PFC-CIS-IXB-024

UNIVERSIDAD NACIONAL DE LOJA

Área de la Energía las Industrias y los Recursos Naturales No Renovables

---

Carrera de Ingeniería en Sistemas

Compilador para realizar la suma y resta de dos números enteros de dos cifras.

Trabajo Final: Compiladores

Noveno B

Autor:

- Henry Paúl Vivanco Encalada

Docente: Ing. Henry-Paz,

Loja-Ecuador
2014

Índice

Índice de figuras

A. DESCRIPCIÓN

Como se planteó en el tema el compilador realiza la suma o resta de dos números enteros de dos cifras, si se desea sumar o restar un número de una cifra se debe anteponer el cero a su izquierda, para poder realizar esta suma o resta se debe colocar los sumandos cambiando el orden de los mismos.

Es decir si se desea suma el número “15” más el número “15”; la sentencia a colocar seria “51” mas “51”, dándonos como respuesta la suma de los numero en el orden invertido es decir en este ejemplo nos daría como resultado “30”, de la misma manera para la realización de restas, siempre anteponiendo la palabra reservada “Imp”, para poder así mostrar en pantalla la suma o resta de estos valores.

Otra restricción es que los operadores deben ir escritos en letras mayúsculas es decir el signo “+” en el compilador seria “MAS”, así mismo con el operador “-” en el compilador se debería plantearlo de esta manera “MENOS”.

Algo adicional en el compilador es que para que haga valida la sentencia a resolver se debe agregar un fin de línea que en este caso va descrito por la letra “h”.

B. EJEMPLO

Entrada:

Imp 51 MAS 51 h
Imp 30 MENOS 20 h

Consola:

51 MAS 51 IGUAL 30
30 MENOS 20 IGUAL 1

1 . Pablabras Reservadas

h: Para finalizar cada línea de código.
Imp: Imprimir en consola.

2 . Operadores

Los operadores serán escritos por su nombre y con letra mayúscula.

- IGUAL
- MENOS

C. AUTÓMATA DEL COMPILADOR PLANTEADO

Para realizar el autómata del compilador planteado necesitamos fijar las reglas que va a tener nuestro compilador, por ejemplo se dice que FINLINEA sera con la letra "h"minuscula, asi mismo los datos enteros aceptaran unicamente 2 numeros para cada cantidad, por ello el valor ENTERO se declaro de la siguiente manera:

Mas adelante se indicara de una manera más explicativa lo referente a las declaración y reglas que se deben plantear en el analizador léxico.

FINLINEA: h
SUMA: MAS
RESTA: MENOS
IMPRIMIR: Imp
ENTERO: Entero = [0-9][0-9]

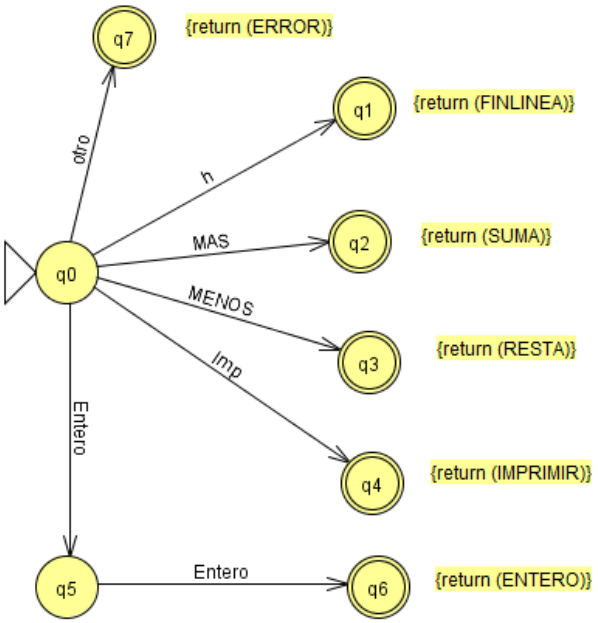

Figura 1: Autómata

D. ANALIZADOR LEXICO (.flex)

El analizador léxico es la parte principal del compilador ya que es aquí en donde declaremos nuestras definiciones y nuestras reglas léxicas, es decir reglas de cómo se nombraran

los diferentes componentes del compilador.

