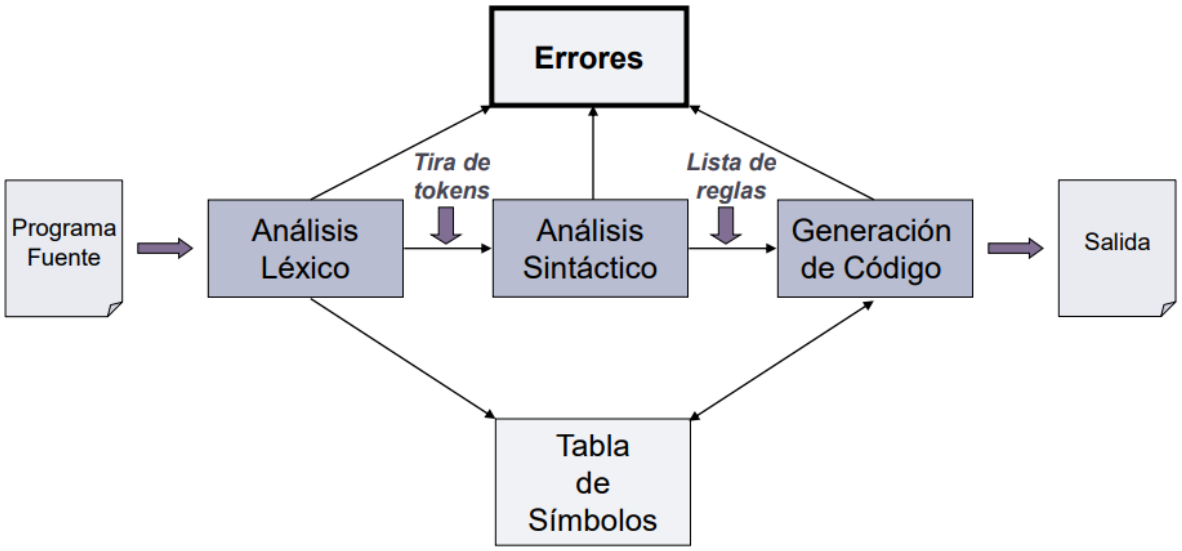
Trabajo Práctico Especial - Diseño de Compiladores

GRUPO 10



Integrantes:

- Tomas Elordi - telordi@alumnos.exa.unicen.edu.ar
- Felipe González Matarras - felipegnz00@gmail.com

- Eduardo Vibart - evibart@alumnos.exa.unicen.edu.ar

Consigna:

Desarrollar un Analizador Léxico que reconozca los siguientes tokens:

- Identificadores cuyos nombres pueden tener hasta 25 caracteres de longitud. El primero debe ser una letra, y el resto pueden ser letras, dígitos y "_". Los identificadores con longitud mayor serán truncados y esto se informará cómo Warning. Las letras utilizadas en los nombres de identificadores pueden ser minúsculas y/o mayúsculas.
- Constantes correspondientes al tema particular asignado a cada grupo.
  Nota: Para aquellos tipos de datos que pueden llevar signo, la distinción del uso del símbolo „-„ como operador aritmético o signo de una constante, se postergó hasta el trabajo práctico Nro. 2.
- Operadores aritméticos: "+", "-" , "*", "/" agregando lo que corresponda al tema particular.
- Operador de asignación: "=:"
- Comparadores: ">=", "<=", ">", "<", "=", "=!"
- "(", ")","{", "}", "," y ";"
- Cadenas de caracteres correspondientes al tema particular de cada grupo.
- Palabras reservadas (en minúsculas): if, then, else, end-if, out, fun, return, break
- y demás símbolos / tokens indicados en los temas particulares asignados al 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.

