

Trabajo Práctico N°1 y N°2
Diseño de Compiladores I
Grupo 4



Integrantes:
Freire, Agustín
Loguercio, Giorgio
Sottile, Gianni

agustinfreire2002@outlook.com.ar
loguerciogiorgioivan@gmail.com
giannisottile7@gmail.com

Facultad de Cs. Exactas
Universidad Nacional del Centro de la Provincia de Buenos Aires

Introducción

Durante el presente trabajo se desarrollará las primeras etapas de un compilador siguiendo las instrucciones de la cátedra, contando con las etapas del análisis léxico y análisis sintáctico. Se desarrollará en JAVA utilizando la herramienta de yacc.

El objetivo de estas etapas será analizar un código de programa ingresado por el usuario, analizándolo tanto léxica como sintácticamente, detectando y notificando la mayor cantidad posible de errores, en caso de que los haya, en cada etapa.

Debido a que tanto la parte Léxica como Sintáctica están muy conectadas entre sí y a su vez se entregan ambas juntas, decidimos no separar el informe en las etapas de análisis léxico y sintáctico, sino que explicamos las consignas del informe de manera conjunta.

Consignas Generales TP 1

Se nos pidió desarrollar un Analizador Léxico que reconozca los siguientes tokens:

- Identificadores cuyos nombres pueden tener hasta 15 caracteres de longitud. El primer carácter sólo puede ser una letra, y el resto pueden ser letras, dígitos y “_”. Los identificadores con longitud mayor serán truncados y esto se informará como Warning. Las letras utilizadas en los nombres de identificadores pueden ser minúsculas o mayúsculas.
- Constantes correspondientes al tema particular asignado a cada grupo.
- Operadores aritméticos: “+”, “-”, “*”, “/”.
- Operador de asignación: “:=”
- Comparadores: “>=”, “<=”, “>”, “<”, “=”, “!=”
- “(”, “)”, “;”, “:”, “.” y “,”
- Cadenas de caracteres correspondientes al tema particular de cada grupo.
- Palabras reservadas (que pueden escribirse con mayúsculas o minúsculas): IF, THEN, ELSE, BEGIN, END, END_IF, OUTF, TYPEDEF, FUN, RET y demás símbolos / tokens indicados en los temas particulares asignados a cada grupo.
- El Analizador Léxico debe eliminar de la entrada (reconocer, pero no informar como tokens al Analizador Sintáctico), los siguientes elementos.
- Comentarios correspondientes al tema particular de cada grupo.
- Caracteres en blanco, tabulaciones y saltos de línea, que pueden aparecer en cualquier lugar de una sentencia.

Consignas Generales TP 2

Se deberá construir un Parser (Analizador Sintáctico) que invoque al Analizador Léxico creado en el Trabajo Práctico N° 1, y que reconozca un lenguaje con las siguientes características:

SINTAXIS GENERAL

Programa:

- Constituido por un conjunto de sentencias, que pueden ser declarativas o ejecutables.
- Las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control.
- Los elementos declarados sólo serán visibles a partir de su declaración (esto será chequeado en etapas posteriores).
- Cada sentencia debe terminar con punto y coma ";".
- El programa comenzará con un nombre, seguido por un conjunto de sentencias delimitado por BEGIN y END.

Sentencias declarativas:

Declaración variables

Sentencias de declaración de datos para los tipos de datos correspondientes a cada grupo según la consigna del Trabajo Práctico 1, con la siguiente sintaxis: <tipo> <lista_de_variables>;

Donde <tipo> puede ser (Según tipos correspondientes a cada grupo): integer, uinteger, longint, ulongint, single, double.

Las variables de la lista se separan con coma (",")

Declaración funciones

Incluir declaración de funciones, con la siguiente sintaxis:

```
<tipo> FUN ID (<parametro>) begin  
<cuerpo_de_la_funcion>  
end
```

Donde:

<parametro> será un identificador precedido por un tipo: <tipo> ID

La presencia del parámetro es obligatoria. Este chequeo debe efectuarse durante el Análisis Sintáctico

<cuerpo_de_la_funcion> es un conjunto de sentencias declarativas (incluyendo declaración de otras funciones) y/o ejecutables, incluyendo sentencias de retorno con la siguiente estructura: RET (<expresión>;

Sentencias ejecutables

Asignacion de variables

Asignaciones donde el lado izquierdo puede ser un identificador, y el lado derecho una expresión aritmética.

Los operandos de las expresiones aritméticas pueden ser variables, constantes, invocaciones a función u otras expresiones aritméticas.

No se deben permitir anidamientos de expresiones con paréntesis.

Las invocaciones a función tendrán el siguiente formato: ID(<parametro_real>)

El parámetro real puede ser cualquier expresión aritmética, variable o constante.

Cláusula de selección IF/ELSE

- Cada rama de la selección será un bloque de sentencias.
- La estructura de la selección será, entonces:
IF (<condicion>) THEN <bloque_de_sent_ejecutables> ELSE <bloque_de_sent_ejecutables> END_IF ;
- El bloque para el ELSE puede estar ausente.
- La condición será una comparación entre expresiones aritméticas, variables, invocaciones a función o constantes, y debe escribirse entre “(“ ”)”.
“ (“ ”)”
- El bloque de sentencias ejecutables puede estar constituido por una sola sentencia, o un conjunto de sentencias ejecutables delimitadas por begin end.

Salida de mensajes

Sentencia de salida de mensajes por pantalla.

El formato será OUTF(<cadena>); o OUTF(<expresion>);

Temas particulares asignados

4. Enteros largos sin signo (32 bits): Constantes enteras con valores entre **0** y **$2^{32} - 1$** .

Se debe incorporar a la lista de palabras reservadas la palabra **ulongint**.

6. Punto flotante de 64 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra “d” (minúscula) y el signo del exponente es opcional. Su ausencia implica signo positivo para el exponente. La parte entera, la parte decimal, y el ‘.’ son obligatorios.

Considerar el rango:

- $2.2250738585072014d-308 < x < 1.7976931348623157d+308$ U
- $-1.7976931348623157d+308 < x < -2.2250738585072014d-308$ U 0.0

Se debe incorporar a la lista de palabras reservadas la palabra **double**.

7. Constantes octales, que se escribirán comenzando con un 0, y serán una secuencia de dígitos que pueden ir de 0 a 7. El rango para las constantes octales será el mismo que el del tipo entero asignado al grupo:

d. Para ulongint: 00 a 037777777777

10. Cadenas multilínea: Cadenas de caracteres delimitadas por corchetes. Estas cadenas pueden ocupar más de una línea. (En la Tabla de símbolos se guardará la cadena sin los saltos de línea).

Ejemplo:

```
[ ¡Hola  
mundo! ]
```

12. La primera letra de un identificador podrá definir el tipo de una variable:

d. Para ulongint, los identificadores que comienzan con x, y o z serán de tipo entero largo sin signo

f. Para double, los identificadores que comiencen con d serán de tipo double.

16. Incorporar a la lista de palabras reservadas, las palabras **FOR**, **UP** y **DOWN**

FOR (i := m ; <condición> ; up/down n) < bloque_de_sentencias_ejecutables > ;

- i debe ser una variable entera.
- m y n serán constantes enteras
- <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por begin end.

Nota: Las restricciones de tipo de la variable de control y los componentes del rango serán chequeadas en la etapa 3 del trabajo práctico.

19: Pattern Matching Se debe incorporar a las condiciones, la posibilidad de comparar listas de expresiones entre paréntesis. Las expresiones se separarán con coma (“,”).

