2013

**INFORME DE DISEÑO DE COMPILADORES**

**Primer parte**

**GRUPO N 13**

**INTEGRANTES:**

* **Fernández, Sebastián (**[**sebachk90@gmail.com**](mailto:sebachk90@gmail.com)**)**
* **Gómez Ortiz, Ibrian (**[**ib.gomezo@gmail.com**](mailto:ib.gomezo@gmail.com)**)**
* **Pagano, Diego (**[**diegoipagano@gmail.com**](mailto:diegoipagano@gmail.com)**)**

**TEMAS ASIGNADOS**

El siguiente describirá y explicara la creación de analizador léxico como parte de un trabajo practico propuesto por la catedra de Diseño de Compiladores.  
Al grupo le fue asignado como actividad particular:

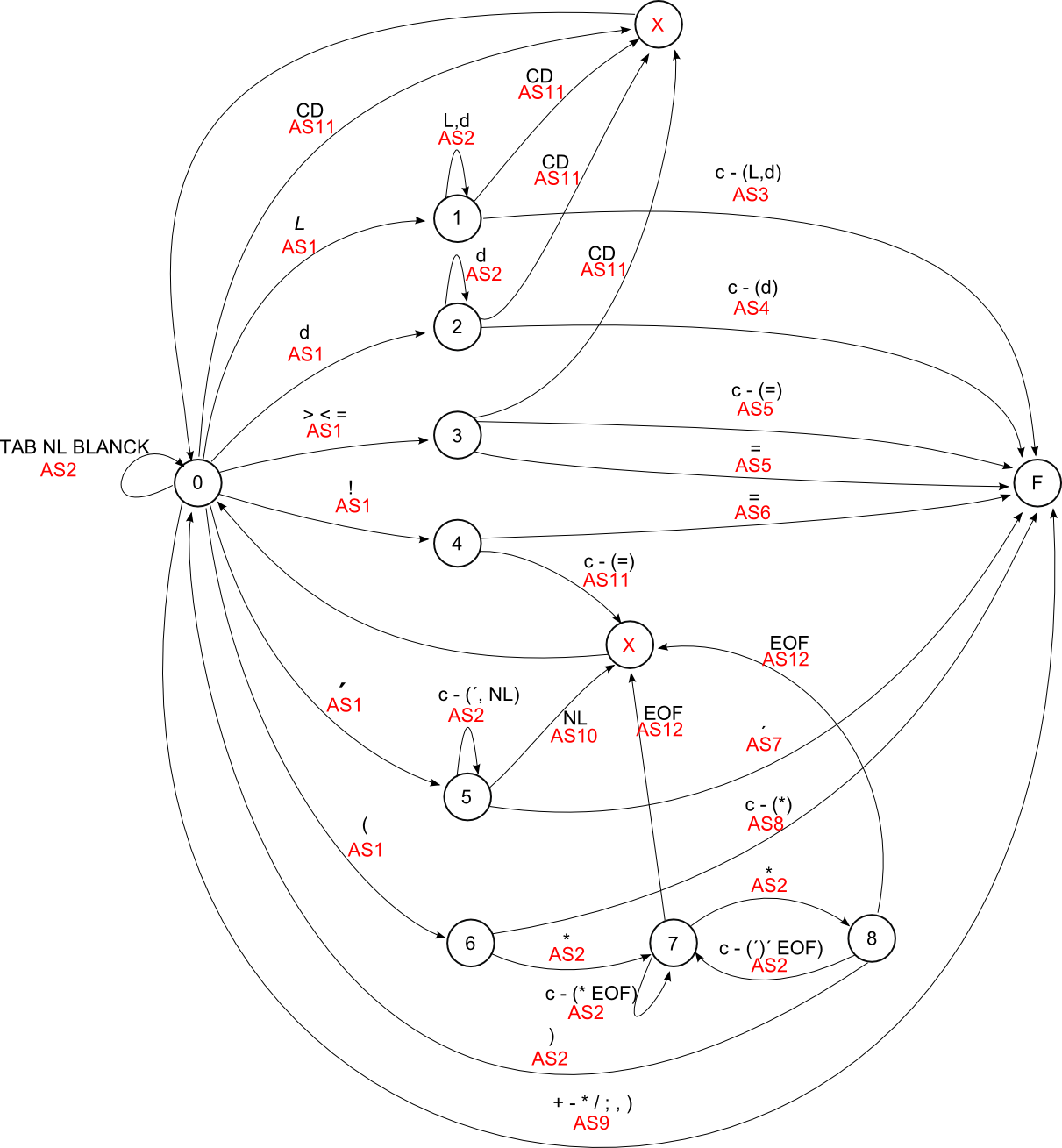
* funciones con pasajes de parámetro por referencia(4)
* Unsigned int
* Iteración de LOOP y UNTIL (1)
* Comentarios mixtilíneas
* Cadenas de una línea