Temas particulares TPE 1 y 2

-**Enteros largos (32 bits):** Constantes enteras con valores entre –2^31 y 2^31 – 1. Se debe incorporar a la lista de palabras reservadas la palabra i 32. (Tema 5)

-**Punto Flotante de 32 bits:** Números reales con signo y parte exponencial. El exponente comienza con la letra F (mayúscula) y el signo es opcional. La ausencia de signo, implica un exponente positivo. La parte exponencial puede estar ausente.
Puede estar ausente la parte entera, o la parte decimal, pero no ambas. El "." es obligatorio.
Ejemplos válidos: 1. .6 -1.2 3.F–5 2.F+34 2.5F-1 15. 0. 1.2F10
Considerar el rango 1.17549435F-38 < x < 3.40282347F+38 U
-3.40282347F+38 < x < -1.17549435F-38 U 0.0
Se debe incorporar a la lista de palabras reservadas la palabra f32.
(Tema 7)

-Incorporar a la lista de palabras reservadas la palabra **when**.
(Tema 10)

-Cláusulas de compilación.
Incorporar, a las sentencias declarativas, la posibilidad de definir constantes, con la siguiente estructura:
const <lista _de_constantes> ;
donde la lista incluirá una lista de constantes separadas por ",". Cada constante se definirá con la siguiente sintaxis:
<nombre_de_constante>=:<valor_de_constante>
Ejemplo: **const** c1 =:10, x=:5;
Incorporar, a la sentencias ejecutables, la siguiente estructura:
when (<condicion>) then <bloque_de_sentencias>
donde:
<condicion> tendrá la misma estructura que las condiciones de las sentencias de control <bloque de sentencias> podrá contener tanto sentencias ejecutables como declarativas.
// LÉXICO: Incorporar a la lista de palabras reservadas, la palabra **const**.
(Tema 10)

-Incorporar a la lista de palabras reservadas las palabras **while**.
While con expresión (tema 12 en TP1)
while (<condicion>) : (<asignacion>) <bloque_sentencias_ejecutables>;
<condicion> tendrá la misma definición que la condición de las sentencias de selección. <bloque_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias break (se escribe con la palabra reservada seguida de ';')
Ejemplo:
while (i < 10) : (i =: i + 1) { out('ciclo'); };
(Tema 12)

-Incorporar a la lista de palabras reservadas las palabras **for y continue**.
for con continue

for (i =: n;<condicion_for> ; +/- j) <bloque_de_sentencias_ejecutables> ;
i debe ser una variable de tipo entero (1-2-3-4-5-6).
n y j serán constantes de tipo entero (1-2-3-4-5-6).
<condicion_for> será una comparación de i con m. Por ejemplo: i < m
Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4-5-6).
El signo + o - al final del encabezado es obligatorio.
<bloque_de_sentencias_ejecutables> Podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves. Además, podrá contener sentencias break y sentencias continue. Ambas se escriben con la palabra reservada seguida de ';'
(Tema 15)

-**Continue con etiquetado.** Incorporar, a la sentencia de control asignada (temas 11 al 16), la posibilidad de antecederla con una etiqueta. Incorporar, a la sentencia continue, la posibilidad de incluir una etiqueta.
Por ejemplo:

```
outer: while (i < 10) : (i =: i + 1) {
        while (j < 5) : (j := j + 1) {
                continue :outer;
        };
};
```

(Tema 18)

-**Sentencia de control como expresión.** Incorporar, a la sentencia de control asignada en los temas 11 a 16, la posibilidad de devolver un valor utilizando la sentencia break, o un valor por defecto en la finalización normal de la sentencia. (Tema 20)

-**Sin conversiones**: Se explicará y resolverá en trabajos prácticos 3 y 4. (Tema 23)

-**Comentarios de 1 línea**: Comentarios que comiencen con "<<" y terminen con el fin de línea. (Tema 24)