22. Incorporar a la lista de palabras reservadas la palabra **TRIPLE**.

Incorporar la posibilidad de declarar tipos triple de un tipo determinado (considerando los tipos asignados al grupo).

Se podrán declarar variables del tipo definido (en el ejemplo t₃ing), y los componentes de cada variable se referenciarán con el nombre de la variable y la posición de la componente entre corchetes, considerando 1, 2 y 3 para la primera, segunda y tercera componente, respectivamente.

23. Incorporar a la lista de palabras reservadas la palabra **GOTO**.

Permitir, como sentencia ejecutable, la sentencia goto <etiqueta>, donde la etiqueta será un identificador seguido de dos puntos (“:”).

Esta etiqueta podrá aparecer en cualquier lugar del código, como una sentencia más

27. Sin conversiones en expresiones / Conversiones explícitas de parámetros

Se debe incorporar a la sintaxis de la invocación a una función, la posibilidad de anteponer un tipo al nombre del parámetro real, con el fin de convertirlo al tipo del parámetro formal. Todos los chequeos asociados con compatibilidad de tipos se explicarán y resolverán en trabajos prácticos 3 y 4.

28. Comentarios de 1 línea: Comentarios que comiencen con “##” y terminen con el fin de línea.

Decisiones de diseño e implementación

Ingreso del nombre del archivo a compilar

Primero, se decidió manejar el ingreso del archivo poniendo el nombre del archivo mediante la consola, razón por la cual el archivo debe estar en la carpeta de "...\\codigos". Si el archivo no es encontrado se indicará un mensaje de error.

Manejo de archivo a compilar en java

El archivo en Java se decidió manejarlo como un ArrayList de Strings donde cada elemento tiene una línea.

Para tratar los saltos de línea, ya que cada línea es un elemento de la lista, los detectamos al final del String de dicha línea, es decir, cuando la posición que se va recorriendo dentro del String llega a su fin.

Manejo de octales

Al tener que permitir la representación en octal de los ulongint, no dejaremos que se declaren ulongint con un 0 de prefijo, es decir, no admitiremos por ejemplo 008, 05, 00302;

008 no se tomará como entero ni como octal, 05 será un octal y 00302 también.

Aun así, se permite por ejemplo un 0.23 como ulongint

Notificación de errores léxicos y sintácticos

Se definió una clase estática ErrorHandler el cual se encarga de añadir los errores, léxicos y sintácticos, y contar la cantidad de los mismos.

Al encontrar un error léxico o sintáctico se añade un texto, indicando el error y la línea donde ocurre, a una cadena de caracteres y se suma 1 a la cantidad de errores encontrados. Se decidió utilizar la misma estructura para guardar ambos tipos de errores para así mantener el orden de aparición de los errores.

La cantidad de errores se utilizará de manera futura para saber si el código debe generarse o no.

Matrices

Para representar el grafo de transición de estados y sus acciones se utilizaron 2 matrices.

Columna de la matriz a acceder

Para saber a qué columna hay que acceder de la matriz se utilizó un case que dado el último carácter leído devuelve qué columna de la matriz representa.

Matriz de transición de estados

Es una matriz de Integers donde cada línea es un estado, cada columna se accede según el último carácter leído y esa posición contiene el estado a transicionar.

Matriz de acciones semánticas

Se creó una clase AccionSemantica(explicada más adelante en detalle), con la cual se definió una matriz con ese tipo, donde cada elemento de la matriz es un objeto de tipo AccionSemantica que ejecuta un metodo “ejecutar” que es la acción tomar dado el estado y el símbolo leído.

Lexemas y palabras reservadas

Se utilizó un set de tipo <String> para guardar cada palabra reservada. Simplemente se utiliza para saber si un lexema ingresado es una palabra reservada, con un contains.

Para conseguir el número de token es mediante el que le asigna el Parser a los tokens declarados en el mismo, comenzando a partir del último ASCII (256). Para los lexemas que tienen una única representación se devuelve dicho ASCII directamente.

Tabla de Símbolos

La tabla fue implementada con la estructura Map<String, Map<String, String>>, donde la clave del mapa principal es el lexema, y en el segundo mapa se guarda como clave la clase de atributo que se guarda en el valor.

Esta tabla de símbolos se accede tanto en el analizador léxico para agregar la información y en el sintáctico tanto como para agregar información como para tomar decisiones en función de esta.

Como por ejemplo, el manejo de las constantes negativas, donde en la parte léxica no podemos saber si la constante es negativa, pero a la hora de llegar al análisis sintáctico una vez que se descubre si la constante es negativa o no puede modificarse la tabla de símbolos eliminando la entrada vieja o agregando una nueva.

Próximamente nos servirá para diferenciar si un ID es un tipo definido por el usuario o un identificador de una variable.

Esto nos permite a su vez guardar información adicional que nos sirve para otras cosas.

Manejo de errores y warnings léxicos

Al realizar el análisis léxico, como siempre se debe devolver un token, se decidió que al encontrar un error en el análisis lo que se hace es descartar lo leído hasta ese momento y buscar el siguiente token que pueda ser encontrado, es decir, se realiza una recursión sobre el mismo método que obtiene el próximo token.

En el caso de los warnings, se soluciona el problema que hay agregando/borrando caracteres u de otra forma, para poder devolver el token pero aún así se avisa de la acción realizada en el warning.

Token de sincronización con token error

Para poder utilizar el token de error es necesario un token de sincronización.

El más común es el “;”, el problema es que a veces la detección del punto y coma se realiza a nivel sentencia pero los token error los utilizamos en niveles más bajos de reglas.

Esto provoca que al encontrar un error, la regla utiliza el “;” lo que genera que en la regla de nivel sentencia piense que falta el “;”.

Esto lo solucionamos poniendo en el analizador léxico un método que lo obliga a devolver el token de sincronización que fue consumido por la regla.

Por ejemplo, en la regla
goto : GOTO TAG

```
        | GOTO error ';' {ErrorHandler.addErrorSintactico("falta la etiqueta en el  
GOTO, en caso de faltar también el punto y coma es posible que no compile el resto del  
programa o lo haga mal.", lex.getLinealInicial());  
                                lex.setErrorHandlerToken(";");}  
        ;
```

Al hacer el setErrorHandlerToken, el léxico en vez de leer el código que está compilando lo que hace es devolver el “;”

Comunicación entre analizador léxico y sintáctico

El analizador sintáctico lo único que espera recibir del léxico es el número de token, por lo tanto, para comunicar información como los lexemas de aquellos tokens que tienen más de una representación se utiliza la clase ParserVal con el atributo yylval en Parser. Con esto lo que hacemos es guardar la entrada a la tabla de símbolos de ese lexema para que el analizador sintáctico pueda diferenciarlo.

Modificaciones realizadas al Parser

Atributos agregados:

String nombreArchivo;

Para guardar el nombre del archivo a compilar

AnalizadorLexico lex;

Al crear el parser se crea el analizador léxico para poder llamar al método del mismo que devuelve el número de token.

TablaSimbolos ts;

String tipoVar;

Al estar analizando las reglas de declaración de variables o funciones, se guarda el tipo del mismo para poder guardar el tipo en la tabla de símbolos al terminar la regla correspondiente.

ArrayList<Integer> cantRetornos;

Utilizado para asegurar que las funciones tengan mínimo un retorno, controlando incluso las funciones anidadas.

String estructuras;

Guardamos en orden las estructuras que se van encontrando junto con su número de línea.