**INTRODUCCION**  
**¿Qué es un analizador léxico?**   
Un analizador léxico (AL) es un componente de un compilador de lenguaje de programación, que se encarga de analizar y encontrar errores léxicos en un código fuente dado. El AL buscará palabras reservadas propias del lenguaje, además de detectar identificadores, constantes, comentarios, etc. A estos elementos se los denominan *Tokens.*

**¿Cómo funciona?**  
El AL funciona en conjunto con otro componente llamado analizador sintáctico (AS). En éste caso el sintáctico se desarrollada utilizando una herramienta ya implementada que se denomina YACC. El AS es quien pone en funcionamiento al procesador léxico solicitándole un token.  
El AL trabaja recorriendo un archivo de texto caracter a caracter procesándolos en un autómata finito. Cada vez que se arribe a un estado final el analizador habrá encontrado un token que deberá ser entregado al procesador sintáctico.  
Es importante aclarar que el AL se pone en funcionamiento solo cada vez que el AS le solicita un token y que una vez encontrado y entregado, se detiene hasta nuevo pedido.  
Cada token está compuesto por un número que lo identifica y un valor de posición que indica en que parte de la tabla de símbolos está almacenado.

**Desarrollo de la aplicación**Como primer medida el grupo comenzó por la creación del autómata que describe los estados por los que atraviesa el AL cada vez que lee un carácter y hasta encontrar un token. (Imagen 1)  
  
L = carácter letra  
d = carácter digito   
TAB = carácter de tabulación  
NL = carácter de Nueva Línea  
EOF = fin de archivo  
BLANK = carácter de espacio  
CD = carácter desconocido  
c – () = cualquier carácter menos los que se encuentran especificados entre paréntesis  
Cualquier otro símbolo o carácter se representa así mismo.

Los escritos en rojos hacen referencia a las Acciones Semánticas que se ejecutan en cada transición de estados.



*Imagen 1 correspondiente al autómata que representa el Analizador Léxico  
Los estados de error (X) son iguales pero por cuestiones de disposición espacial se dibujaron separados*

Matriz de transición de estados

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Carácter** | | | | | | | | | | | | | | | | | | | |
| **Est** | L | d | > | < | = | ! | ‘ | ( | ) | + | - | \* | / | ; | , | NL | TAB | BLANK | EOF | CD |
| 0 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 6 | F | F | F | F | F | F | F | 0 | 0 | 0 | F | **X** |
| 1 | 1 | 1 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | **X** |
| 2 | F | 2 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | **X** |
| 3 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | **X** |
| 4 | **X** | **X** | **X** | **X** | F | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | F | **X** | **X** | F | **X** |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | F | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | **X** | 5 | 5 | F | **X** |
| 6 | F | F | F | F | F | F | F | F | F | F | F | 7 | F | F | F | F | F | F | F | **X** |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 7 | 7 | 7 | 7 | 7 | 7 | **X** | 7 |
| 8 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 0 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | **X** | 7 |
| F |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Los estados marcados con **X** son estados de error.