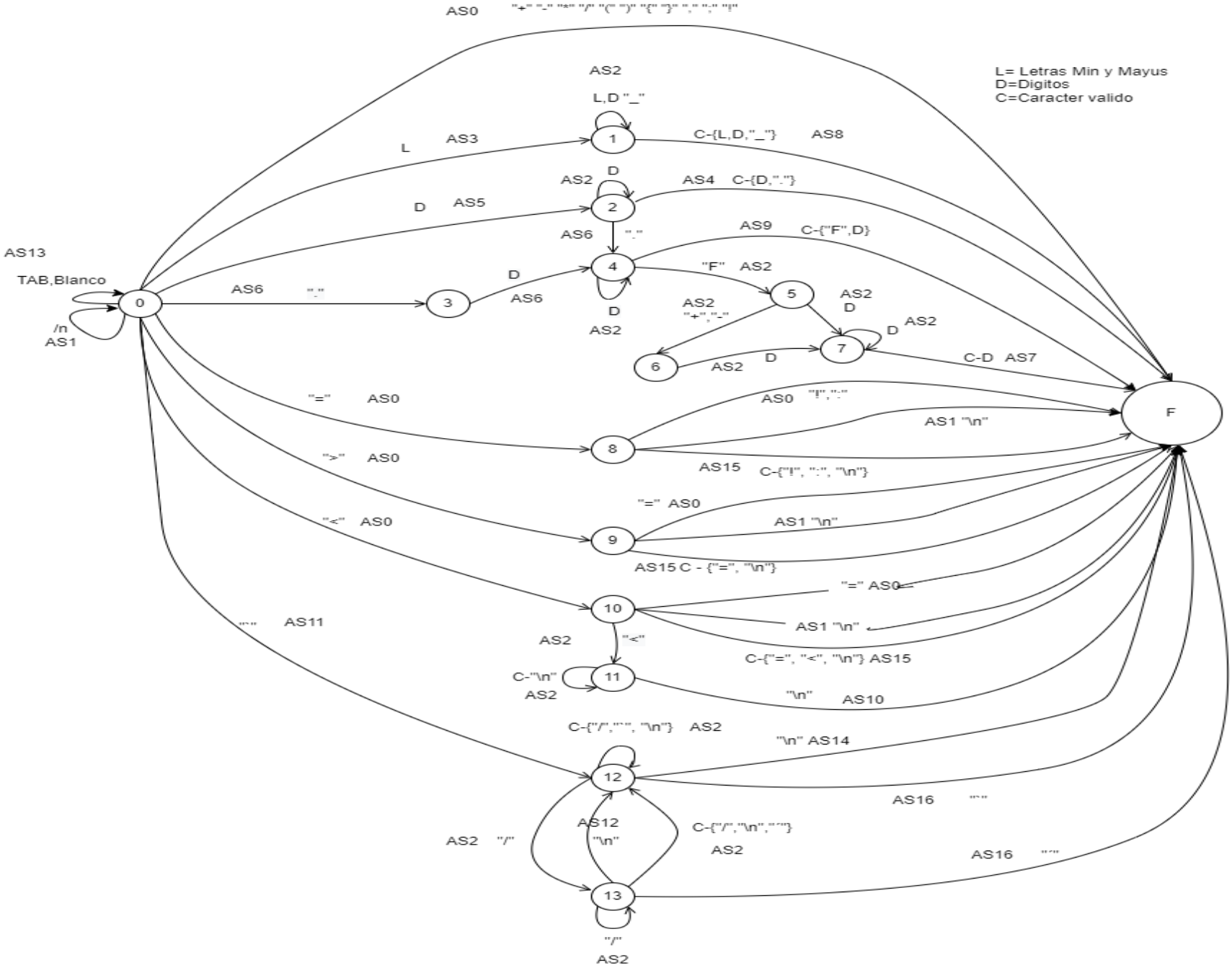
-**Cadenas multilínea**: Cadenas de caracteres que comiencen y terminen con " ' " . Estas cadenas pueden ocupar más de una línea, y en dicho caso, al final de cada línea, excepto la última debe aparecer una barra " / ". (En la Tabla de símbolos se guardará la cadena sin las barras, y sin los saltos de línea).
Ejemplo: '¡Hola /
mundo!'
(Tema 27)

Introducción

Para la realización del trabajo fue requerido el cumplimiento de la consigna mediante la implementación de un compilador para un lenguaje con características particulares por grupo.
En ésta primera parte fue requerida la implementación del analizador léxico (Trabajo Práctico número uno) así como también el analizador sintáctico (Trabajo Práctico número dos). Además de éstos dos componentes, son requeridos un programa fuente (entrada), la tabla de símbolos (la cual debe poder ser accedida por cualquier 'etapa' de la compilación) y el módulo de errores.
El lenguaje seleccionado para la implementación fue Java. Además de ésto se utilizó la herramienta YACC para la construcción del Paser o analizador sintáctico, especificada para Java.
Para la realización del analizador léxico se utilizaron dos matrices, la matriz de transición de estados, la cual describe el diagrama de transición de estados en forma autómata finito encargado de reconocer la gramática regular. Y la matriz de acciones semánticas, la cual describe la acción semántica a tomar dado un estado y un carácter entrante en concreto.