Métodos agregados:

```
public Parser(String nombreArchivo, TablaSimbolos t)
```

Se creo un nuevo constructor para el parser donde se le pasa el nombre del archivo a analizar y la tabla de símbolos creada.

```
int yylex() {  
    return lex.yylex();  
}
```

Implementamos el método yylex del parser llamando al método con el mismo nombre del analizador léxico, básicamente devuelve el nro de token y se utiliza en el método yyparse que básicamente hace el análisis sintáctico y devuelve si la gramática es válida o no.

```
String errores() {  
    return ErrorHandler.getErrores();  
}
```

Obtenemos los errores del ErrorHandler para mostrarlos.

```
boolean matcheanTipos(){  
    char firstC=yylval.sval.charAt(0);  
    if ( (tipoVar.equals(AccionSemantica.DOUBLE) && (firstC == 'x' || firstC == 'y' || firstC == 'z')) || (tipoVar.equals(AccionSemantica.ULONGINT) && (firstC == 'd')) ) {  
        return false;  
    }  
    return true;  
}
```

implementamos el método matcheanTipos para verificar que si tenemos una variable que comienza con (x,y,z que indica que es ulongint) no esté declarada como double, y lo mismo para las que comienzan con d (indica que es double) que no estén declaradas como tipo ulongInt

En la misma clase del parser a su vez es donde está ubicado el main del programa.

```
void checkRet(String nombreFuncion) {  
    if (!nombreFuncion.isEmpty()) {  
        if (this.cantRetornos.get(this.cantRetornos.size()-1) > 0){  
            estructurasSintacticas("Se declaró la función: " + nombreFuncion);  
            ts.addClave(nombreFuncion);  
  
            ts.addAtributo(nombreFuncion,AccionSemantica.TIPO,AccionSemantica.FUNCION);  
  
            ts.addAtributo(nombreFuncion,AccionSemantica.TIPORETORNO,tipoVar);  
        } else {  
            ErrorHandler.addErrorSintactico("Falta el retorno de la función: " +  
            nombreFuncion, lex.getLineaInicial());  
        }  
  
    } else {
```

```

        if (this.cantRetornos.get(this.cantRetornos.size()-1) == 0){
            ErrorHandler.addErrorSintactico("Falta el retorno de la función",
lex.getLinealInicial());
        }
    }
    this.cantRetornos.remove(this.cantRetornos.size()-1);
}

```

Es para modularizar todo el chequeo de que una función tenga retorno, y no ponerlo directamente en la gramática lo cual dificulta la visión de la misma.

```

public static String getNombreVariable(int numero) {
    switch (numero) {
        case YYERRCODE: return "YYERRCODE";
        case ID: return "ID";
        case CTE: return "CTE";
        case MASI: return ">=";
        case MENOS: return "<=";
        case ASIGN: return "!=";
        case DIST: return "!=";
        case GOTO: return "GOTO";
        case UP: return "UP";
        case DOWN: return "DOWN";
        case TRIPLE: return "TRIPLE";
        case FOR: return "FOR";
        case ULONGINT: return "ULONGINT";
        case DOUBLE: return "DOUBLE";
        case IF: return "IF";
        case THEN: return "THEN";
        case ELSE: return "ELSE";
        case BEGIN: return "BEGIN";
        case END: return "END";
        case END_IF: return "END_IF";
        case OUTF: return "OUTF";
        case TYPEDEF: return "TYPEDEF";
        case FUN: return "FUN";
        case RET: return "RET";
        case CADMUL: return "CADMUL";
        case TAG: return "TAG";
        default:
            // Si el número está en el rango ASCII (0-255), convierte a carácter
            if (numero >= 0 && numero <= 255) {
                char num = (char) numero;

```

```
        return String.valueOf(num);
    } else {
        return null;
    }
}
}
```

Es un método para retornar el string asociado al número de token para posteriormente imprimirlo.

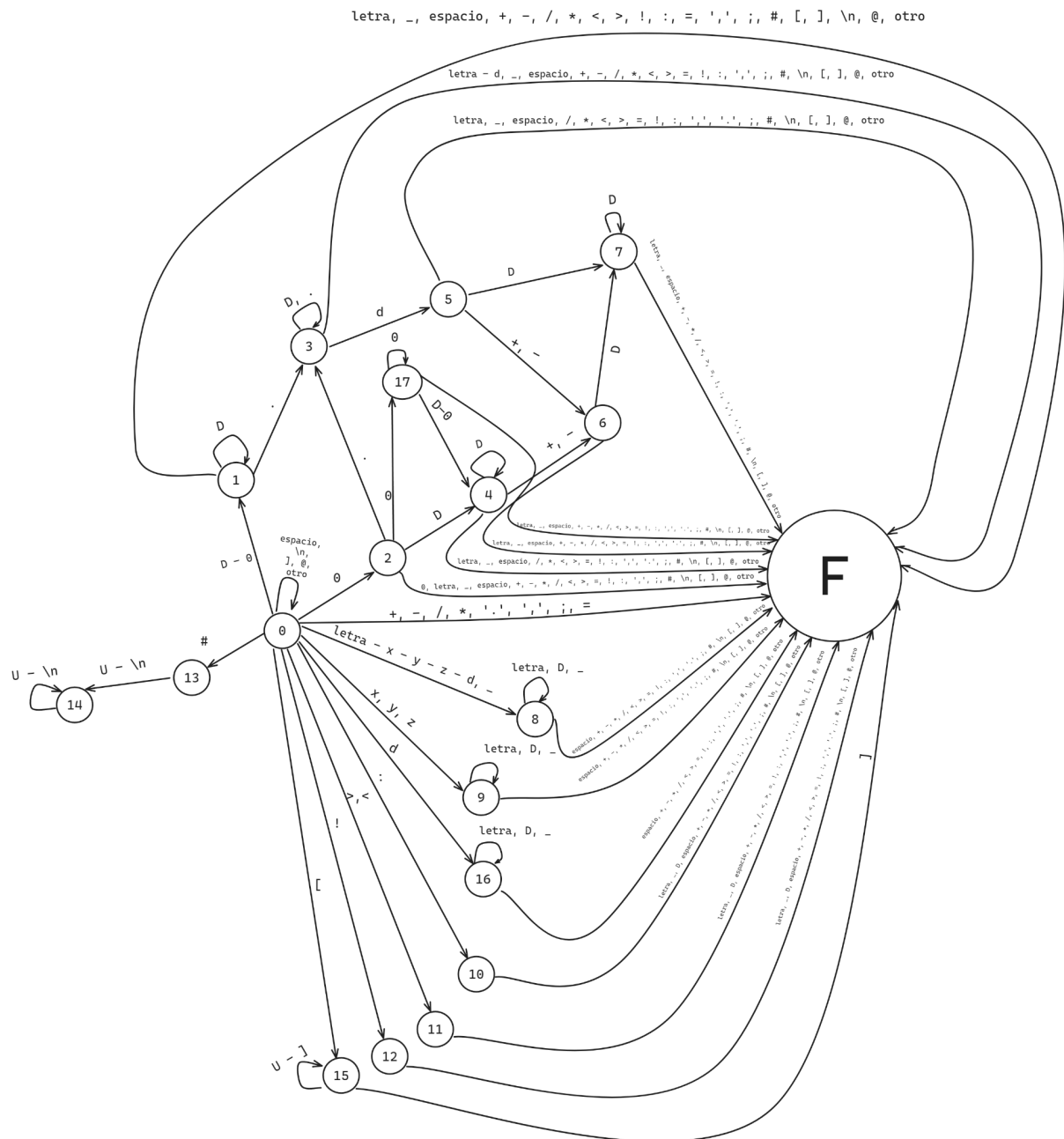
Decisión sobre las variables que su primera letra indica su tipo

Para estos casos igualmente obligamos a que el usuario la declare de ese tipo a la variable, es decir no puede declarar una variable solo escribiendo `xnombreVariable`, sino que debe declararla como `ulongint xnombreVariable`.