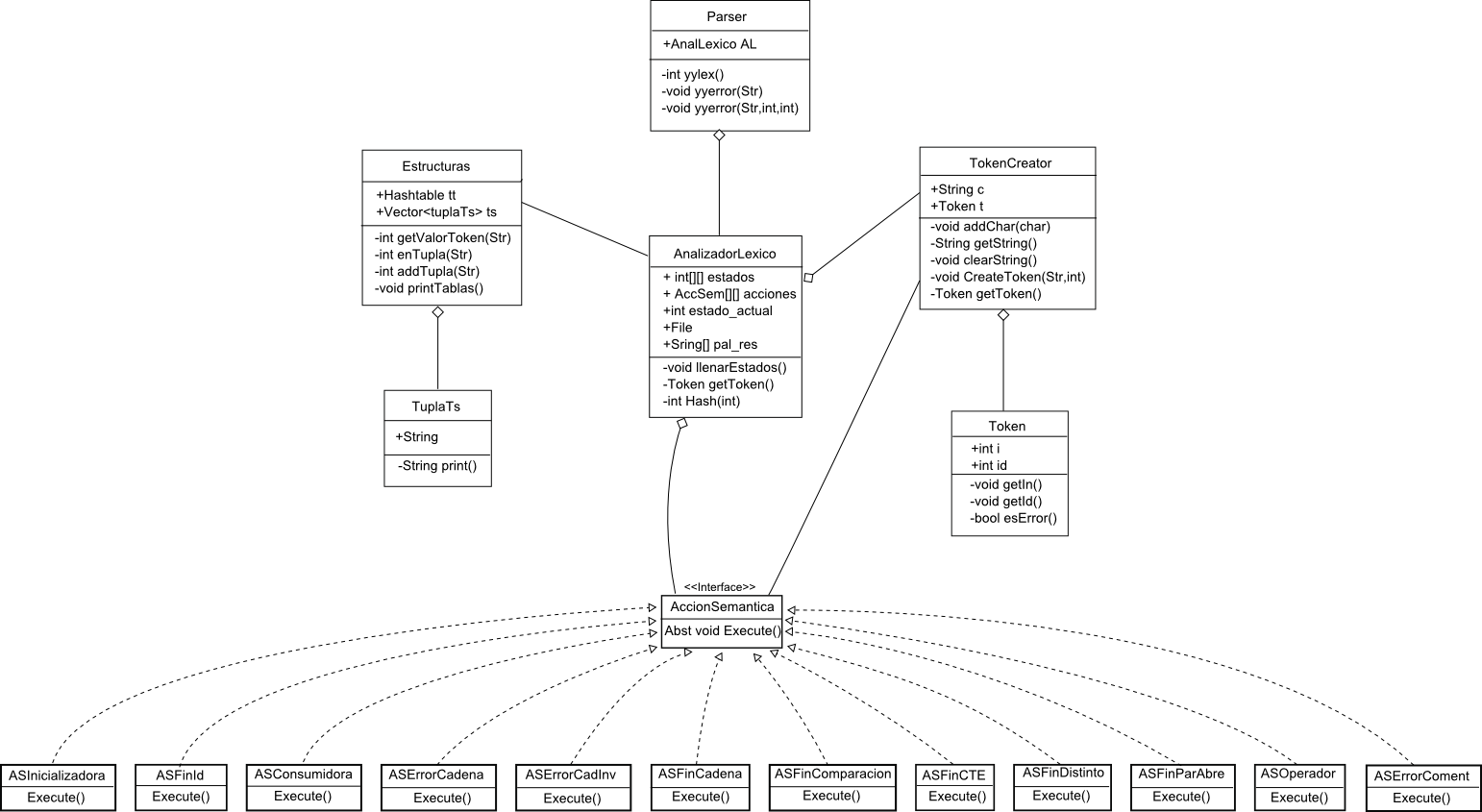
Matriz de Acciones semánticas

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Est** | **Carácter** | | | | | | | | | | | | | | | | | | | |
|  | L | d | > | < | = | ! | ‘ | ( | ) | + | - | \* | / | ; | , | NL | TAB | BLK | EOF | CD |
| 0 | AS 1 | AS 1 | AS 1 | AS 1 | AS 1 | AS 1 | AS 1 | AS 1 | AS 9 | AS 9 | AS 9 | AS 9 | AS 9 | AS 9 | AS 9 | AS 2 | AS 2 | AS 2 | AS 2 | AS 11 |
| 1 | AS 2 | AS 2 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 3 | AS 11 |
| 2 | AS 4 | AS 2 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 4 | AS 11 |
| 3 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 5 | AS 11 |
| 4 | AS 11 | AS 11 | AS 11 | AS 11 | AS 6 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 | AS 11 |
| 5 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 7 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 10 | AS 2 | AS 2 | AS 2 | AS 11 |
| 6 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 2 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 8 | AS 11 |
| 7 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | A S2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 12 | AS2 |
| 8 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 2 | AS 12 | AS2 |
| F |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Una vez delineado el autómata y las matrices que lo representan se comenzó con la implementación. El grupo eligió el lenguaje Java para desarrollarse.  
Se crearon las clases:

* Analizador Léxico: Encargada de recorrer un archivo de texto (código fuente) carácter a carácter y encontrar tokens y devolverlos. Una vez encontrado algún token se detiene hasta nuevo pedido.
* Estructuras: Contiene la tabla de símbolos (implementada como un vector<TuplaTs>) y la tabla de tokens (implementada con una hashtable). Contiene funcionalidades para agregar y consultar entradas en ambas tablas.
* TuplaTs: Implementa la tupla de la tabla de símbolos. Los atributos de ésta pueden variar
* TokenCreator: Se encarga de crear los tokens. Almacena los caracteres que se le van pasando y cuando se le solicita crea un token con los caracteres que venía almacenando
* Token: Clase que representa un token. Está compuesta de un entero que representa el número de token y un puntero a la posición de la tabla de símbolos donde se encuentra almacenado.
* Interface Acción Semántica: Contiene el comportamiento común de toda acción Semántica.
* ASInicializadora (AS1): Lee el primer carácter, marca el archivo en la posición donde lo leyó y lo almacena
* ASConsumidora (AS2): Consume un carácter y lo almacena.
* ASFinId (AS3): Detecta un carácter que no pertenece a la cadena, no lo consume y vuelve hacia atrás una posición en el archivo. Crea un token y además trunca el tamaño del token a 15 caracteres.
* ASFinCTE (AS4): Detecta un carácter que no pertenece a la constante, no lo consume y vuelve hacia atrás una posición en el archivo. Crea un token y controla que el valor de la constante no supere el límite permitido
* ASFinComparacion (AS5): Detecta si el/los caracter/es de comparación está/n bien escrito/s. Se ejecuta siempre después de haber leído un > < o =. Luego si el carácter que viene es un =, puede saber si se trata de un >=, <=, ==. Caso contrario vuelve una posición en el archivo.
* ASFinDistinto (AS6): Detecta el token != (distinto). Si lo encuentra lo crea.
* ASFinCadena (AS7): Detecta la comilla simple final y de esta manera reconoce un fin de cadena. La consume y crea el token correspondiente.
* ASFinParAbre (AS8): Si luego de leer un paréntesis que abre viene cualquier otro carácter que no es un asterisco estaremos entonces en el inicio de un nuevo token. Esta acción semántica retrocede una posición en la lectura del archivo y crea un token con el carácter “(”.
* ASOperador (AS9): Ésta acción detecta los operadores + - \* / ; , ) y crea el token correspondiente.
* ASErrorCadena (AS10): Detecta e informa el error provocado por cadenas que contienen en su interior el carácter de nueva línea.
* ASErrorCarInv (AS11): Detecta inversiones de caracteres. Se ejecuta luego de leer un ! cuando el analizador léxico espera un = y esto no ocurre.
* ASErrorComentario (AS12): Detecta el carácter de fin de archivo (EOF) en medio de un comentario.