Matriz de transición de estados

Para facilitar la tarea de la creación de la matriz de transición de estado lo que se hizo fue realizar un diagrama que representa la transición entre los diferentes estados dependiendo y cada una de las entradas posibles, para después este diagrama escribirlo en forma de matriz en el programa.
Esta matriz es la que se utiliza para identificar los diferentes tokens en la entrada del usuario.
En las filas pueden verse los diferentes estados de la matriz la cual va del Estado 0(Estado Inicial) al Estado 13. Luego se tiene dos estados en los cuales se realiza una acción diferente, el Estado F que se refiere al estado Final y el Estado E indica que se está en un estado de error. En el caso de que se llegue a un estado de error en la salida se mostrará un error léxico con la línea en la que generó. Luego en las Columnas se pueden ver los diferentes caracteres permitidos por el lenguaje y a lo último una columna en la cual se detectarán todos los caracteres no permitidos


AS0 "+" "-" "*" "/" "(" ")" "{" "}" "," ";" "!"
L= Letras Min y Mayus
D=Digitos
C=Caracter valido
AS2
L,D "_"
L AS3
C-{L,D,"_"} AS8
D
AS2 AS4 C-{D,"."}
D AS5
AS6 "."
AS9 C-{"F",D}
AS13
TAB,Blanco
AS6 "."
D
AS6
"F" AS2
AS2
"+","-"
D
AS2
D
AS2
D
AS2
D
AS2
C-D AS7
/n
AS1
"=" AS0
AS0 "!",":"
AS1 "\n"
">" AS0
AS15 C-{"!", ":", "\n"}
"=" AS0
"<" AS0
AS1 "\n"
AS15 C - {"=", "\n"}
"=" AS0
"`" AS11
AS1 "\n"
AS2 "<"
C-{"=", "<", "\n"} AS15
C-"\n"
AS2
"\n" AS10
C-{"/","`", "\n"} AS2
"\n" AS14
AS16 "`"
AS12
"\n"
C-{"/","\n","`"}
AS2 "/"
AS2
AS16 "`"
"/"
AS2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
F

	bl	tab	/n	L	D	_	.	F	+	-	/	(	)	`	,	:	;	=	>	<	*	{	}	!	otro
0	0	0	0	1	2	F	3	1	F	F	F	F	F	12	F	F	F	8	9	10	F	F	F	E	E
1	F	F	F	1	1	1	F	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	E
2	F	F	F	F	2	F	4	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	E
3	E	E	E	E	4	E	4	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
4	F	F	F	F	4	F	F	5	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	E
5	E	E	E	E	7	E	E	E	6	6	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
6	E	E	E	E	7	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
7	F	F	F	F	7	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	E
8	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	E
9	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	E
10	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	11	F	F	F	F	E
11	11	11	F	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	E
12	12	12	F	12	12	12	12	12	12	12	13	12	12	F	12	12	12	12	12	12	12	12	12	12	E
13	12	13	12	12	12	12	12	12	12	12	13	12	12	F	12	12	12	12	12	12	12	12	12	12	E

Matriz de acciones semánticas

Cuando se reconoce que se cambia de estado se deben ejecutar determinadas acciones para cada uno de los diferentes estados, eso es lo que hace la matriz de acciones semánticas, dependiendo del estado en que se encuentre la matriz de transición de estados y el carácter de entrada se realiza una acción semántica específica. Se cuenta con 18 acciones semánticas para la realización correcta de la detección de tokens, en las cuales si tiene una acción de error será la encargada de mostrar un error en la salida en el caso que este ocurra.
A continuación se describe el listado de acciones semánticas indicando la acción que se lleva a cabo en c/u.

