una representación intermedia del programa fuente (árbol sintáctico)
  + Fases que abarca:
    - A. léxico, A. sintáctico, A. semántico, generación de código intermedio, tabla de símbolos, manejo de errores.
* Síntesis (back-end):
  + Construye el programa destino a partir del árbol sitacosis y de la información en la tabla de símbolos.
  + Fases que abarca:
    - Optimización, generación de código intermedio, manejo de errores.

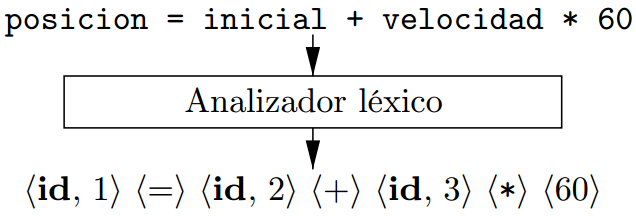


**Fig. 2 Fases de un compilador**

**ANÁLISIS LÉXICO (SCANNER)**

* Es la primer fase de un compilador.
* Su tarea principal es leer el flujo de caracteres de entrada que componen el programa fuente, agruparlos en secuencias significativas, conocidas como **lexemas** y produce como salida una secuencia de **tokens** para cada lexema en el programa fuente.
* Para cada lexema el A. Léxico produce como salida un **token** de la forma:

<nombre-token , valor-atributo>

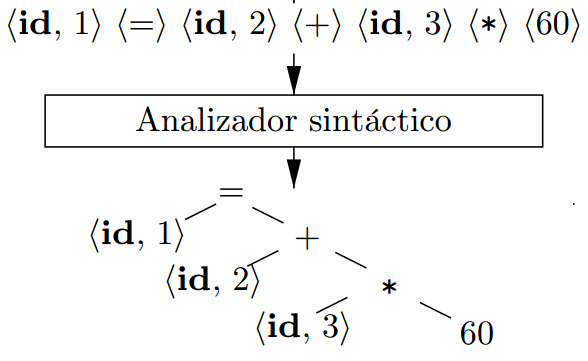


**Fig. 3 Analizador léxico (entrada y salida)**

* el componente nombre-token es un símbolo abstracto usado durante el análisis sintáctico, y el componente valor-atributo apunta a una entrada en la tabla de símbolos para ese token, necesario para el análisis semántico y la generación de código.

**ANÁLISIS SINTÁCTICO (PARSER)**

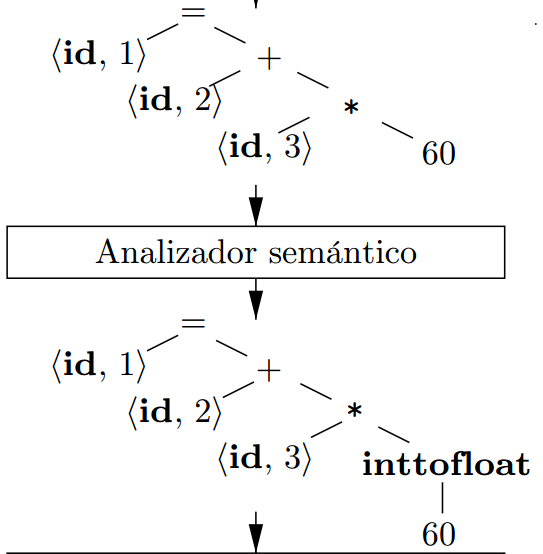
* comprueba que la estructura de los componentes básicos sea correcta según ciertas reglas gramaticales.
* a partir de los tokens determina la estructura sintáctica de las sentencias, y utiliza los primeros componentes de los tokens para crear la representación intermedia en forma de árbol (árbol sintáctico) que describa la estructura gramatical del flujo de tokens.
* Cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.



**Fig. 4 Analizador Sintáctico (entrada y salida)**

**ANÁLISIS SEMÁNTICO**

* comprueba que la estructura de los componentes básicos sea correcta según ciertas reglas gramaticales.
* Utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje fuente con al definición del lenguaje.
* recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.
* Una parte importante es la comprobación de tipos, donde verifica que cada operador tenga los operandos apropiados.



**Fig. 5 Analizador semántico (entrada y salida)**

**HERRAMIENTAS**

**LEX/FLEX: generador de analizador léxico**

Flex es una herramienta que nos permite especificar un analizador léxico mediante la especificación de expresiones regulares que describen patrones para identificar tokens.