Un archivo .flex se encuentra estructurado por tres partes: Codigo de usuario, Directivas y las reglas para las expresiones regulares; todas las partes separadas por doble porcentaje.[1]

A continuación se mostrara de una manera más explicativa la estructura del archivo y como está definido para el compilador planteado.

1 . Codigo de Usuario

```
package ejemplocup;

import java_cup.runtime.*;
import java.io.Reader;
```

Figura 2: Codigo de Usuario

En la Figura 2 se puede apreciar en la primera línea el nombre del paquete en donde se encuentra nuestro compilador, seguido de las importaciones a usarse.

2 . Directivas

```
%%

%class AnalizadorLexico


%line
%column
%cup

%{

    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }

    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}


Salto = \r|\n|\r\n

Espacio     = {Salto} | [ \t\f]

Entero = [0-9][0-9]

%%
```

Figura 3: Directivas

Como se indicó con anterioridad cada sección va dentro de doble porcentaje como se aprecia en la Figura 3, primero se debe indicar el nombre de la clase de nuestro analizador léxico, seguido se hace el llamado a line y column para en su posterioridad ser usados para el conteo de donde se halla un error, se hace la llamada de cup para la sincronización o comunicación con el analizador sintáctico; además se realiza dos métodos de la clase Symbol, estos métodos nos sirven para la identificación de los errores, identificando la línea y

la columna en donde se halla el mismo.

A continuación se realizan las declaraciones a ser usadas en las reglas léxicas; se ha declarado una variable salto en donde se indica el retorno de carro y el salto de línea.

Se declara una variable Espacio en donde se indica que contiene la variable antes declarada salto o un espacio y tabulación.

Por último se declara una variable más en donde se indica cuáles serán los números enteros para el compilador. Como mi compilador solamente acepta valores de dos dígitos se realizó de la manera que se indica en la Figura 3.

3 . Reglas para las Expresiones Regulares

```
%%

<YYINITIAL> {

    "h"                 { return symbol(sym.FINLINEA); }

    "MAS"               {  System.out.print(" MAS ");
                           return symbol(sym.SUMA); }

    "MENOS"             {  System.out.print(" MENOS ");
                           return symbol(sym.RESTA); }

    "Imp"               { return symbol(sym.IMPRIMIR); }


    {Entero}    {   System.out.print(yytext());
                    return symbol(sym.ENTERO, new Integer(yytext())); }


    {Espacio}       { /* ignora el espacio */ }

}

[^] { System.err.println("Caracter Ilegal: " + "<<"+ yytext() + ">>" + "[" + yyline + ":" + yycolumn + "]"); }
```

Figura 4: Reglas para las Expresiones Regulares

YYINITIAL es un estado inicial el cual va a ir almacenando nuestros tokens que serán declarados.

Las reglas se las declara de la siguiente manera:

En el compilador la letra “h” será el fin de línea para dar esto se lo declara de la forma que se presenta en la Figura 4, dando un retorno de un symbol y declarando que el token es FINLINEA.

De la misma manera se lo realiza para los operadores y para la palabra reservada “Imp”. En los operadores se agrega un System.out.print ya que se desea que se muestre la palabra “MAS” o “MENOS”, según sea el caso, en consola.

Para declarar los valores enteros se acude a la variable antes creada con el nombre Entero en esta regla se agrega, después de declarar su token, new Integer ya que deseamos

trabajar con enteros.

Se toma en cuenta la variable creada Espacio para ingnorar si se da estos casos.

Por último se realiza un System.err.println para la mostrar los errores indicando el texto del mismo y su ubicación.