Lista de acciones semánticas

AS0 = Dependiendo el carácter define su id correspondiente.
AS1 = Suma una línea.
AS2 = No hace nada.
AS3 = Agregar al id del token el id del identificador.
AS4 = Se chequea el rango para los números enteros. Se agrega a la tabla de símbolos.
AS5 = Agrega el id de Dígito al token.
AS6 = Agrega el id de Decimal al token.
AS7 = Chequea el rango para los punto flotante, en el caso de no cumplir envía un warning. Se guarda el valor en la tabla de símbolos.
AS8 = Chequea longitud de los identificadores y busca en la tabla de palabras reservadas, en el caso que sea palabras reservadas agrega el id correspondiente al token. Se guarda el valor en la tabla de símbolos.
AS9 = Borra último carácter del token y resetea la entrada.Se agrega a la tabla de simbolos
AS10 = Vuelve al estado 0, suma una línea y vuelve al inicio del lexema.
AS11 = Setea el id en id de Cadena y borra el último carácter del token.
AS12 = Suma una línea y borra el último carácter del token.
AS13= borra el último carácter del token.
AS14= borra el último carácter y marca un error
AS15= borra el último carácter y vuelve la entrada un posición.
AS16 = Borra el último carácter y agrega la cadena a la tabla de símbolos.
ASE = Agrega el error al string que guarda los errores del programa.

	bl	tab	/n	L	D	_	.	F	+	-	/	(	)	`	,	:	;	=	>	<	*	{	}	!	otro
0	13	13	1	3	5	E	6	3	0	0	0	0	0	11	0	0	0	0	0	0	0	0	0	E	E
1	8	8	8	2	2	2	8	2	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	E
2	4	4	4	4	2	4	6	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	E
3	E	E	E	E	6	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
4	9	9	9	9	2	9	9	2	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	E
5	E	E	E	E	2	E	E	E	2	2	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
6	E	E	E	E	2	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
7	7	7	7	7	2	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	E
8	15	15	1	15	15	15	15	15	15	15	15	15	15	15	15	0	15	15	15	15	15	15	15	0	E
9	15	15	1	15	15	15	15	15	15	15	15	15	15	15	15	15	15	0	15	15	15	15	15	15	E
10	15	15	1	15	15	15	15	15	15	15	15	15	15	15	15	15	15	0	15	2	15	15	15	15	E
11	2	2	10	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	E
12	2	2	14	2	2	2	2	2	2	2	13	2	2	16	2	2	2	2	2	2	2	2	2	2	E
13	2	2	12	2	2	2	2	2	2	2	2	2	2	16	2	2	2	2	2	2	2	2	2	2	E

Tabla de Símbolos

La implementación de la tabla de símbolos se realizó utilizando un mapa, que su clave de acceso es un String y en el valor cuenta con una lista la cual va tener los diferentes atributos de la clave.
En esta etapa cada vez que detecta un identificador o una constante, esta debe ser agregada a la tabla de símbolos. En el caso de los identificadores lo único que se sabe, en esta etapa, es el lexema. Por lo que la clave para los identificadores será el lexema y contará con un atributo que es el ID del token. Luego para el caso de las constante la clave en la tabla de símbolos también será el lexema, pero como en esta etapa se sabe el tipo de que tiene la constante este se agrega a la en tabla de símbolos como atributo “Tipo”.

Decisiones de implementación

- El valor de los lexemas para las palabras reservadas fue definido en la clase abstracta AccionSemantica mediante constantes estáticas. Esta es la clase padre de todas las acciones semánticas que heredan e implementan el método ‘ejecutar’, distinto para cada acción semántica.
- La acción semántica (ASE) llama a la función addError del analizadorLexico que concatena en un String los textos producidos por los errores, guardando también la línea donde fue producido el error.
- La clase TPR (tabla de palabras reservadas) guarda en un hashmap las palabras reservadas con el valor de token que las representa.
- Se implementó la clase Token como contenedor del ID y el lexema.
- En la clase GeneradorMatrices se inicializa la matriz de acciones semánticas y la matriz de transición de estados a partir de los archivos AccionesSemanticas.txt y TransicionDeEstados.txt respectivamente.
- En la clase analizador léxico mientras que el estado actual sea distinto de final (-1) o error (-2) se toma el carácter, se ejecuta la acción semántica correspondiente y se obtiene el siguiente estado. Se utiliza la biblioteca Reader para leer el código y que el analizador léxico lo procese.
- La tabla de símbolos fue implementada como un HashMap anidado de la forma <String,<String,Object>> <lexema,<nombre_atributo, atributo>> con el objetivo de poder agregar nuevos atributos de distintos tipos.