La clase Parser y ParseVal surgen de ejecutar la aplicación YACC que recibe como parámetro la gramática del lenguaje creado por el grupo. La gramática se encuentra en el archivo Gramatica.y que se adjunta en ésta entrega.



**Problemas surgidos durante la implementación**  
A lo largo del trabajo el grupo se topó con problemas de distinta índole, siendo en su mayoría los generados por la gramática.  
Durante la generación del Parser, surgió un conflicto reduce/reduce dado que nos enfrentábamos a un problema con la lista de parámetros informada dentro de un llamado a función. Este problema se debe a que la lista de parámetros dentro de una función puede estar vacía, por lo que se debe añadir en la gramática la posibilidad de que esto ocurra. Se solucionó añadiendo a la declaración de la llamada a función, ID '('')', y de esta forma se aceptan tanto parámetros, lista de parámetros o ningún parámetro.  
Otro fue la aparición de un shift/reduce, provocado en las sentencias IF y ELSE dado que llegado a ese punto la gramática no especificaba que camino escoger. Se solucionó dando precedencia a una sentencia sobre la otra.

**Errores léxicos detectados por el compilador**

* Carácter invalido: Se da cuando espera un “=” luego de leer un “!” para intentar crear un token “!=”. Cuando el carácter no es un “=” se muestra el siguiente mensaje:  
  "Linea #: Caracter invalido, se esperaba '=' luego del '!'"
* Constante fuera de rango: cuando una constante posee un valor menor a 0 y mayor a 65535 se muestra el mensaje:  
  "Linea #: El entero se encuentra fuera del rango permitido"
* Cadena mixtilínea: ocurre cuando dentro de una cadena se encuentra el carácter “\n”. Se muestra el mensaje:  
  "Linea #: No se admiten cadenas mixtilínea"
* Error por fin de archivo: Sucede cuando se lee el carácter EOF dentro de un comentario. El mensaje que se muestra es:  
  "Linea # : Error EOF leído en medio de un comentario"
* Error de Carácter desconocido: Se da cuando el Analizador Léxico lee una carácter que no es reconocido por el lenguaje. Se muestra el siguiente mensaje:  
  "Linea #: El Caracter leído “Caracter”no es válido"
* Truncamiento de ID: ocurre cuando el tamaño de un identificador es mayor a 15 caracteres. El mensaje que se muestra es:  
  "Linea #: WARNING: El identificador tiene un nombre muy largo, será truncado"

**Errores sintáctico detectados por el compilador**Error que se produce cuando el parser recibe un token diferente al que está esperando según la gramática.  
Se informa el mensaje:  
“Token leído: # y se esperaba $”

**Lista de no terminales de la gramática**programa: no terminal de entrada a la gramática.  
lista\_declaraciones: Se refiere al listado de declaraciones del programa.  
Declaración: Es una declaración en el programa.  
bloque\_sent: Un bloque de sentencias (ya sean ejecutables o declarativas).  
lista\_sent: Lista de sentencias ejecutables.  
sentencia\_ejec : Sentencia ejecutable que acepta el programa.  
llamada\_funcion: Sentencia ejecutable de llamado a función.  
sentencia\_declarativa\_tipo: Sentencia declarativa de tipo de constante.  
lista\_var: Lista de variables.  
bloque\_IF : Bloque de un IF (ya sea con o sin ELSE).  
cond : Condición de un IF o LOOP-UNTIL.  
comparador : Operador de comparación.  
sentencia\_declar\_funcion: Sentencia declarativa de una función.  
parámetros: Parámetros, o bien de sentencia declarativa de función, o de llamado a una función.  
lista\_parametros : Lista de parámetros.  
parametro : Parámetro expresado como Expresión.  
bloque\_funcion : Lista de sentencias dentro de una función.  
lista\_sent\_declar: Lista de sentencias declarativas.  
asignacion : Asignación de una variable.  
expresion : Expresión matemática de suma o resta.  
termino : Término matemático de producto o división.  
factor : Reductor de constante o identificador.  
tipo : Tipo de variable.