E. ANALIZADOR SINTACTICO (.cup)

El analizador sintáctico se estructura por cinco partes: Import java, código del usuario para el parser, código del usuario para las acciones de la gramática, declaración de variables para la gramática y gramatica.[2]

La estructura del analizador sintactico del compilador planteado se muestra a continuacion:

1 . Imports Java

```
package ejemplocup;

import java_cup.runtime.*;
import java.io.FileReader;
```

Figura 5: Imports Java

De la misma manera que en el analizador léxico se antepone el paquete en donde se halla nuestro compilador, seguido por las importaciones a usar.

2 . Código del Usuario para el Parser

```
parser code {:

    public void report_error(String message, Object info) {
        StringBuilder m = new StringBuilder("Error");
        if (info instanceof java_cup.runtime.Symbol) {
            java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
            if (s.left >= 0) {
                m.append(" in line "+(s.left+1));
                if (s.right >= 0)
                    m.append(", column "+(s.right+1));
            }
        }
        m.append(" : "+message);
        System.err.println(m);
    }

    public void report_fatal_error(String message, Object info) {
        report_error(message, info);
        System.exit(1);
    }

    public static void main(String[] args){
        try {
            AnalizadorSintactico asin = new AnalizadorSintactico(
                    new AnalizadorLexico( new FileReader(args[0])));
            Object result = asin.parse().value;
            System.out.println("\n*** Resultados ***");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

:};
```

Figura 6: Código del Usuario para el Parser

En la Figura 6 se muestra codigo java en el cual existen tres métodos:

- Especifica un reporte de error indicando la línea y columna en donde se halla el mismo.
- Otra manera de indicar un error en la parte gramatical el cual será usado para identificar el tipo de error que se halle.
- Es el método main el ejecutable del compilador.

3 . Código del Usuario para las Acciones de la Gramática

En el ejemplo planteado no se hace uso de esta parte de la estructura, ya que las acciones que realizara el compilador fueron implementadas directamente en la gramatica del analizador sintáctico. No generadas en un método para luego ser llamadas.

4 . Declaración de Variables para la Gramática

Las variables declaradas pueden ser de dos tipos: terminales o no terminales.

```
terminal          FINLINEA, SUMA, RESTA, IMPRIMIR;
terminal Integer  ENTERO;


non terminal Object lista, sentencia;
non terminal Integer exprecion, termino;
```

Figura 7: Declaración de Variables para la Gramática

En la parte de las variables terminales se declaró los tokens declarados en el analizador léxico.

Se ha declarado un terminal Integer Entero; se lo declaro de esta manera ya que para cumplir con el propósito del compilador se debe trabajar con valores enteros.

Las variables no terminales son aquellas que haremos uso para la declaración de las reglas gramaticales.

5 . Gramática

La parte de las reglas gramaticales se la han especificado de la siguiente manera:

```
lista ::= lista sentencia
          |
          sentencia
          ;
```

Figura 8: Lista de Sentencias

En la Figura 8 se muestra la declaración de una lista de sentencias o una solo sentencia a analizarse.

```
sentencia ::= error {:
                parser.report_fatal_error("Palabra reservada <<Imp>>", 5 );
                :}
              |IMPRIMIR error {:
                parser.report_fatal_error("La Operacion debe realizarse con dos numero de 2
                digitos cada uno y el fin de linea debe ser <<h>>", 6 );
                :}
              |IMPRIMIR ENTERO  error {:
                parser.report_fatal_error("Falta Operador <<SUMA>> o <<RESTA>>", 4 );
                :}
              |IMPRIMIR exprecion:e
              {:
                System.out.println(" IGUAL " + e);
              :}
              FINLINEA
            ;
```

Figura 9: Sentencia

La sentencia viene dada primeramente identificando los errores gramaticales que puedan existir. Más adelante se indicara con más claridad cómo se realiza la captura de estos errores.