Implementación de matriz de transición de estados

La matriz de transición de estados fue implementada como una matriz de enteros de Java a la que se le cargan los valores a partir de un archivo (el cual posee '-1' para indicar el estado final 'F' y '-2' para indicar errores 'E'). Luego esta matriz es accedida desde la clase AnalizadorLexico con el valor del estado actual y el valor del carácter que se esté procesando para obtener el estado siguiente.

Implementación de matriz de acciones semánticas

La matriz de acciones semánticas fue implementada como una matriz que guarda acciones semánticas definidas a partir de números guardados en un archivo. Por lo tanto, se leen números del archivo, se los traduce a una acción semántica y se guarda esta acción semántica en la matriz. Luego se accede a esta matriz en la clase AnalizadorLexico con el valor del carácter que se está procesando y el estado actual.

Errores Léxicos Considerados

- Caracteres inválidos (caracteres que no pueden aparecer en el estado actual del analizador léxico)
- Dígitos, ya sean enteros o reales, fuera de rango
- Nombres de variables de más de 25 caracteres

Descripción del proceso de desarrollo del Analizador Sintáctico

Para el desarrollo del Analizador Sintáctico, se utilizó la herramienta YACC, con lo que solo restó describir la gramática para poder generar el Parser.
Si bien YACC restó mucho trabajo, la descripción de la gramática no fue tan simple como se esperaba, dando en el camino diversos inconvenientes, tales como problemas de reducción y conflictos de precedencia (reduce/reduce o shift/reduce) los cuales conllevan un seguimiento de la gramática. Para poder solucionarlos se procedió a ejecutar el YACC con el 'debug me' activo para poder visualizar el seguimiento del parser y así localizar el momento del conflicto.
Para describir la gramática, se inició colocando las reglas básicas comprobando de menor a mayor complejidad su funcionamiento con una entrada definida por nosotros para comprobar diversas características requeridas. En el transcurso se presentaron errores arrastrados de la etapa anterior, como por ejemplo conflictos con la verificación de rangos en la acción semántica encargada de tal.

Una vez testeada su correctitud, se prosiguió a la adición de errores, permitiendo la característica de no finalizar la ejecución del compilador ante la ocurrencia de un error como tal. De esta forma, el Parser cuando reconoce un error lo informa y continúa con la ejecución.

No terminales usados

A continuación se listan los no terminales utilizados en la gramática.

- **program**: define al programa con su nombre y su bloque de sentencias.
- **bloque_sentencias**: define a una sentencia o un programa vacío.
- **sentencia**: define a una sentencia declarativa o ejecutable.
- **sentencia_declarativa**: define a una sentencia declarativa de datos, de funciones o una lista de constantes.
- **sentencia_decl_fun**: declara a una función con su ID, los parámetros y cuerpo.
- **nombre_program**: indica el nombre del programa con el terminal ID
- **list_var**: representa la declaración de una o más variables con su ID separadas por coma.
- **sentencia_decl_datos**: es una sentencia declarativa usada para declarar una o más listas de variables del mismo tipo.
- **parametro**: representa un parámetro formal de una función con su tipo y nombre.
- **cte**: representa las palabras clave I32 y F32 para una constante
- **factor**: representan números positivos, números negativos y nombres de variables.
- **termino**: utilizados para representar expresiones con mayor precedencia que las expresiones (división y multiplicación).
- **expresión**: representan las expresiones dentro de los return, de las asignaciones y de las condiciones. Suman y restan términos. También llamados a funciones, sentencias for y while
- **llamado_func**: define a la función con su ID y, entre paréntesis, su lista de parámetros.
- **retorno**: define al retorno de la función con el terminal RETURN y entre paréntesis una expresión.
- **asignación**: define la asignación de una expresión a una variable.
- **comparación**: son los terminales =, <, >, <=, >=.
- **condición**: utilizado para comparar dos expresiones con los operadores definidos en comparación.
- **sentencia_out**: define a la sentencia del out con los terminales out y, entre paréntesis, una cadena.
- **cuerpo_fun**: define recursivamente el cuerpo de la función con las sentencias de función.