**Tratamientos de errores**Para el tratamiento de errores, el grupo optó primero por el modo pánico. Esto quiere decir que para las sentencias simples, cuando se encuentra un error, se buscará el token ';' y se recuperará a partir de allí.  
Pero existen otras situaciones de posibles errores, donde el token ‘;’ no es útil para recuperarse rápidamente, por ejemplo:

*if(a<<b) then  
 begin   
 sentencia1;  
 sentencia2;  
end*

En éste caso va a recuperarse recién luego de la sentencia 1, lo que implica que también va a ignorar el *Begin*, por lo tanto va a encontrar otro error cuando lea *end.*

Luego de eso, el grupo empezó a buscar la manera de tratar los errores más complejos, empezando por la falta del ';'. Cuando falta este token al final de una sentencia, el analizador llega a un estado de error, pero no puede salvarse buscando el token ';' porque es el que falta, por lo tanto ignora todo lo que sigue luego del error hasta encontrarlo y esto genera varios problemas:

1. si hay error antes de la *sentencia2;*, ignora la *sentencia2*, y si ésta tenia error, no lo detecta
2. si se da justo antes de un *if* o un *loop* sucede lo mismo que antes , ignora todo el *if* o *loop*
3. si se da al final de un bloque de sentencias, ignora el end.

Básicamente trata como un error grande todo desde el momento que encuentra el error hasta el próximo ‘;’.

Fue ahí cuando nos inclinamos por trabajar analizando todos los casos de error por separado.  
Se pensó en separar las sentencias bien escritas de las mal escritas, y dentro de las que se encontraban con error especificar los diferentes casos.  
Por ejemplo en una sentencia *IF (if (condición) then <bloqueIF>)* se puede dar:  
 1) que falte la condición o que esté mal escrita, falta de algún paréntesis

2) que falte el *then*

En un bucle *loop (loop <bloqueLOOP> until (condicion)):*

1. que falte el *Until*
2. algún error en la condición

Por el lado de las sentencias, una correcta es o bien una incompleta con el ‘;’ o una con error y un ‘;’, eso es para evitar errores shift/reduce.

**Conclusión**  
El trabajo fue de mucha utilidad para la comprensión del funcionamiento de un compilador. Tanto la creación del Analizador léxico como el del Sintáctico fueron una experiencia clara que ayudaron al entendimiento del problema y comprensión del tratamiento de errores.

**2da Parte  
Código intermedio y Assembler**

**Introducción**

Para la segunda parte, se le asignó al grupo el desarrollo de las partes faltantes del compilador, referidas a la generación de código. Para la misma se pidió desarrollar una generación de código intermedio y a continuación desarrollar el generador de código Assembler MASM.

**Temas Asignados**

Al grupo se le asignaron los siguientes puntos específicos:  
\*Polaca inversa para el código intermedio  
\*Pasaje de parámetros por referencia.

**Código Intermedio**

La polaca inversa fue implementada con un arreglo, la misma contiene dentro, operandos (referencias a la tabla de símbolos), operadores, etiquetas (de funciones y de sentencias de control) y referencias a posiciones de la propia polaca.  
Para llenar la polaca se tuvieron en cuenta ciertas situaciones:  
\*Operadores y operando: se insertan directamente en la polaca, estas inserciones se realizan siempre al final, por lo tanto desde la gramática se tuvo especial cuidado para que las inserciones se hagan en el orden adecuado 🡪 a+b, primero se inserta ‘a’ luego ‘b’ luego ‘+’.  
Dentro de nuestra gramática tenemos la regla “factor: ID|CTE;”, es aquí cuando agregamos a la polaca el token leído ($1.ival, su posición en la tabla de símbolos).  
Gracias a la recursividad a izquierda, siempre vamos a encontrar que el primer operando llega a la regla mencionada antes que el segundo. Luego en las reglas referidas a operaciones, ya sea en operadores lógicos como aritméticos, se insertan en la polaca los operadores correspondientes. (\*1)  
\*Precedencia: gracias a que se tuvo en cuenta la precedencia de operadores desde un principio, no se necesito tomar recaudos para que se refleje correctamente la precedencia en la polaca.  
\*Asignación: “ID ‘=’ expresión”, llegados a este punto, la expresión esta correctamente insertada en la polaca, faltan insertar los token ID e ‘=’, ahora se debe insertar primero el ID y luego el ‘=’, por lo tanto insertamos, $1.ival e “=” en ese orden. (2\*)

(\*1) En la gramática se cambio la regla condición, antes era “cond: expresión comparador expresión” donde luego la regla del comparador era “comparador: <|>|<=|>=|==|!=”. Esto se cambió y reemplazó en condición para poder insertar correctamente en la polaca (ver gramática actual)  
(\*2) la asignación es un caso especial de la polaca, dado que para a=b por ejemplo, la polaca resultante es b|a|=, por lo tanto se debe tener en cuenta al hacer el código assembler.

\*Funciones: en la declaración de funciones, se agrega a la polaca la etiqueta de la función (en la regla se corresponde con $2.sval), además se almacena la posición de la misma en una estructura que será luego accedida para conocerla.

