Trabajo Práctico Sintaxis y Semántica de los Lenguajes

2do Cuatrimestre 2024

Dr. Oscar Bruno

Indice

En esta sección del trabajo práctico hablaremos sobre como es el paso a paso para la construcción del compilador mediante un análisis sintáctico ascendente. Consideramos prudente separar este proceso en varias etapas:

1. Introducción
2. Configuración del ambiente.
3. Preparación de ficheros de entrada.
4. Relación entre Flex y Bison.

**Introducción**

Antes de empezar, es importante declarar nuestro objetivo dentro de este proyecto: la creación de un programa que pueda leer ciertas entradas y un programa fuente para reportar cualquier posible error sintáctico o léxico que pueda tener durante el procesamiento. Entonces, a partir de esta definición vamos a ir tomando los pequeños pasos para llegar al resultado esperado.

En este proceso donde se detectan posibles errores juegan un rol importante dos programas. El primero de estos dos es el analizador léxico, donde la función de este es asegurarse de que el programa fuente ingresado contenga todos caracteres válidos para un lenguaje regular que expresemos. En nuestro caso, nosotros apuntamos a un lenguaje regular que incluya dígitos, letras en mayúsculas y minúsculas, identificadores, constantes enteras y ciertas palabras reservadas como los operadores aditivos, los operadores multiplicativos, símbolo de asignación, inicio, fin, entre otros. Entonces, para llegar a declarar nuestro lenguaje regular debemos tener el programa de Flex instalado y contar con un fichero de entrada de extensión “.l”. Este fichero será el encargado de definir el lenguaje regular para que el Flex, también llamado Scanner, mediante lexemas para detectar palabras que clasifican como posibles *candidatos para ser un token*. Por otro lado, en el momento que los lexemas se clasifican como tokens, empezaría el trabajo del analizador sintáctico, también llamado Parser, en nuestro caso usaremos el programa “Bison” que se ocupará de trabajar con los tokens dados por el Scanner para determinar si la sintaxis de este programa fuente ingresado es sintácticamente correcto. En este caso, mediante el uso de una BNF, con el formato de una GIC, para definir las expresiones y también se debe representar correctamente la precedencia y asociatividad, ya que, estamos hablando de un lenguaje independiente de contexto, para esto, definimos un fichero de entrada para el Bison con extensión “.y”. Finalmente, se determinan la cantidad de errores sintácticos, de errores léxicos y algunos otros problemas específicos que se cubrirán más adelante, de nuestro programa fuente de extensión “.m”. Este mismo va a contener sentencias que serían las instrucciones completas que, en teoría, debería seguir las reglas sintácticas del lenguaje del fichero del Bison y realiza alguna operación o acción específica. Donde cada sentencia está compuesta de tokens y debe terminar con punto y coma.

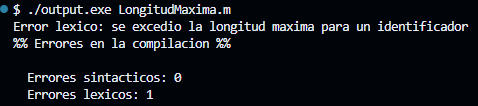
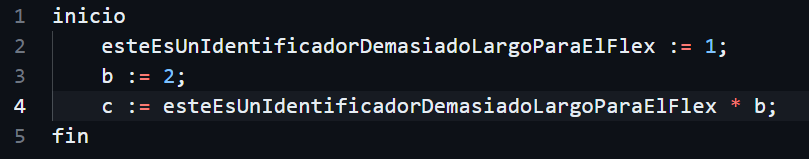
Este proceso es mejor representado gráficamente mediante un árbol que, de un modo jerárquico, representa a la secuencia analizada, donde los tokens que forman la construcción son las hojas del árbol.

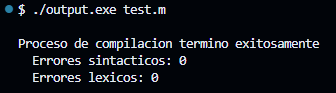
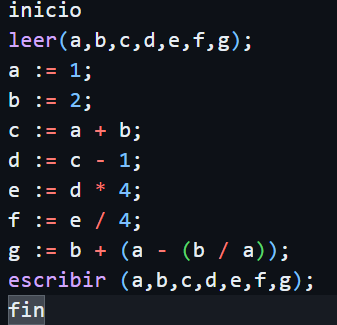
**Configuración del ambiente**

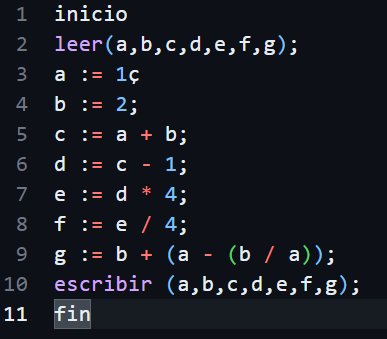
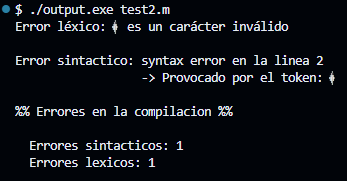
Al tener en cuenta toda la introducción podemos describir como es el flujo para preparar el ambiente para su uso. Entre los pasos encontramos:

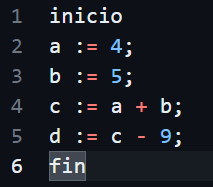
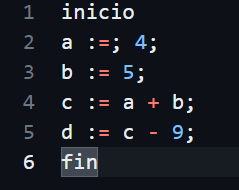
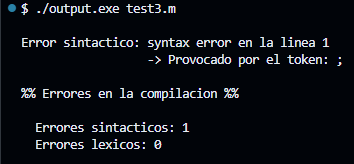
1. Crear una carpeta en donde se encuentren los ficheros de entrada de extensión “.l” y extensión “.y” y un archivo de prueba donde tenemos el programa fuente a analizar, de extensión “.m”.
2. Llamar a una terminal con la dirección en esta misma carpeta e introducir los siguientes comandos:
   1. flex ficheroLéxico.l
   2. bison -yd ficheroSintactico.y
   3. gcc y.tab.c lex.yy.c -o output
3. A partir de esto, podremos usar la misma terminal para ejecutar nuestro programa de compilación con el programa fuente de extensión “.m” de la siguiente manera:
   1. ./output.exe achivoFuente.m

Este es el último paso para el uso del compilador, a continuación, se mostrarán imágenes de como funciona este proceso y las posibles salidas que puede tener.

  
Tenemos un identificador que supera la longitud máxima permitida por el Flex.

  
Un archivo de texto sin error alguno que usa todas las operaciones que le agregamos al compilador.

   
En este caso, se deteca un error léxico por poseer un carácter (ç) que no pertenece al lenguaje y un error sintáctico por la falta del “;”

Apartir del siguiente archivo:  
  
Forzamos unos errores sintácticos:  
  
  
Esto nos da un error sintáctico y nos indica la línea en la que se provocó.

**Preparación de Ficheros de Entrada**

Esta sección es la más importante para definir nuestro lenguaje regular y BNF que van a arbritar los lexemas, tokens y sentencias para determinar si son correctas o si se encuentra algún error. Esto es puede separar en dos partes, el fichero de entrada para el Flex y el fichero de entrada para el Bison.

Fichero de Entrada Léxico

El flex es una herramienta que permite generar scanners, también llamados analizadoes léxicos, donde a partir de ciertas expresiones regulares se busca concordancias en el fichero de entrada que le introduzcamos y ejecuta acciones asociadas a estas expresiones. Este fichero de entrada se encarga de indicarle al Flex el tipo de scanner que debe generar y se divide con la estructura a continuación:

|  |
| --- |
| Definiciones |
| %% |
| Reglas |
| %% |
| Código de usuario |

*Definiciones*: contiene declaraciones de definiciones de nombres sencillos para simplificar la especificación del escáner y desclaraciones de condiciones de arranque.

*Reglas*: contiene serie de reglas de la forma: *patrón {acción*} donde el patrón debe estar sin sangrar y la acción debe comenzar en la misma línea. Su orden es esencial para el funcionamiento correcto (el criterio de términos)

*Código* *de usuario* (opcianal): se copia de ‘ lex.yy.c ‘ de forma literal y se usa para rutinas de complemento que llama al scanner o son llamadas por este.

Los **patrones** usados en la sección de reglas del fichero de entrada del Flex se escriben utilizando un conjunto extendido de expresiones regulares y el alfabeto de cualquier carácter ASCII donde destacamos:

|  |  |
| --- | --- |
| “ . “ | Cualquier carácter excepto nueva línea. |
| [A-Z] | Conjunto de letras en mayúsculas. |
| [^B-E] | Cualquier carácter menos los que están en el conjunto. |
| r | s | ER “r” o bien ER “s”. |
| r {4} | ER “r” exactamente 4 veces. |
| r\* | Cero o más ER “r”. |
| r+ | Una o más ER “r” |
| ^r | ER “r”, pero solo al comienzo de una línea. |

Mientras tanto, las acciones son código en C que están siempre asociadas a un patrón. Se coloca a una linea de su patrón y se coloca entre llaves, pero si no hay nada, o sea, dejamos vacío donde iría una acción, el token se emparejará con el patrón y se descartará (Si se lee comentarios en C, tomará eso).

Por otro lado, al ejecutar los comandos de “flex ficheroLexico.l” en la terminal, se genera una salida llamada ‘lex.yy.c’ que contiene la función de análisis ‘yylex()’, varias tablas usadas para emparejar tokens, rutinas auxiliares y macros. Por defecto, ‘yylex()’ se define:

int yylex(){  
 … acciones y definiciones …  
}

Siempre que se llame a ‘yylex()’, este analiza tokens desde el fichero de entrada global ‘yyin’. La función continúa hasta llegar al final del fichero, desde donde se retorna un valor 0 o una de sus acciones ejecute una sentencia de return.

Además, para considerar en el uso del Flex son las variables disponibles para el usuario:

char \*yytext: apunta al texto del token actual (última palabra reconocida en algún patrón). Ej: printf(“%s”, yytext); -> va a imprimir en pantalla el token actual.

int yyleng: contiene la longitud del token actual.  
  
Antes de finalizar, será de suma importancia en que orden declaramos nuestras reglas, ya que, al recibir un lexema por ‘yytext’ este se va a matchear/coincidir con el primero que corresponda. Por ejemplo, si colocamos la regla del identificador antes de la regla de constante Entera, entonces habrá lexemas como “123” que se reconocerán como identificadores.