Después de identificar los errores se realiza la sentencia de la expresión y por último se incluye el FINLINEA mencionado con anterioridad. Como codigo java se realiza un System.out.println para indicar el resultado que se desea.

```
exprecion      ::= exprecion:e SUMA termino:f
                   {:
                     e = Integer.parseInt(new StringBuffer(String.valueOf(e)).reverse().toString());
                     f = Integer.parseInt(new StringBuffer(String.valueOf(f)).reverse().toString());
                     RESULT = new Integer(e.intValue() + f.intValue());
                   :}
                   |
                   exprecion:e RESTA termino:f
                   {:
                     e = Integer.parseInt(new StringBuffer(String.valueOf(e)).reverse().toString());
                     f = Integer.parseInt(new StringBuffer(String.valueOf(f)).reverse().toString());
                     RESULT = new Integer(e.intValue() - f.intValue());
                   :}
                   |
                   termino:n
                   {:
                     RESULT = n;
                   :}
                   ;
```

Figura 10: Expresión

Como se indica en la Figura 10 la expresión viene dada por un valor entero, a continuación se agrega el operador deseada, sumando o restando a un término dado; en codigo java se realizó lo siguiente:

- Se cambia el orden de los términos a ser sumados o restados.
- Se realiza la suma o resta de los términos y se los almacena en una variable RESULT la cual contiene el valor o respuesta deseada. Se realiza de la misma manera para cualquiera de las dos operaciones, cambiando el signo de la operación.

Así mismo se dice que la expresión puede venir dada solamente por el término.

```
termino        ::= ENTERO:e
                   {:
                     RESULT = e;
                   :}
                   ;
```

Figura 11: Termino

En la Figura 11 se muestra que el termino viene dado por el token ENTERO y en codigo java se almacena este valor en una varible RESULT.

F. ESTRUCTURA EN NetBeans

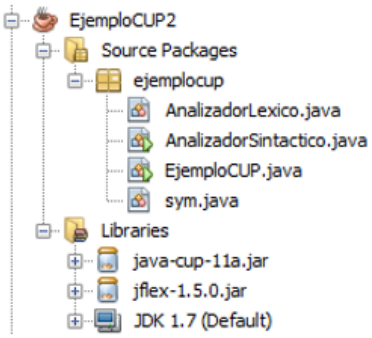

Figura 12: Estructura en NetBeans

Para la ejecución de compilador se debe importar las librerías que se indican en la Figura 12. Se debe crear la clase que contiene el método main, el cual se ejecuta y automáticamente crea las clases: AnalisadorLexico, AnalisadorSintactico y la clase sym.

G. RESULTADOS

Al momento de ejecutar la clase principal del compilador mostrara en consola lo siguiente:

```
run:
Elija una opcion:
1) Generar
2) Ejecutar
3) Salir
Opcion:
```

Figura 13: Menu de Ejecución

Al ingresar el número 1 se generan las clases antes mencionadas, y si ya existiensen remplazara sus datos.

Al insertar el número 2 en consola como se muestra en la Figura 14 se ejecutara el compilador mostrando los resultados.

Si se ingresa el número 3 en consola se dejara de ejecutar el compilador.

```
Elija una opcion:
1) Generar
2) Ejecutar
3) Salir
Opcion: 2
51 MAS 51 IGUAL 30
30 MENOS 20 IGUAL 1
```

Figura 14: Resultados

En la Figura 14 se puede apreciar los resultados obtenidos después de haber ejecutado el compilador. Para ello se debe escribir la o las sentencias en un archivo de entrada como se muestra en la Figura 15.

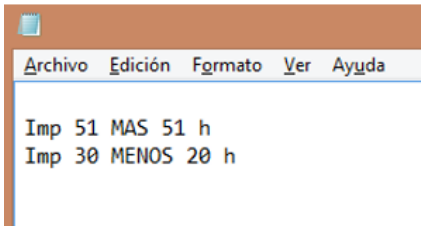

Figura 15: Datos de Entrada

En la Figura 15 indica dos sentencias ingresadas por el usuario en la primera de ellas realiza la suma de los dos dígitos; restringiéndose a las condiciones mencionadas con anterioridad, así mismo realiza la resta de la misma manera.