Modificación a la gramática de la etiqueta

La declaración de una etiqueta es la única sentencia que no lleva `;`, esto genera un problema en nuestra gramática debido a en qué regla ubicamos los `;` que no es la misma regla donde estan todas las sentencias, por lo tanto el mensaje necesario para hacer que esa sentencia en particular no lleve `;` generaba muchos problemas y decidimos que la declaración de una etiqueta lleve `;`.

Diagrama de transición de estados




Aclaraciones

- En el caso del estado 2, si viene un 8 o un 9 vamos al estado 4 pero la acción semántica es distinta (a las del 0-7) ya que descartamos estos valores pero seguimos formando el octal.
- En el estado 4 ocurre lo mismo del estado 2 pero al ciclar sobre si mismo.
- D son todos los dígitos del 0 al 9.
- letra es todo el abecedario.
- U es todo el universo del lenguaje, conteniendo los caracteres especiales que el mismo admite.


Matriz de transición de estados

Link al excel con la matriz de transicion de estados

 [matriz](#)

Matriz de acciones semánticas

Link al excel con la matriz de acciones semanticas

 [matriz](#)

Lista de Acciones Semánticas

Implementación

Se creó una clase abstracta AcciónSemántica con el metodo “public abstract void ejecutar(AnalizadorLexico al);”, este metodo es el que se va utilizar al transicionar estados básicamente ubicando objetos de clases hijas de Acción Semántica que llaman al método “ejecutar” al hacer una transicion de estados.

Entonces para todas acciones semánticas se crea una clase que extiende de AccionSemantica y se sobrescribe el método ejecutar con la acción que se crea necesaria para esa transición.

Acciones

AS0

Descarta los espacios estando en estado cero (avanzando posición en la línea)

AS1

Concatena y avanza posición, se utiliza cuando se quiere seguir leyendo el token ya que el próximo caracter es válido.

ASF1

Hace AS1 y retorna el token detectado. Se utiliza cuando se termina de detectar un token, principalmente para símbolos, y se lo quiere devolver.

ASF1Comp

Hace ASF1

ASFBR

Chequea que el lexema esté en la tabla de símbolos, si lo está actualiza el atributo de cantidad. Si no lo está lo añade a la tabla de símbolos, añade el atributo tipo y cantidad. Luego retorna el número de token y actualiza el valor de yylval del Parser con el lexema.

ASF1OCTAL

Chequea rango de octal, setea el tipo de la constante, hace AS1 y luego hace la acción de ASFBR

ASF2Double

Chequea rango del double, setea el tipo de la constante, hace AS1 y luego hace la acción de ASFBR

ASF2OCTAL

Chequea rango de octal, setea el tipo de la constante y luego hace la acción de ASFBR.

ASF2LongInt

Chequea rango de ulongint, setea el tipo de la constante, hace AS1 y luego hace la acción de ASFBR

ASF2COMP

Solamente setea el numero de token.

ASBR

No realiza nada ya que los saltos de línea están tratados implícitamente por nuestro manejo del archivo.

ASBR2

Concatena un espacio al comentario multilínea, ya que al terminar la línea y leer la próxima, deben estar en la misma línea.

ASFBR3

Chequea si es una palabra reservada, y si es de un tipo setea el tipo que es una variable del parser, y luego da el token.

Si no era palabra reservada, es un ID entonces chequea el rango del string y lo añade a la tabla de símbolos y actualiza yylval del Parser.

ASFBR4

Se termina de leer un ID, chequea el rango del string, lo añade a la tabla de símbolos y añade a ésta el tipo que se identificó en el primer caracter del ID (en este caso ulongint). Retorna el token y actualiza yylval del Parser.

ASFBR5

Chequea si es una palabra reservada, y si es de un tipo setea el tipo que es una variable del parser, y luego da el token.

Si no era palabra reservada, es un ID, chequea el rango del string, lo añade a la tabla de símbolos y añade a ésta el tipo que se identificó en el primer caracter del ID (en este caso double). Retorna el token y actualiza yylval del Parser.

ASE1

Falta la letra que inicie el identificador, es cuando se arranca a leer desde el estado 0 un “_”.

ASE2

Se encuentra "]" que cerrará una cadena multilínea, pero como no se tenía un "[" que la abra, lo descartamos avanzando posición sin concatenar y tiramos un warning.

ASE3

Caracter inesperado (no esperado por el lenguaje), se descarta y se avanza posición sin concatenar. Se avisa con un warning.

ASE4

Se está armando una constante entera pero viene una "d", se descarta y se avanza posición pero se avisa que si se quiso armar un double faltó la parte decimal.

ASE5

Se está armando un octal y viene un dígito que no pertenece a los octales (8 o 9), se descarta, se avanza y se avisa con un warning.

ASE7

Se está armando un float y vino más de un punto seguido, por lo que se lo descarta y se avanza y se avisa con un warning.

ASE8

Se termina de armar un double pero no tiene dígitos el exponente, por lo que se tira un ERROR.

ASE9

Se está realizando una asignación o una comparación y faltó el "=", se lo agrega, se avisa con un warning y se setea el numero de token.

ASE10

Vino un @ sin antes especificar un ID, es descartado

ASE11

Faltó un #, se lo agrega, se avisa con un warning y hace AS1.

ASDescartaComentario

Setea el concatenado actual a un String vacío ""

ASFGOTO

Se chequea el rango del string del TAG, se añade a la tabla de símbolos, se retorna el número de token, se añade a la tabla de símbolos el atributo tipo con un "tag" y se setea el yylval del Parser.

Errores y Warnings Léxicos considerados

Warnings

Se notificará de un warning cuando:

- No se tiene un “[“ que inicie la cadena multilínea y viene un “]” que la cierra, lo consideramos como un error de tipeo y descartamos.
- Se lee un caracter no esperado por el lenguaje, se lo descarta ya que es algo que el programador no debería haber ingresado apropiado.
- Se va a realizar una asignación, que se lee “:” y falta el “=”, se lo agrega ya que la única razón por la cual se ingresaría un “:” es para realizar una asignación.
- Se encuentra un # ya se sabe que se ingresará un comentario, por lo que si falta el segundo # se lo agrega.
- Se está armando una constante entera y viene una “d”, faltó toda la parte decimal, por lo que suponemos que no se hizo apropiado ya que siempre va precedida por ésta, se descarte y se sigue.
- Se está armando un octal y viene un dígito que no pertenece a los octales (8 o 9) se lo descarta (se sabe que es octal porque es la única razón por la que la constante ingresada empiece por 0)
- Se está armando un float y viene más de un punto seguido, por lo que lo descartamos.

Errores

- Falta la letra que inicie un identificador, es cuando se arranca un identificador con “_” y se lo considera un error.
- Se termina de armar un double pero no tiene dígitos el exponente, por lo que se considera un error.