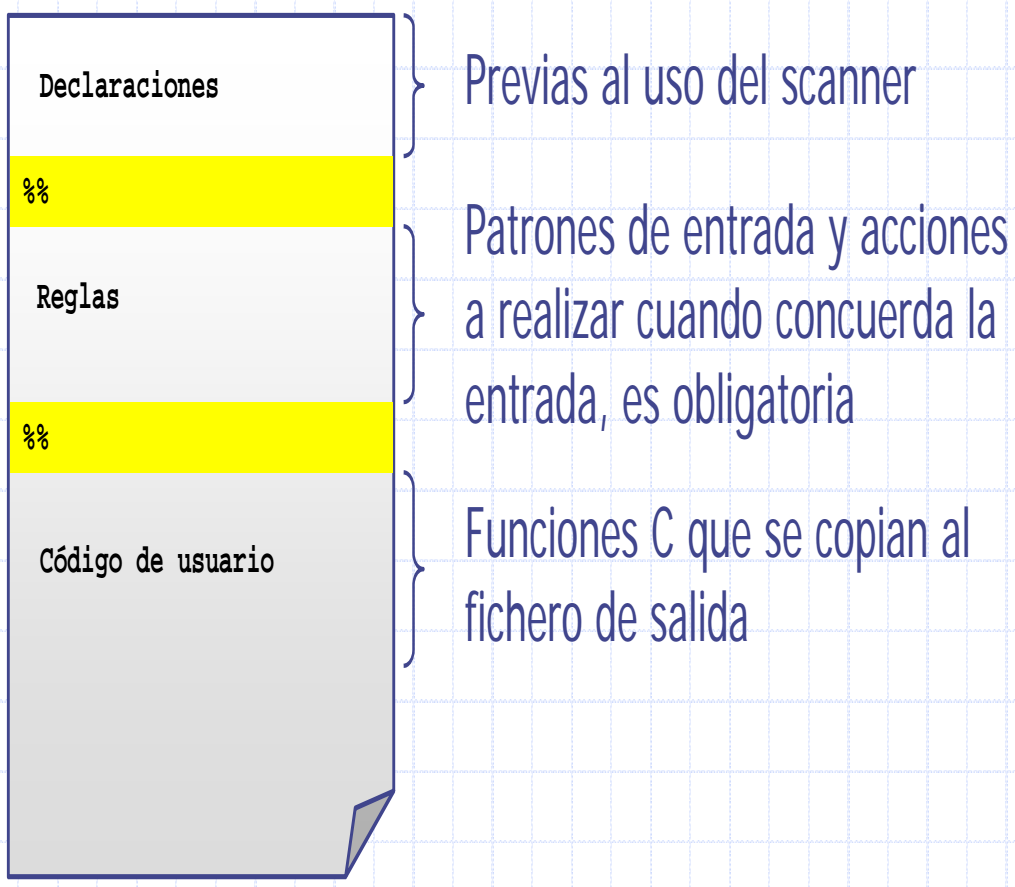
El analizador léxico de un compilador es el responsable de identificar los tokens en el fichero del programa fuente.

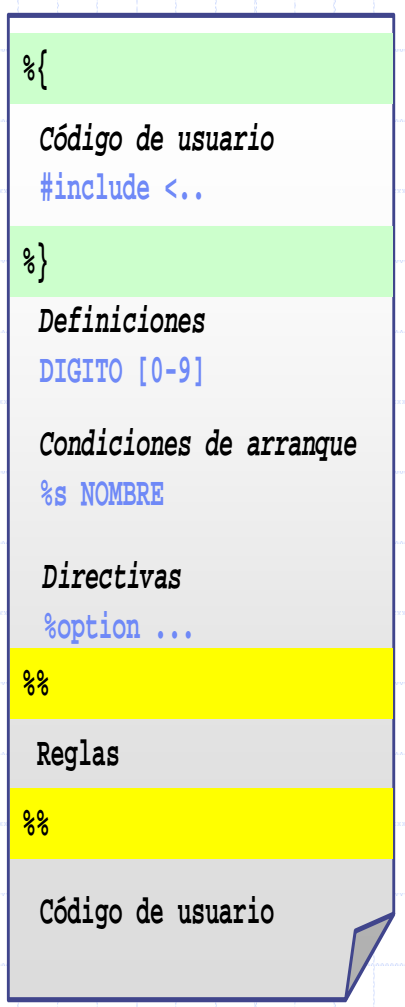
Los tokens que debe reconocer son:

* palabras reservadas (si, entonces, sino, finsi, mientras, etc.)
* constantes
* variables
* operadores matemáticos (+, -, \*, /, ^, etc⁾
* operadores relacionales (>, <, == , != , etc.)
* operadores lógicos (AND, OR, etc.)
* comentarios (!! , !!% %,)

**ESTRUCTURA DEL FICHERO DE ESPECIFICACIÓN LEX/FLEX:**

Se compone de tres secciones separadas por líneas que contienen el separador %%.