\*Call, Return y fin de funciones: para el llamado de funciones, dado el ID ($2.sval) se busca en la estructura antes mencionada la posición del arreglo donde se encuentra el label, y se agregan este número y el operador de CALL. En los return simplemente se añade el operador RETURN, que indicara el fin de la función. A su vez se añade un operador return al final de la función, haya otro antes o no. Esta decisión se basó en que la lógica necesitada para evitar el doble Return era compleja, y es inaceptable que no haya ninguno.

\*Parámetros: para el pasaje de parámetros, ya sea en llamados a función o en las declaraciones, se insertan en el orden que llegan dentro de la polaca, y hay que asegurarse que existan 2 etiquetas que los encierren (de manera de poder reconocer cuando comienzan y cuando terminan), para el caso de llamados una etiqueta es el operador CALL, y la otra es el puntero a la función (“PI(x)”). En la declaración, se utiliza como primera etiqueta el label de la función y para el final se agregó uno nuevo (“Func\_param”) como ayuda.

\*Sentencias de control: para las sentencias de control se procedió de la siguiente manera.  
*IF:* luego de insertar la operación de condición, se apila la posición actual de la polaca, se inserta un vacío (saltear la posición) y se añade el operador de salto por falso🡪 …|“<” | “ ”|#BF|…  
a continuación se procede normalmente, con las sentencias que siguen, luego al encontrar el ELSE se desapila el valor de la pila, se apila la posición actual, se deja un lugar vacío en la polaca y luego se inserta el operador de salto incondicional. Ahora se toma la posición actual de la polaca, y se la inserta, en la posición dada por el valor recién desapilado (indicando donde debe saltar el #BF).Finalmente se inserta una etiqueta, que sólo es necesaria para el Assembler.  
Cuando finaliza el bloque de sentencias del ELSE, y por ende finaliza el IF, se desapila el valor guardado anteriormente, y se utiliza como índice, para insertar la posición de la polaca, en el último lugar vacío, luego se coloca la etiqueta correspondiente.  
En caso que la sentencia IF no posea un bloque ELSE, al terminar el único bloque que hay, se procede igual que al finalizar el Bloque ELSE.  
*LOOP:* Al comenzar la sentencia loop, se carga en la polaca la etiqueta de control y se apila la posición donde comienza el mismo. Luego, al terminar de insertar la condición, se hace lo siguiente: obtengo el valor de la pila (desapilo), inserto en la polaca el valor a donde saltan en caso no cumplirse la condición (salir del loop), luego el operador de salto correspondiente (#BF), a continuación se inserta la posición donde comienza el bloque de sentencias loop (valor desapilado) seguido del salto incondicional. Finalmente se agrega la etiqueta correspondiente.

**Name Mangling**

Para el manejo de ámbitos, se utilizó un singleton que lleva registro de la entrada y salida de los ámbitos. Utiliza una estructura tipo pila que inserta ámbitos cuando se entra, y los desapila cuando se sale (para la salida no se necesita conocer de qué ámbito se sale, puesto que siempre será del último al que entro). Cuando se declara una función o una variable, la misma es insertada en la tabla de símbolos utilizando su ID ($x.sval) y el ámbito actual. Previamente se verifica que la variable (ID+ámbito) pueda ser declarada, en caso contrario esto da un error semántico.  
Posteriormente, cuando se utilizan variables o llamados a funciones, se verifica que haya sido declarada, chequeando con el ámbito completo primero, y progresando hacia ámbitos externos, llegando a la conclusión de que no existe si ya se pasó el ámbito raíz.

**Generación de código Assembler:**

Para realizar la generación del código Assembler, se crearon las clases Ensamblador, Manejador de Registros y las clases referentes al mecanismo de operaciones aritméticas. Las mismas se describirán a continuación:

\*Ensamblador: el ensamblador consta de una pila (que se rellena a partir de la lectura de la Polaca Inversa, y se utiliza para generar el código Assembler correspondiente a las diferentes operaciones, ya sean binarias o unarias), un manejador de registros al que se le delega la administración de los registros del assembler. La lógica utilizada en el ensamblador, es recorrer la polaca inversa (generada durante la ejecución del código intermedio), apilando aquellos elementos que sean operandos en la pila del ensamblador, hasta el momento de encontrar un operador, aquí resuelve la lógica de la operación (cuántos operandos necesita, cómo se refleja esto en el código assembler, etc.) utilizando los operadores aritméticos o bien los métodos propios del ensamblador.

\*Manejador de Registros: Es el encargado del seguimiento de los registros. Controla el estado (ocupado/libre), administra la lógica de utilización (liberar u ocupar un registro dado, o reasignar registros) comunicándole cambios al ensamblador para que los refleje en el código de salida.

\*Operaciones: Son las operaciones aritméticas involucradas en el assembler.   
Identifica el tipo de los operandos a procesar (cte., variables, parámetros o registros) y realiza el manejo correspondiente para generar las instrucciones assembler que resolverán correctamente la operación.