Desarrollo del Analizador Sintáctico

Problemas surgidos y soluciones adoptadas

Uno de los principales y únicos problemas que surgieron, fue que cuando ya habíamos realizado la gramática, sin contemplar los errores, no tuvimos ningún tipo de conflicto y funciona bien. Pero a la hora de detectar los errores que fueron pedidos detectar, nos surgieron muchos conflictos. Las soluciones a estos, se explicarán en la próxima sección correspondiente.

Tuvimos problemas en la conexión del Parser con el analizador léxico y la tabla de símbolos, y en ciertas cosas que debía tener el Parser y, por lo tanto, tuvimos que hacerle varias modificaciones (explicadas [más arriba](#))

Manejo de errores

Para mostrar los errores solo mostraremos las reglas añadidas para poder detectar específicamente tal error.

Falta de punto y coma

Se añade una regla en la que falta el punto y coma para poder detectarlo.

cuerpo_error : BEGIN sentencias ',' END

```
| BEGIN sentencia END { ErrorHandler.addErrorSintactico("Falta punto y  
coma");}  
;
```

cuerpo : BEGIN sentencias ',' END

```
| BEGIN sentencia END { ErrorHandler.addErrorSintactico("Falta punto y  
coma");}  
;
```

sentencias : sentencias ',' sentencia
| sentencia

```
| sentencias sentencia {ErrorHandler.addErrorSintactico("Falta punto y  
coma");}  
;
```

cuerpo_control : BEGIN multip_cuerp_fun ',' END
| sentec_eject ','

```
| BEGIN multip_cuerp_fun END { ErrorHandler.addErrorSintactico("Falta  
punto y coma");}  
;
```

cuerpo_iteracion: BEGIN multip_cuerp_fun ',' END
| sentec_eject ','

```
| BEGIN multip_cuerp_fun END { ErrorHandler.addErrorSintactico("Falta  
punto y coma");};
```

```
multip_cuerp_fun: multip_cuerp_fun ';' sentec_eject  
| sentec_eject
```

```
| multip_cuerp_fun sentec_eject { ErrorHandler.addErrorSintactico("Falta  
punto y coma");}  
;
```

```
bloques_funcion : bloques_funcion ';' bloque_funcion  
| bloque_funcion
```

```
| bloques_funcion bloque_funcion {ErrorHandler.addErrorSintactico("Falta  
punto y coma");}  
;
```

```
for : FOR '(' ID ASIGN CTE ';' condicion ';' foravanc CTE ')' cuerpo_iteracion  
{estructurasSintacticas("Se declaró un bucle FOR en la linea: " + lex.getLinealInicial());}
```

```
| FOR '(' ID ASIGN CTE ';' condicion foravanc CTE ')' cuerpo_iteracion {  
ErrorHandler.addErrorSintactico("Falta punto y coma entre condicion y avance");}  
| FOR '(' ID ASIGN CTE condicion ';' foravanc CTE ')' cuerpo_iteracion {  
ErrorHandler.addErrorSintactico("Falta punto y coma entre asignacion y condicion");}  
| FOR '(' ID ASIGN CTE condicion foravanc CTE ')' cuerpo_iteracion {  
ErrorHandler.addErrorSintactico("Faltan todos los punto y coma del for");}  
;
```

Falta de nombre de programa

```
prog : ID cuerpo { estructurasSintacticas("Se declaró el programa: " + $1.sval);}  
  
| cuerpo_error { ErrorHandler.addErrorSintactico("Falta el nombre del  
programa");}  
;
```

Falta de delimitador de programa

```
cuerpo : BEGIN sentencias ';' END
```



```

| BEGIN sentencias ';' { ErrorHandler.addErrorSintactico("Falta delimitador
END del programa");}
| sentencias ';' END { ErrorHandler.addErrorSintactico("Falta delimitador
BEGIN del programa");}
| sentencias ';' { ErrorHandler.addErrorSintactico("Falta delimitador BEGIN y
END del programa");}
;

```

Falta de nombre en función

declaracion_fun : tipo_fun FUN ID '(' lista_parametro ')'

```

| tipo_fun FUN '(' lista_parametro ')' cuerpo_funcion_p {
ErrorHandler.addErrorSintactico("Falta nombre de la funcion declarada");}
;

```

Falta de sentencia RET en función

Esto se resuelve mediante acciones dentro de cada regla.

```

retorno      : RET '('expresion')' { this.cantRetornos.set(this.cantRetornos.size()-1,
this.cantRetornos.get(this.cantRetornos.size()-1) + 1); }
;

```

Acá aumentamos la cantidad de retornos dentro del nivel de anidación de funciones (Esto lo controlamos con una lista con un elemento de cantidad de retornos para cada funcion)

```

declaracion_fun : tipo_fun FUN ID '(' lista_parametro ')' { this.cantRetornos.add(0); }
cuerpo_funcion_p {
    if (this.cantRetornos.get(this.cantRetornos.size() - 1) > 0) {
        estructurasSintacticas("Se declaró la función: " + $3.sval);
        ts.addClave($3.sval);
        ts.addAtributo($3.sval, AccionSemantica.TIPO,
AccionSemantica.FUNCION);
        ts.addAtributo($3.sval, AccionSemantica.TIPORETORNO, tipoVar);
    } else {
        lex.addErrorSintactico("Falta el retorno de la función: " + $3.sval);
    }
    this.cantRetornos.remove(this.cantRetornos.size() - 1);
}
;

```

Cuando se declara la función, antes del cuerpo, añadimos un elemento con cantidad nula de retornos. Si la cantidad de retornos del nivel de anidación actual es mayor a 0, cumple con el requisito del retorno. De lo contrario, se genera un error sintáctico.

Cuando termina el cuerpo de la función eliminamos la entrada de cantidad de retornos de ese nivel de anidación ya que terminó.

Falta de “,” en declaración de variables

```
lista_variables : lista_variables ',' ID
```

```
    | ID
    | lista_variables error ID { ErrorHandler.addErrorSintactico("Falta coma en la
lista de variables, puede haber parado la compilacion en este punto");}
;
```

Para detectar este error tuvimos que utilizar el token de error, comiendo todo el programa hasta encontrar un ID.

No es la mejor solución ya que si no encuentra un ID puede comer mucha parte del programa hasta sincronizar.

Si el usuario pone un carácter erróneo en lugar de la coma, se avisa que falta la coma.

Si no pone nada, el error es detectado a un nivel más alto como “Sentencia inválida”, pero debe tener ;

Falta de nombre/tipo de parámetro formal en declaración de función

```
parametro      : tipo ID {estructurasSintacticas("Se declaró el parámetro: " + $2.sval + " en la
linea: " + lex.getLinealInicial());}
```

```
    | ID ID {estructurasSintacticas("Se declaró el parámetro: " + $2.sval + " en la
linea: " + lex.getLinealInicial());}
```

```
    | tipo { ErrorHandler.addErrorSintactico("Falta el nombre del parametro");}
    | ID { ErrorHandler.addErrorSintactico("Falta el nombre del parametro o el
tipo");}
;
```

Dado que nos tocó el tema de TRIPLA, al poder ser el ID tanto un ID como un tipo definido por el usuario de tripla, la detección de si falta el nombre del parametro tripla o el tipo de la tripla definido por el usuario

A su vez, la misma regla que tiene tal ambigüedad con la tripla, puede estar asociada a que se iba a ingresar un “tipo id” y faltó el tipo (no definido por el usuario, basico)

Tal detección se postergará para la proxima etapa

Cantidad errónea de parámetros en declaración o invocación de función

Dado que únicamente se debe poder declarar o invocar con un solo parámetro, agregamos una regla que permita más para poder detectar que se pasó una cantidad errónea.

```
lista_parametro : lista_parametro ',' parametro { ErrorHandler.addErrorSintactico("Se declaró
más de un parametro");}
```

```
| parametro  
;
```

Esto es el caso de que haya más de un parámetro. Aún así, no decidimos considerar la falta de coma entre más de un parámetro ya que de por sí éste no es un comportamiento esperado.

```
declaracion_fun : tipo_fun FUN ID '(' lista_parametro ')'  
                | tipo_fun FUN ID '(' ')' cuerpo_funcion_p {  
ErrorHandler.addErrorSintactico("Falta el parametro en la declaracion de la funcion");}  
                ;
```

Y este es cuando NO hay parámetros.