****

****

**SECCIÓN DECLARACIONES:**

* Código de usuario: Código C encerrado con los caracteres %{ y %} que se copia literalmente en el fichero de salida lex.yy.c antes de la definición de la función yylex().
  + Incluir ficheros (#include <...>)
  + Declarar variables globales
  + Declarar procedimientos que se describirán en la parte de sección Código de usuario
* Definiciones: Permiten dar nombre a patrones (expresiones regulares) facilitando la legibilidad, estos son utilizados luego en la sección de reglas.
  + entero [0-9]+
  + real {entero}.{entero}
  + numero {entero}|{real}
* Condiciones de arranque:
  + Permiten modificar el flujo de análisis.
  + Hay dos tipos de condiciones de arranque:
    - inclusivas (%s):

Son evaluados los patrones con la condición de arranque y los que no utilizan ninguna **%s variable**.

* + - Exclusivas (%x):

Solo se evalúan los de la condición de arranque **%x comentario**

**SECCIÓN REGLAS:**

* La sección de reglas contiene patrones que describen los tokens y código C.
* Utiliza el formato PATRÓN ACCIÓN
* Los patrones pueden utilizar definiciones, expresiones regulares y condiciones de arranque.
* Las acciones son código C, excepto la activación y desactivación de las condiciones de arranque, y otras (ECHO).
* Reglas para la identificación de patrones
  + Siempre que para la entrada puedan aplicarse varias reglas:
    - Se aplica el patrón que concuerda con el mayor número de caracteres en la entrada.
    - Si hay dos patrones que concuerdan con el mismo número de caracteres en la entrada, entonces se aplica el que esté definido primero.

**SECCION DE CODIGO DE USUARIO**:

* Código C que es copiado al fichero lex.yy.c

**YACC/BISON: generador de analizador sintácticos**

Bison es una herramienta generadora de analizadores sintácticos

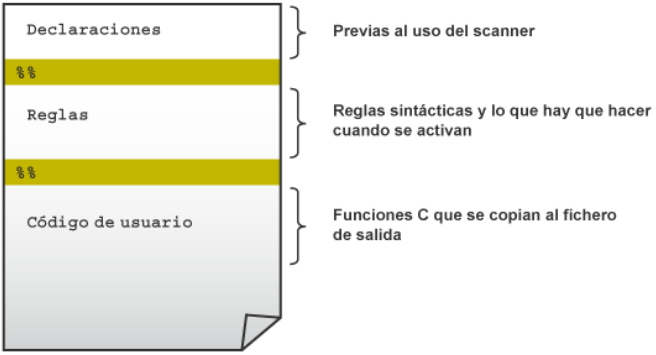
Un analizador sintáctico convierte una descripción gramatical para una gramática independiente del contexto LALR(1) en un programa en C que analizara esa gramática.

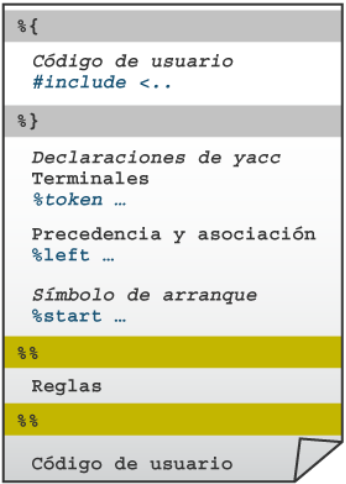
Para que el analizador sintáctico analice un lenguaje, este debe ser descrito por una gramática independiente del contexto. Para esto se debe especificar uno o mas grupos sintácticos y dar reglas para construirlos desde sus partes.

El sistema formal mas común de presentar tales reglas para ser leídas por los humanos es la Forma de Backus-Naur o “BNF".

**ESTRUCTURA DEL FICHERO YACC/BISON:**

Se compone de tres secciones separadas por líneas que contienen el separador %%.



**SECCIÓN DECLARACIONES:**

* Código de usuario: idéntico a la de la sección que tiene Lex/Flex.
* Declaraciones:
  + Terminales y no terminales (opcional):
    - * %token TOKEN
      * %type TOKEN
* Precedencia y asociatividad de operadores:
  + Asociatividad:
    - %noassoc operacion
    - %left operacion
    - %right operación
  + Precedencia:
    - Los declarados en líneas posteriores tienen más precedencia. Por ejemplo:
      * %left ‘+’ ‘-’
      * %left ‘\*’ ‘/’
* Símbolo inicial de la gramática:
  + - * %start símbolo.