En lo referente a chequeos en tiempo de ejecución se añaden dos rótulos especiales: uno por operaciones que generan overflow (sumas y multiplicaciones) y otro en el caso de operaciones de valores negativos (resta). Para esto se agregan los Jumps (JO y JS) luego de las operaciones que podrían generar este tipo de errores.

**Sentencias Assembler:**

Dentro del assembler nos podemos encontrar grupos de sentencias referidos a simples operaciones:

Sumas:

a+b: En este caso primero se copia un operando en un registro, luego se realiza la operación entre el registro y el otro operando, finalmente se apilar el registro con el resultado (no se ve reflejado en assembler). Luego de la suma se hace el chequeo de overflow, saltando a la etiqueta correspondiente (“overflow”).

MOV R1, a  
ADD R1, b  
JO overflow  
…\*

a+b+c: en este caso, luego de hacer la suma entre a y b se guarda en un registro y después es usado para la próxima operación. Se hace el chequeo de overflow como en toda suma.

…\*  
ADD R1, c  
JO overflow   
  
Restas:

a-b: Como en toda operación binaria, se necesita pasar uno de los operando a un registro (assembler no permite operaciones entre 2 posiciones de memoria o 2 ctes, o una combinación de ambas), y luego se opera con este registro y el segundo operando. Luego de de la operación se hace el chequeo de signo, saltando a la etiqueta correspondiente (“signo”)

MOV R1, a  
SUB R1, b  
JS signo  
…\*  
Para el caso de operaciones entre un registro y otro operando (var/cte), se chequea cual de los 2 es el registro, dado que la resta no es conmutativa, no se puede ahorrar el uso de registros, es decir, si el registro es el segundo operando, entonces sí o sí se debe pasar el primer operando a otro registro y luego operar.  
ej: a-R1  
MOV R2, a  
SUB R2, R1 ; R1 queda liberado.

Multiplicación:

Este, al igual que la división -que se verá a continuación-, es un caso especial de operación binaria. La multiplicación exige que el primer operando este alojado en AX (R1) y luego se opera usando sólo el segundo operando. Tenemos 3 casos.  
1) el primero y más sencillo es tener el primer operando ya alojado en R1.  
R1 \*c:  
MUL c  
…  
  
2) cuando tenemos 2 operandos directos (var o cte) a \* b, se pasa el primero a R1:  
MOV R1 a  
MUL b  
…

3) cuando el registro está asignado al segundo operando a \* R1, dado que la multiplicación es conmutativa, se opera como en el caso uno, pero de manera inversa:  
MUL a  
…  
Luego de la multiplicación (cualquiera de los 3 casos) se agrega el chequeo de overflow, visto en la suma  
…  
JO overflow

División:  
En la división hay que tener ciertas consideraciones:  
\*el registro dx (R4) debe estar en 0 (no solamente debe no estar usado, sino que su valor deber ser 0)  
\*el dividendo está compuesto por dx:ax (por el rango uint, sólo es necesario ax, por eso dx debía ser 0), el cociente se almacena en ax y el resto en dx.  
\* para los 3 casos vistos en la multiplicación se procede de la siguiente manera; 1) se puede operar directo (AX / a); 2) se almacena al primer operando en AX y luego se opera (a/b); 3) se copia AX en algún registro libre, luego se pasa el primer operando a AX y se operan estos 2 registros (a/AX).  
El assembler queda de la siguiente manera:  
PUSH dx  
MOV dx, 0  
;caso1  
DIV a  
;caso 2  
MOV AX, a  
DIV b  
;caso3  
MOV BX, AX  
MOV AX, a  
DIV BX  
pop dx ;

Comparaciones:  
Es una operación binaria no conmutativa, la misma se comporta como la resta, luego se le agrega el salto correspondiente según el tipo de comparación.  
MOV R1, a  
CMP R1, b  
JNE labelX ; labelX está definida en otro lado  
  
Como en la polaca se usa el salto por falso, entonces este salto corresponde a la comparación a==b.  
  
Asignación:  
Operación binaria no conmutativa. Puede ser directa (a=b), o el resultado de alguna operación (a=b+c).   
En el primer caso son 2 simples movimientos con registros:  
MOV R1, b  
MOV a, R1  
En el segundo, la operación b+c, se guardará en algún registro, y este es el que se usa para asignar la variable.  
MOV a,R1.

Luego, existen sentencias más complejas, que utilizan combinaciones de las anteriores.