La misma estrategia se utilizó para la invocación de funciones

```
invoc_fun      : ID '(' lista_parametro_real ')' {estructurasSintacticas("Se invocó a la función:  
" + $1.sval + " en la linea: " + lex.getLinealInicial());}
```

```
                | ID '(' ')' { ErrorHandler.addErrorSintactico("Falta de parámetros en la  
invocación a la función");}  
                ;
```

```
lista_parametro_real : lista_parametro_real ',' param_real {  
ErrorHandler.addErrorSintactico("Se utilizó más de un parámetro para invocar ala función");}  
                    | param_real  
                    ;
```

Falta parámetro en sentencia OUTF

```
sald_mensaj   : OUTF '(' mensaje ')'
```

```
                | OUTF '(' ')' { ErrorHandler.addErrorSintactico("Falta el mensaje del OUTF");}  
                ;
```

Se utiliza una regla que falta el mensaje.

Parámetro incorrecto en sentencia OUTF

```
sald_mensaj   : OUTF '(' mensaje ')'
```

```
                | OUTF '(' error ')' { ErrorHandler.addErrorSintactico("Parámetro invalido del  
OUTF");}  
                ;
```

Si se detecta cualquier cosa que no sea un mensaje, utilizamos un token de error.

Falta de paréntesis en condición de selecciones e iteraciones

```
condicion     : '(' condicion_2 ')'
```

```
                | condicion_2 { ErrorHandler.addErrorSintactico("Falta de paréntesis en la  
condición");}
```

```

        | '(' condicion_2 { ErrorHandler.addErrorSintactico("Falta de paréntesis
derecho en la condición");}
        | condicion_2 ')' { ErrorHandler.addErrorSintactico("Falta de paréntesis
izquierdo en la condición");}
    ;

```

Todas las combinaciones de falta de paréntesis.

Falta de cuerpo en iteraciones

```

cuerpo_iteracion: BEGIN multip_cuerp_fun ';' END
                | sentec_eject ';'

```

```

        | BEGIN ';' END { ErrorHandler.addErrorSintactico("Falta el cuerpo de la
iteracion");}
        | BEGIN END { ErrorHandler.addErrorSintactico("Falta el cuerpo de la
iteracion");}
    ;

```

Pusimos ambos casos con y sin punto y coma, en los que falta el cuerpo.

Falta de END_IF

```

seleccion      : IF condicion THEN cuerpo_control END_IF {estructurasSintacticas("Se
definió una sentencia de control sin else, en la línea: " + lex.getLinealInicial());}
                IF condicion THEN cuerpo_control ELSE cuerpo_control END_IF
{estructurasSintacticas("Se definió una sentencia de control con else, en la línea: " +
lex.getLinealInicial());}

```

```

        | IF condicion THEN cuerpo_control ELSE cuerpo_control {
lex.addErrorSintactico("Falta END_IF con ELSE");}
        | IF condicion THEN cuerpo_control { ErrorHandler.addErrorSintactico("Falta
END_IF");}
    ;

```

Pusimos las posibles combinaciones de falta de END_IF

Falta de contenido en bloque THEN/ELSE

```

seleccion      : IF condicion THEN cuerpo_control END_IF {estructurasSintacticas("Se
definió una sentencia de control sin else, en la línea: " + lex.getLinealInicial());}
                | IF condicion THEN cuerpo_control ELSE cuerpo_control END_IF
{estructurasSintacticas("Se definió una sentencia de control con else, en la línea: " +
lex.getLinealInicial());}

```

```

        | IF condicion THEN END_IF{ ErrorHandler.addErrorSintactico("Falta el
cuerpo de control del then");}
        | IF condicion THEN cuerpo_control ELSE END_IF{
lex.addErrorSintactico("Falta el cuerpo de control del ELSE");}
        | IF condicion THEN ELSE END_IF{ ErrorHandler.addErrorSintactico("Falta el
cuerpo de control tanto en THEN como ELSE");}

```

;

Todas las combinaciones posibles de cuerpos ausentes.

Falta de operando en expresión

```
expresion_matematica      : expresion_matematica '+' termino
                           | expresion_matematica '-' termino
                           | termino
```

```
                           | '+' termino { lex.addErrorSintactico("Falta operando izquierdo");}
                           | expresion_matematica '+' error ')' { ErrorHandler.addErrorSintactico("Falta
operando derecho");}
                           | expresion_matematica '+' error ';' { ErrorHandler.addErrorSintactico("Falta
operando derecho");}
                           | expresion_matematica '-' error ')' { ErrorHandler.addErrorSintactico("Falta
operando derecho");}
                           | expresion_matematica '-' error ';' { ErrorHandler.addErrorSintactico("Falta
operando derecho");}
                           //| '-' termino
                           ;
```

Es imposible detectar la falta de operando izquierdo en una resta, por la constante negativa.

```
termino                   : termino '*' factor
                           | termino '/' factor
                           | factor
```

```
                           | '*' factor { ErrorHandler.addErrorSintactico("Falta operando izquierdo");}
                           | '/' factor { ErrorHandler.addErrorSintactico("Falta operando izquierdo");}
                           | termino '/' error ')' { ErrorHandler.addErrorSintactico("Falta operando
derecho");}
                           | termino '*' error ')' { ErrorHandler.addErrorSintactico("Falta operando
derecho");}
                           | termino '*' error ';' { ErrorHandler.addErrorSintactico("Falta operando
derecho");}
                           | termino '/' error ';' { ErrorHandler.addErrorSintactico("Falta operando
derecho");}
                           ;
```

Falta de operador en expresión

```
expresion_matematica      : expresion_matematica '+' termino
```

```

| expresion_matematica '-' termino
| termino

//| expresion_matematica termino { ErrorHandler.addErrorSintactico("Falta
operador en la expresión matematica");}
;

termino      : termino '*' factor
              | termino '/' factor
              | factor

//| termino factor { ErrorHandler.addErrorSintactico("Falta operador en el
término");}
;

```

No se pudo solucionar directamente pero como la expresión también puede ser una expresión booleana, ya que se puede asignar una comparación a una variable, entonces cuando falta el operador lo toma como que falta el comparador en la comparación.

Por lo tanto, si bien no está solucionado, el programa sigue ejecutándose y lo considera como que es una comparación errónea.

Falta de comparador en comparación

```

condicion_2  : expresion_matematica comparador expresion_matematica
              | patron comparador patron

              | patron error patron {ErrorHandler.addErrorSintactico("Falta comparador
entre los patrones");}
              | expresion_matematica error expresion_matematica {
ErrorHandler.addErrorSintactico("Falta comparador entre las expresiones");}
;

```

Ambos fueron solucionados con un token de error.