1 . Errores

Los errores son un plus que se ha dado al compilador; errores indicando cual es y que se debe hacer el caso de darse uno de ellos al plantear la o las sentencias a ser leídas por el compilador. En la figura 9 se indicó como se realizó la captura de los errores que se pueden dar al leer la entrada del compilador. En la figura 16 se muestra las sentencias que son usadas para ser capturados estos errores.

```
sentencia ::= error {:
                parser.report_fatal_error("Palabra reservada <<Imp>>", 5 );
                :}
              |IMPRIMIR error {:
                parser.report_fatal_error("La Operacion debe realizarse con dos numero de 2
                digitos cada uno y el fin de linea debe ser <<h>>", 6 );
                :}
              |IMPRIMIR ENTERO  error {:
                parser.report_fatal_error("Falta Operador <<SUMA>> o <<RESTA>>", 4 );
                :}
              |IMPRIMIR exprecion:e
              {:
                System.out.println(" IGUAL " + e);
              :}
              FINLINEA
             ;
```

Figura 16: Captura de Errores

Como se indica en la Figura 16 se declara como va a ir estructurada la sentencia y la

parte de la misma que falta.

En el primer error encontrado como se indica en la Figura 16 es porque la sentencia a ser leída por el compilador no inicia con la palabra reservada “Imp”, para ello en consola mostrara un mensaje de cuál es el error que se presenta.

En la segunda captura del error se indica que la sentencia inicia con la palabra reservada “Imp” para ello se ingresa el token IMPRIMIR acompañado del token ENTERO que representa que: una vez ingresada la palabra reservada “Imp” debe ser acompañada por un numero entero.

De la misma manera se presenta mensaje de error si hace falta la colocación de los operadores dentro de la sentencia a ser leída.

Para lograr estos objetivos se debe hacer el llamado al método parserreportfatalerror, dentro de codigo java, el método viene dado por dos parámetros de entrada:

Mensaje: Es de tipo String, en este parametro escribimos el mensaje que se decía que el usuario visualize al momento de encontrar el error en la sentencia.

Info: Es de tipo Object, en este parámetro lo que hacemos es dirigirnos a la clase sym generada con anterioridad en ella encontramos el nombre del token y la llave del mismo, la llave del token será ingresada en este parámetro de acuerdo al error existente.

A continuación se indica los errores que pueden darse al leer la o las sentencias de entrada.

Si el archivo de entrada se halla vacío o falta la palabra reservada “Imp”, al momento de ejecutar el archivo en consola mostrara lo siguiente:

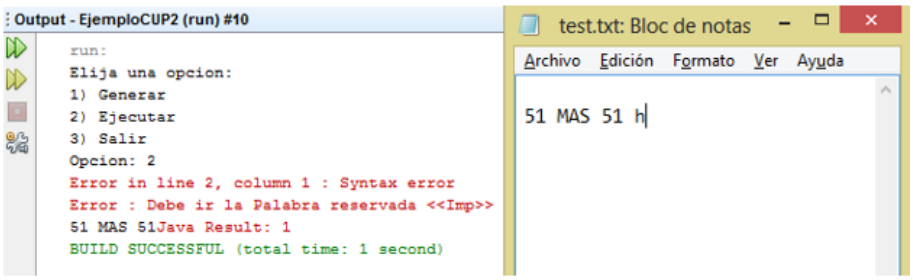

Figura 17: Error Imp

Si se trata de un error de falta del operador se mostrara en consola lo siguiente:

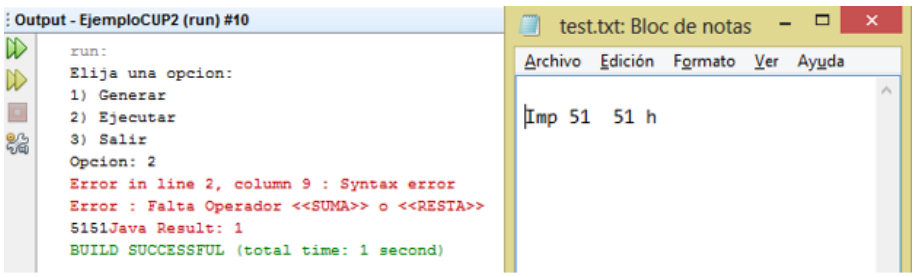

Figura 18: Error Operador

En la Figura 19 y 20 se muestra un error al ingresar solamente la palabra reservada "Imp", el mismo mensaje mostrara si para terminar la sentencia no se ajusta a lo especificado en el archivo léxico. Asi mismo mostrara este mensaje cuando se el número de dígitos a sumarse o restarse no sea el especificado.