**SECCIÓN DE REGLAS:**

Las reglas tiene el formato <producción> acción

* Producción: Son las producciones que pertenecen a una gramática tipo 2 en BNF.
* Acción: Fragmentos de código C que especifican qué hacer cuando se reduce una producción.

Se utiliza un formato BNF simplificado:

LIzq: LDer acción;

* LIzq Es un símbolo no-terminal del lenguaje.
* LDer Secuencia de símbolos no-terminales y terminales.

Ejemplo

expr: expr ´+´ expr {....}

| expr ´-´ expr {.…}

| /\*lambda\*/

;

**SECCIÓN DE COGIDO DE USUARIO:**

* Código C que es copiado al fichero lex.yy.c

**ESTRUCTURA DEL LENGUAJE FUENTE QUE SE DESEA TRADUCIR**

**Inicio** Nomb\_Programa

Sección de declaración de constantes

Sección de declaración de variables

Sección de sentencias

**Fin**

La sección de declaración de constantes: se utiliza para declarar todas las constantes numéricas o de carácter que se van a utilizar durante el programa.

La sección de declaración de variables: se utiliza para declarar todos los identificadores que se van a utilizar durante el programa.

Cada declaración se hace de la siguiente manera:

Variables:

identificador1, identificador2,... **tipo** tipo1

identificador3, identificador4,... **tipo** tipo2

identificador\_vector[]  **tipo** tipo3

1. Los identificadores pueden ser cadenas de caracteres alfabéticos o bien pueden comenzar por una letra, seguida de uno o más letras o dígitos.
2. Los tipos disponibles son:

- Numérico: un número, que puede ser entero o entero con dos decimales

- Cadena: un conjunto de caracteres.

La sección de sentencias permite la escritura de las instrucciones del programa, cada sentencia debe escribirse en una nueva línea y debe finalizar con una coma al final. Las sentencias pueden ser simples o compuestas. A saber:

1. Sentencia simple:
   1. Instrucción de asignación. Se la declara de la siguiente manera:
      1. identificador=expresion\_matematica,
      2. identificador=constante,
      3. identificador=identificador,
      4. identificador = [valor 1 valor 2 valor 3 ...valor n ],

importante: al asignar un arreglo se debe tener en cuenta que todos los valores sean del mismo tipo que el tipo del que fue declarado en la sección de declaración de variables

* + 1. identificador= (expresion\_logica)? expresion\_por\_si:expresion\_por\_no,

1. Instrucción de entrada/salida. Se la declara de la siguiente manera:
   1. Imprimir(cadena) imprime un mensaje por pantalla
   2. bitacora(cadena) imprime un mensaje del estado de la aplicación en el log del sistema o del runtime
   3. leer(variable) pide un valor al usuario y lo guarda en variable
   4. cargar\_vector(archivo, variable) lee desde un archivo con formato valor1, valor2, valor3,.....,valorn y lo almacena en variable como un vector
2. Sentencia compuesta: sentencias condicionales y las repetitivas. A saber:
   1. Condicionales:
      1. Si expresión\_lógica

sentencia1,

sentencia2,...

sino

sentencia1,

sentencia2,...

Finsi

* + 1. evaluar variable

caso valor1:

sentencia,

sentencia 2,...

caso valor2:

sentencia,

sentencia 2,...

otro

sentencia 1,

sentencia 2,...

finevaluar

* 1. Repetitivas:
     1. Mientras expresion\_lógica

sentencia1,

sentencia2,...

Finmientras

* + 1. interar arreglo:interador

sentencia 1,

sentencia 2,...

* + 1. finiterar

Una expresión se define de la siguiente manera:

* identificador operador identificador o
* identificador operador constante o
* constante operador identificador o
* constante operador constante

y puede ser de dos tipos:

- Matemática: cuando se realiza una operación matemática. Puede ejecutarse entre constantes y/o identificadores, siempre y cuando ambos sean numéricos.

Los operadores disponibles para declarar una expresión numérica son **+, -, \* y /.**

- Lógica: cuando se realiza una comparación. Puede ejecutarse entre constantes y/o identificadores de cualquier tipo.

Los operadores disponibles son **<, >, <=, >=, != e ==.**

Las constantes, al igual que los identificadores, pueden ser caracteres, numéricas o cadena.

También se puede incluir comentarios en el código fuente de la siguiente manera

**!! comentario 1**

este es un comentario de una sola línea

**!!%**

**comentario 2**

**%**