Tema 16: Falta “,”. Falta de UP/DOWN

```

for          : FOR '(' ID ASIGN CTE ';' condicion ';' foravanc CTE ')' cuerpo_iteracion
{estructurasSintacticas("Se declaró un bucle FOR en la linea: " + lex.getLinealInicial());}

```

```

| FOR '(' ID ASIGN CTE ';' condicion foravanc CTE ')' cuerpo_iteracion {
ErrorHandler.addErrorSintactico("Falta punto y coma entre condicion y avance");}
| FOR '(' ID ASIGN CTE condicion ';' foravanc CTE ')' cuerpo_iteracion {
ErrorHandler.addErrorSintactico("Falta punto y coma entre asignacion y condicion");}
| FOR '(' ID ASIGN CTE condicion foravanc CTE ')' cuerpo_iteracion {
ErrorHandler.addErrorSintactico("Faltan todos los punto y coma del for");}

```

```

| FOR '(' ID ASIGN CTE ';' condicion ';' CTE ')' cuerpo_iteracion {
ErrorHandler.addErrorSintactico("Falta UP/DOWN");}
| FOR '(' ID ASIGN CTE ';' condicion ';' foravanc ')' cuerpo_iteracion
{ErrorHandler.addErrorSintactico("Falta valor del UP/DOWN");}
| FOR '(' ID ASIGN CTE ';' condicion CTE ')' cuerpo_iteracion {
ErrorHandler.addErrorSintactico("Falta UP/DOWN y punto y coma entre condicion y
avance");}
| FOR '(' ID ASIGN CTE ';' condicion foravanc ')' cuerpo_iteracion {
ErrorHandler.addErrorSintactico("Falta valor del UP/DOWN y punto y coma entre condicion
y avance");}
| FOR '(' ID ASIGN CTE ';' condicion ';' ')' cuerpo_iteracion { {
ErrorHandler.addErrorSintactico("Falta UP/DOWN, su valor, y punto y coma entre condicion
y avance");}}
;

```

Todas las posibles combinaciones de falta de punto y coma, y up y down.

Tema 22: Falta triple. Falta <>. Falta identificador al final de la declaración

declar_tipo_trip: TYPEDEF TRIPLE '<' tipo '>' ID {System.out.println("Se declaró un tipo TRIPLE con el ID: " + \$6.sval + " en la línea:" + lex.getLinealInicial());}

```

| TYPEDEF TRIPLE tipo '>' ID {ErrorHandler.addErrorSintactico("falta < en la
declaración del TRIPLE"); }
| TYPEDEF TRIPLE '<' tipo ID {ErrorHandler.addErrorSintactico("falta > en la
declaración del TRIPLE"); }
| TYPEDEF TRIPLE tipo ID {ErrorHandler.addErrorSintactico("falta > y < en
la declaración del TRIPLE"); }
| TYPEDEF '<' tipo '>' ID { ErrorHandler.addErrorSintactico("Falta la palabra
clave TRIPLE");}
| TYPEDEF TRIPLE '<' tipo '>' error ';' {
ErrorHandler.addErrorSintactico("Falta el ID de la tripla definida.");}
;

```

Tema 23: Falta etiqueta

goto : GOTO TAG

```

| GOTO error ';' {ErrorHandler.addErrorSintactico("falta la etiqueta en el
GOTO, en caso de faltar también el punto y coma es posible que no compile el resto del
programa o lo haga mal.");}
;

```

Solución de conflictos shift/reduce y reduce/reduce

No tenemos documentados todos los errores shift/reduce o reduce/reduce que fueron ocurriendo.

Conflicto shift/reduce de falta ID o Begin en nombre

No se puede simultáneamente la falta de un ID del programa junto a la falta del delimitador BEGIN del programa, razón por la cual utilizamos un cuerpo_error para el caso que falta el nombre del programa nunca falte el BEGIN (ya que provoca un shift reduce al estar en el estado "ID . ID", no sabe si reducir o leer el proximo ID)

```
prog          : ID cuerpo { estructurasSintacticas("Se declaró el programa: " + $1.sval);}

               | cuerpo_error { ErrorHandler.addErrorSintactico("Falta el nombre del
               programa");}
               ;

cuerpo_error  : BEGIN sentencias ';' END

               | BEGIN sentencia END { ErrorHandler.addErrorSintactico("Falta punto y
coma");}
               | BEGIN sentencias ';' { ErrorHandler.addErrorSintactico("Falta delimitador del
               programa");}
               ;

cuerpo        : BEGIN sentencias ';' END

               | BEGIN sentencia END { ErrorHandler.addErrorSintactico("Falta punto y coma");}
               | BEGIN sentencias ';' { ErrorHandler.addErrorSintactico("Falta delimitador END del
               programa");}
               | sentencias ';' END { ErrorHandler.addErrorSintactico("Falta delimitador BEGIN del
               programa");}
               | sentencias ';' { ErrorHandler.addErrorSintactico("Falta delimitador BEGIN y END del
               programa");}
               ;
```

Conflicto de falta de punto y coma

Antes se tenía el programa de la siguiente forma, obligando a poner el punto y coma a cada sentencia declarativa:

```
sentec_declar : declaracion_var ',';
```



```

        | declaracion_fun ';' { estructurasSintacticas("Se declaro la funcion, en linea:
" + lex.getLinealInicial()); }
        | TAG ';' { estructurasSintacticas("Se declaro una etiqueta goto, en linea:
" + lex.getLinealInicial()); }
        | declar_tipo_trip ';'
    ;

```

Lo mismo para para las sentencias ejecutables.

Esta regla es tanto para el cuerpo de control como para el cuerpo de un FOR.

```

multip_cuerp_fun: multip_cuerp_fun ';' sentec_eject
    | sentec_eject

    | multip_cuerp_fun sentec_eject { ErrorHandler.addErrorSintactico("Falta
punto y coma");}
    ;

```

Al poner las mismas reglas sin punto y coma para generar los errores, se generaba un shift/reduce para cada regla.

La solución que adoptamos fue subir la restricción de punto y coma dos niveles arriba en la gramática y así pudimos controlarla fácilmente.

```

sentencias : sentencias ';' sentencia
    | sentencia

    | sentencias sentencia {ErrorHandler.addErrorSintactico("Falta punto y
coma");}
    ;

```

```

sentencia : sentec_declar
    | sentec_eject
    ;

```

```

sentec_declar : declaracion_var
    | declaracion_fun { estructurasSintacticas("Se declaro la funcion, en linea:
" + lex.getLinealInicial()); }
    | TAG {estructurasSintacticas("Se declaro una etiqueta goto, en linea: " +
lex.getLinealInicial()); }
    | declar_tipo_trip
    ;

```

Se realizó un enfoque similar para poder detectar la falta de punto y coma en el bloque de las funciones (sentencias y retornos).

```

bloques_funcion : bloques_funcion ';' bloque_funcion

```

```

        | bloque_funcion

        | bloques_funcion bloque_funcion {ErrorHandler.addErrorSintactico("Falta
punto y coma");}
        ;

bloque_funcion : retorno
                | sentencia
                ;

```

Conflicto shift/reduce al sacar el id en declaración de triple

```

declar_tipo_trip: TYPEDEF TRIPLE '<' tipo '>' ID {System.out.println("Se declaró un tipo
TRIPLE con el ID: " + $6.sval + " en la línea:" + lex.getLinealInicial());}

```

```

        | TYPEDEF TRIPLE tipo '>' ID {ErrorHandler.addErrorSintactico("falta < en la
declaración del TRIPLE"); }
        | TYPEDEF TRIPLE '<' tipo ID {ErrorHandler.addErrorSintactico("falta > en la
declaración del TRIPLE"); }
        | TYPEDEF TRIPLE tipo ID {ErrorHandler.addErrorSintactico("falta > y < en
la declaración del TRIPLE"); }
        | TYPEDEF '<' tipo '>' ID { ErrorHandler.addErrorSintactico("Falta la palabra
clave TRIPLE");}
        | TYPEDEF TRIPLE '<' tipo '>' error ';' {
ErrorHandler.addErrorSintactico("Falta el ID de la tripla definida.");}
        ;

```

Al intentar detectar la falta del ID en la declaración de una tripla, se generaba un conflicto shift/reduce, que lo solucionamos utilizando con el token de error y sincronizando con un ';', que, potencialmente, debería estar al lado (y no comerse todo el programa).