Fichero de entrada Sintáctico

Bison es un generador de analizadores sintácticas de propósito general que convierte una descipción para una gramática independiente de contexto en un programa en C que analiza esa gramática, que se saca del fichero de entrada de extensión ‘.y’ y este archivo se separa así:

|  |
| --- |
| %{  Declaraciones en C  %} |
| Declaraciones de Bison |
| %% |
| Reglas gramaticales |
| %% |
| Código C adicional |

*Declaraciones en C*: se pueden definir tipos y varibles utilziados en las acciones. También, se usan comandos del preprocesador para definir macros que se utilicen ahí, y el escrbir #include para incluir archivos de cabeceras que realicen cualquier cosa.

*Declaración de Bison*: declaran los nombres de los símbolos terminales y no terminales. Además, se describen precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

*Reglas gramáticales*: son las producciones de la gramática, que además puede llevar asociada acciones o código en C, que se ejecuten cuando el analizador encuentra las reglas correspondientes.

*Código C adicional*: contiene cualquier código en C. Suele ir la definición del analizador léxcio ‘yylex()’ y más subrutinas invocadas por las acciones en las reglas   
gramaticales.

Símbolos, terminales y no terminales

Los símbolos terminales de la gramática en Bison se llaman Tokens y se deben declarar en la sección de deficiones. Un token declarado se describe con un identificador, por convención, en mayúsculas. Mientras que, un token de carácter, se escribe en la gramática usando la misma sintaxis que en C y se usa para representar un token consistente en ese carácter en particular.

Sintaxis de las reglas gramáticales

Una regla gramatical de Bison tiene la forma:

IDENTIFICADOR: componentes  
 ;

Los espacios en blanco son únicamente significantes para la separación de símbolos, además, es posible distribuir acciones en el medio de las componentes que determinan la semántica de la regla, con este formato: {sentencias en C}.

Es posible definir se hay recursión a derecha o a izquierda de la siguiente forma:

Recursión a derecha -> expseq1: exp   
 | exp ‘,’ expseq1  
 ;

Recursión a izquierda -> expseq1: exp   
 | expseq1 ‘,’ exp  
 ;  
Se prefiere el uso de la recursión por la izquierda, ya que, no ocupa espacio de pila al analizar una secuencia de elementos. Sin embargo, en nuestra solución se decide sobre el uso de la recursión a derecha.

Semántica del Lenguaje

La semántica viene determinada por los valores semánticos asociados con varios tokens y agrupaciones, y por las cciones tomadas cuando varias agrupaciones son reconocidas. En la mayoría de programas se necesitará diferentes tipos de datos para diferentes clases de tokens y agrupaciones: para declarar más de un tipo de dato para los valores semánticos se

1. Especifica la colección completa de tipos de datos posibles, con la declaración de Bison: % union{}
2. Se elige uno de estos tipos para cada símbolo terminal o no terminal. Para los terminales se usa %token y para los no terminales se usa %type

Declaraciones en Bison

En esta sección de declaraciones en Bison se define los símbolos utilizados en la formulación de la gramática y los tipos de datos de los valores semánticos.

* Todos los nombre de token se deben declarar (pero no los tokens de char literal simple como ‘+’ y ‘\*’
* Los símbolos no terminales deben ser declarados si se necesita especificar el tipo de dato a utilizar para los valores semánticos.
* La primera relga en el fichero también especifica el símbolo inicial (por defecto)
  + Si se desea explicitar otro símbolo inicial se debe especificar.

Nombres de Token

La forma básica de declarar un nombre de token (símbolo terminal) es: *%token nombre1 nombre2 nombre3 … nombreN*.   
La sentencia *% union{}* especifica varios tipos de valores con nombres y se debe declarar el tipo de valor de cada símbolo no terminal.  
La sentencia *%token <val> NUM* define el token NUM y su tipo.

**Relación entre Flex y Bison**

El uso de Flex y su interacción con Bison

El uso principal del Flex es ser el acompañante de Bison, ya que, este necesita la función ‘yylex()’ que funciona de la siguiente manera:

1. Reconoce lexemas como tokens desde el flujo de entrada y se los devuelve al analizador.
2. Puede poner cualquier valor asociado en la variable global ‘yylval’.

La conexión etnre el Flex con Bison se genera cuando en la línea de comandos introducimos la opción “-d” que crea el archivo ‘y.tab.h’ que contiene las definiciones de todos los ‘%tokens’ que aparecen en el fuente Bison que se inclute después de la fuente de Flex.

El archivo ‘y.tab.c’ se usa para poder compilar con gcc.

Funcionamiento del Analizador (yyparse)

La fuente de Bison se convierte en una función en C llamada ‘yyparse’, esta función:

1. Lee tokens dados por el scanner (yylex()).
2. Ejecuta acciones.
3. Retorna cuando llega al final del fichero o un error de sintaxis del que no puede recuperarse. (es pobile escribir acciones que ordenen a yyparse retonar sin leer hasta el final).