IF:   
El if se compone de una comparación y uno o 2 segmentos de operaciones (2 en caso de que sea “if then else”)  
Por lo tanto el assembler tendra el siguiente formato:  
if(a==b) then “bloquethen” else “bloquelse”  
MOV R1, a  
CMP R1, b  
JNE inicio-else  
bloquethen  
JMP finIF  
inicio-else:  
bloqueelse  
finIF:

LOOP:  
el loop se compone de una etiqueta de inicio, un grupo de operaciones y una comparación al final. Si la misma se cumple, entonces finalizan las iteraciones, caso contrario vuelve a la etiqueta inicial. Este tipo de iteración permite que siempre se ejecute al menos una vez el grupo de operaciones.  
loop sentencias\_loop until (a>b) quedaría en assembler:  
inicio\_loop:  
sentencias\_loop  
MOV R1, a  
CMP R1, b  
JBE inicio\_loop  
JMP fin\_loop  
fin\_loop:  
\*Aclaración: el salto incondicional puede omitirse dado que salta a la instrucción siguiente, sin embargo se dejo así para respetar el algoritmo de loop visto en clase.

Por último está el uso de parámetros por referencia.

Uso de parámetros en las funciones:  
como los parámetros son referencias a las variables reales, estas variables de parámetros tienen un tamaño de 4bytes (32 bits, un dirección de memoria), los mismos no se usan directamente dado que se estaría trabajando con direcciones y no con valores. Se debe buscar la manera de referirse a la variable en cuestión. Se utiliza un registro de 32 bits, para guardar el valor de la referencia (la dirección de la variable real) y luego se utiliza el operador “[]” para referirse a ‘*la variable que se encuentra en la posición almacenada en el registro X*’. Este uso se refiere a una variable, por lo tanto se debe usar como tal, es decir que por ejemplo no se puede utilizar para operaciones binarias que incluyan otra variable.  
Ejemplo: funtion f1 (int a)… a+b;  
el assembler quedaría algo como:  
MOV ebx, a  
MOV R1, [ebx]  
ADD R1, b  
Para las operaciones unarias de assembler (DIV y MUL) no es posible usar [ebx], porque la operación necesita conocer el tamaño del segundo operador, y [ebx] solo indica una variable en una dirección de memoria, sin especificar su tamaño. Para estos casos se envía el valor de [ebx] al propio registro de 16 bits (bx en este caso). Por lo tanto para esos casos nos encontramos con la sentencia:  
MOV bx, [ebx]  
MUL/DIV bx  
 **COMENTARIO:** el uso de R1 y R2 se utilizan para referirse a registros cualesquiera que se encuentren libres en el momento de usarlos, los mismos son obtenidos por el manejador de registros. Para los casos de AX y DX se considerar que se utilizan específicamente dichos registros, y en el caso de [ebx], puede ser sí otro registro, pero se muestra así porque queda más claro que eR1, o R1[32].

**Cambios realizados con respecto a la entrega anterior**

La mayoría se realizaron en la gramática, donde se añadió: el manejo de ámbitos, tanto para el nombramiento de variables y funciones, como para la verificación de errores semánticos (renombramiento o falta de declaración); y la generación de la polaca inversa con todo lo que eso conlleva.

Se modificó también la tabla de símbolos, se le agregaron nuevos campos (tipo, uso y referencia), lo cual llevo a modificar algunas acciones semánticas para completar los mismos en ciertos casos.

**Consideraciones relevantes**

Durante la realización del trabajo surgió la necesidad de realizar las siguientes consideraciones:

\*Distinción entre operaciones conmutativas de las que no lo son: Se necesitó lógica adicional ya que en este tipo de operaciones importa el orden de los factores.

\*Necesidad de tener ciertos registros disponibles: operaciones como la multiplicación y la división requieren de registros especiales, tanto para operar como para guardar el resultado y el resto respectivamente, por lo que se debió considerar la disponibilidad de los mismos antes de llevarlas a cabo.

\*Utilización de parámetros en llamados a funciones: dado que todos los llamados a funciones son llamados a subrutinas, se deben guardar los parámetros necesarios (mediante la instrucción “push”) en la pila de ejecución.   
Dado que a nuestro grupo se le asignó el pasaje por referencia, se debe añadir la palabra “offset” precediendo a la variable, lo que indica una referencia (dirección de memoria).

\*Parámetros en declaración de funciones: dentro de la declaración de la función se deben obtener los parámetros almacenados en la pila de ejecución (mediante la instrucción “pop”). Cabe aclarar que en el llamado a la función (en el CALL) se apila también la dirección de retorno, por lo que antes de obtener los parámetros es necesario desapilar esta dirección, luego los parámetros, y a continuación volver a apilarla.

**Conclusión**

A modo de conclusión es importante destacar que ésta etapa del proyecto tuvo más dificultades que las anteriores, en algunos casos se debió retocar trabajos que se daban por finalizado.  
La experiencia fue muy positiva y sirvió para adquirir conocimientos profundos en lo que concierne a la creación y entendimiento de los compiladores.  
Dado que al grupo se le asignó solo una porción de la creación de un compilador y éste demandó buena cantidad de tiempo y trabajo, los integrantes adquirieron una dimensión de lo complejo que puede llegar a ser realizar un compilador completo.  
Con el presente trabajo, los integrantes del grupo pudieron reivindicar la necesidad de la correlatividad de la materia con Lenguajes de programación pues el buen entendimiento de ciertos aspectos como el uso de parámetros por referencia facilitó la realización del compilador propuesto.