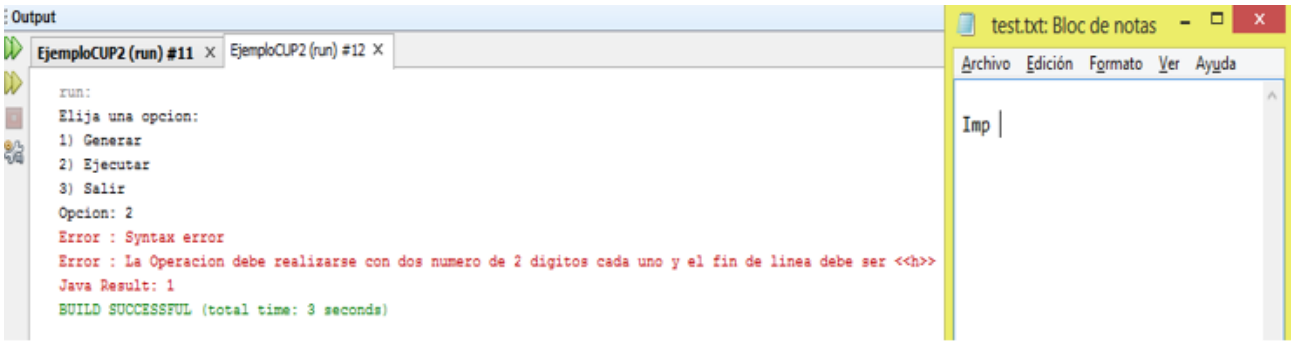

Figura 19: Error dígitos y fin linea 1

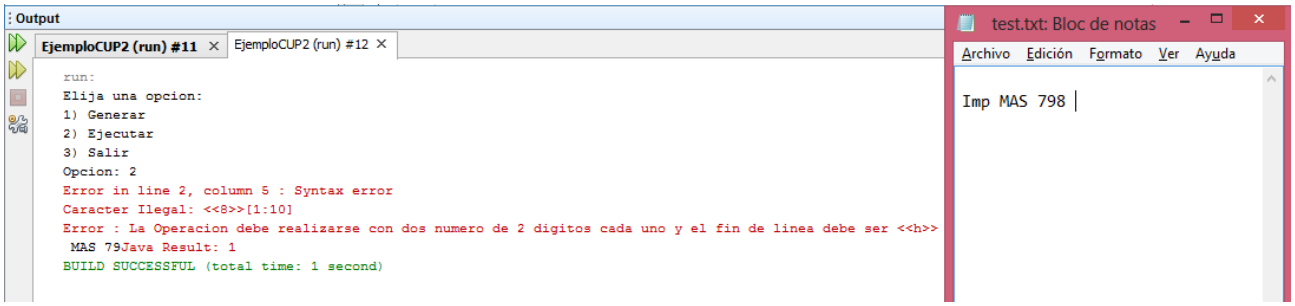

Figura 20: Error dígitos y fin linea 2

El codigo del compilador, con el archivo lexico y sintactico se encuentran disponible en: `https://github.com/henryVivanco/Compiladores/tree/master/CompiladorSumaYResta/EjemploCUP`

H. BIBLIOGRAFIA

Referencias

[1] Analizador léxico, sintáctico y semántico con JFlex y CUP. Crisol. (2007). Visto [10 febrero de 2015]. Disponible en: `http://crysol.org/es/node/819`

[2] Robson cruz. Sintaxis Cup.Themfgeek. Visto [10 febrero de 2015]. Disponible en: `http://themfgeek.blogspot.com/2012/10/sintaxis-cup.html`