Conflicto al utilizar tripla como un tipo

En un principio, dentro de la regla tipo, además de los tipos básicos, agregamos ID para tratar los tipos definidos por el usuario de TRIPLA.

Esto generó conflictos de shift reduce ya que al leer un ID no se sabe si se está leyendo un tipo, y reducir ahí, o esperar otro ID.

```

declaracion_var : tipo lista_variables {estructurasSintacticas("Se declararon variables en la
línea: " + lex.getLinealInicial());}
                ;

```

```

tipo           : DOUBLE

```

```

        | ULONGINT
        | ID
        ;

```

La solución fue quitar ID del tipo y ponerlo en las reglas que utilizaban tipo de la siguiente forma:

```

declaracion_var : tipo lista_variales {estructurasSintacticas("Se declararon variables en la
linea: " + lex.getLinealInicial());}
                | ID lista_variales {estructurasSintacticas("Se declararon variables en la
linea: " + lex.getLinealInicial());}
                ;

```

Conflicto en GOTO sin TAG

Para poder detectar la falta del tag en el goto, se debió utilizar un token de error porque de lo contrario se generaba un shift reduce:

```

goto          : GOTO TAG

```

```

        | GOTO error ';' {ErrorHandler.addErrorSintactico("falta la etiqueta en el
GOTO, en caso de faltar también el punto y coma es posible que no compile el resto del
programa o lo haga mal.");}
        ;

```

No terminales usados en la gramática

prog

no terminal inicial que representa todo el programa

cuerpo_error

cuerpo del programa que no permite falta de BEGIN

cuerpo

cuerpo del programa que permite falta de delimitadores

sentencias

regla recursiva que permite multiple declaración de sentencias forzando ‘;’

sentencia

contiene sentencias ejecutables y declarativas

sentec_declar

permite declaración de variables, funciones, tag y triplas

sentec_eject

permite ejecucion de funciones, asignación, selección, mensajes, for y goto

condicion

añade parentesis a condicion_2

condicion_2

permite comparación normal y pattern matching

patron

obliga a que se utilice más de una expresión matemática para pattern matching.

lista_patron
permite recursión para añadir cualquier cantidad de expresiones matemáticas.

seleccion
permite la declaración de una sentencia de control

comparador
define el tipo de comparador a utilizar

cuerpo_control
permite desarrollar un cuerpo dentro de una sentencia de control

cuerpo_iteracion
permite desarrollar un cuerpo dentro de una iteración del FOR

multip_cuerp_fun
permite recursión generando un cuerpo, forzando la utilización de punto y coma a alto nivel

declaracion_var
permite la declaración de variables de tipos básicos y definidos por el usuario.

lista_variables
permite la recursión para generar una o más variables

tipo
contiene los tipos básicos DOUBLE y ULONGINT

asignacion
permite asignar una expresión a un ID o a un elemento de la tripla

expresion
contiene los dos tipos principales de expresiones, las matemáticas y condicion_2 (que son comparaciones booleanas)

expresion_matematica y termino
permite generar expresiones matemáticas con distintos grados de precedencia

factor
genera los factores de las expresiones matemáticas

constante
engloba las constantes positivas y negativas para poder detectar las negativas

triple
permite el acceso a un elemento de la tripla

declaracion_fun
permite la declaración de una función

tipo_fun
define el tipo de retorno de la función que puede ser un tipo básico o un ID. Se creó solo para no hacer tan grande la gramática copiando y pegando ya que se realizan acciones sobre la misma.

lista_parametro
se creó para poder detectar si el programador intenta pasar más de un parámetro, lo cual no es correcto.

parametro
permite declarar un parametro

cuerpo_funcion_p
cuerpo de la función declarada

bloques_funcion

permite generar recursivamente todo el cuerpo de la funcion, obligando la utilización de punto y coma a un nivel más alto jerárquicamente

bloque_funcion

contiene retorno y sentencia

retorno

permite la funcionalidad del retorno de la funcion

invoc_fun

permite la invocación a una función

lista_param_real

permite controlar si el usuario quiere pasar más de un parámetro, lo cual no debe ocurrir.

param_real

permite pasar parametros y realizar una conversión explícita

sald_mensaje

permite imprimir un mensaje

mensaje

define el mensaje como una expresion o como una cadena multiple

for

define el bucle FOR

foravanc

dirección del avance del contador (positiva/negativa, up/down)

goto

permite el salto a una etiqueta

declar_tipo_trip

permite la declaración de un tipo definido por el usuario, específicamente la tripla.

Archivos para pruebas

Prueba Léxica

Para la prueba de parte del analizador léxico, se debe ingresar por consola el siguiente comando, desde la ruta que se ve:

```
C:\Users\logue\eclipse-workspace\TPCompilador\src>java -jar pruebaLexico.jar  
Se leyo correctamente el archivo
```

Leerá un archivo que contiene errores y warnings léxicos con cosas que se debían tener en cuenta como rangos, truncar IDs y algunas decisiones de diseño nuestras.

Prueba Sintáctica

Para los errores sintácticos se tiene que pasar por parámetro del ejecutable el archivo “pruebaCodigoSintacticoErrores”.

```
C:\Users\logue\eclipse-workspace\TPCompilador\src>java -jar Compilador.jar pruebaCodigoSintacticoErrores  
Se leyo correctamente el archivo
```

Para todo lo que debe reconocer, sin errores, se debe ejecutar el archivo “pruebaCodigoSintactico”

```
C:\Users\logue\eclipse-workspace\TPCompilador\src>java -jar Compilador.jar pruebaCodigoSintactico  
Se leyo correctamente el archivo
```

Conclusión

En esta primera entrega del Trabajo Práctico, hemos implementado con éxito las primeras dos etapas esenciales de un compilador: el análisis léxico y el análisis sintáctico. Siguiendo las consignas establecidas por la cátedra, hemos desarrollado un analizador léxico que identifica correctamente los diferentes tokens, y un parser que, a partir de esos tokens, construye la estructura sintáctica del lenguaje asignado.

El desarrollo de ambas etapas nos permitió aplicar conceptos teóricos en la práctica, utilizando la herramienta yacc en conjunto con Java para lograr una correcta implementación. Se logró detectar una amplia variedad de errores léxicos y sintácticos, asegurando una notificación detallada al usuario cuando el código contiene inconsistencias mediante warnings. También implementamos estrategias para la detección y manejo de errores comunes.

Durante el proceso, nos encontramos con problemas, como la comunicación entre el analizador léxico y el parser, y la solución de conflictos sintácticos mediante reglas y mecanismos de control de errores.

Este trabajo establece una base sólida para continuar con el desarrollo del compilador, permitiendo avanzar en las próximas etapas y agregar chequeos más avanzados.