- **sentencias_fun**: define todas las sentencias posibles que se pueden definir dentro de una función.
- **sentencia_if_fun**: sentencia if usable dentro de una función y con else opcional.
- **sentencia_when_fun**: sentencia when usable dentro de una función.
- **sentencia_while_fun**: sentencia while usable dentro de una función con la opción de utilizar una etiqueta al principio de la misma.
- **cuerpo_fun_break**: define recursivamente a las sentencias de una función que pueden contener break.
- **sentencias_fun_break**: define todos los tipos de sentencias de una función que pueden contener break.
- **sentencia_if_break_fun**: sentencia if dentro de una función que contiene break y puede contener else.
- **lista_asignacion**: define recursivamente una lista de asignaciones de expresiones a variables.
- **sentencia_ejecutable**: define todos los tipos de sentencias ejecutables.
- **sentencia_if**: sentencia if con un else opcional.
- **bloque_ejecutable**: define al bloque ejecutable recursivamente mediante sentencias ejecutables.
- **sentencia_when**: define la sentencia when.
- **sentencia_while**: sentencia while con la opción de utilizar una etiqueta al principio de la misma.
- **bloque_break_continue**: bloque de sentencias que se define recursivamente y que puede contener continue o break.
- **ejecutables_break_continue**: define todas las sentencias ejecutables que pueden contener break o continue.
- **tag**: se define como vacío o dos puntos y un ID.
- **sentencia_if_break**: sentencia if con un else opcional.
- **list_param_real**: define la lista de parámetros reales recursivamente, pudiendo los parámetros ser un ID de variable o una constante.
- **lista_const**: define una lista de asignaciones con la palabra clave const al principio.

Errores Sintáctico considerados

- Ausencia de ‘;’ en el bloque de sentencias, en el cuerpo de la función y en el bloque break continue.
- Error de declaración de datos.
- Error de declaración de función.
- Ausencia de ‘}’ y ‘{‘ en la declaración de program, de funciones, de sentencias if, en la sentencia when, en la sentencia while y en la sentencia for.

- Ausencia del tipo de la función.
- Ausencia de ‘(‘ y ‘)’ en la declaración de funciones, en la sentencia if, en la sentencia when, en la sentencia while, en la sentencia for, en la sentencia out y en el llamado a funciones.
- Ausencia de ‘:’ en la declaración de la función y en la sentencia while.
- Ausencia de declaración de parámetros en la función.
- Error en la sentencia if.
- Ausencia de ‘end_if’.
- Ausencia de ‘then’ en sentencias when e if.
- Ausencia de condición en la sentencia if, en la sentencia when, en la sentencia while y en la sentencia for.
- Error en sentencia when.
- Error en sentencia while.
- Error en sentencia for.
- Error en la declaración de constantes.
- Ausencia de ID en declaración de función.
- Error en la condición por falta de una expresión o de el no terminal comparación.
- Ausencia de cadena en sentencia OUT.
- Ausencia de asignación en sentencia while.
- Se tiene en cuenta para los enteros restarle uno al límite positivo.

Conclusión

En conclusión, en este trabajo práctico especial se pudieron profundizar las primeras etapas del compilador (Analizador lexico y Analizador sintactico) mediante el desarrollo de las mismas, siendo el Analizador Sintáctico el que “comanda” las órdenes de peticiones de tokens al Analizador Léxico y a medida que este último se los suministra, el sintáctico irá reconociendo mediante la gramática la entrada propiamente dicha.
Como reto en esta primera parte se incluye el labor a medida del aprendizaje, ya que a la hora de realizar el Analizador Léxico no se comprendía completamente el funcionamiento del sintáctico. Sin embargo con el pasar de las semanas, éste se introdujo en la cátedra y se comprendió la conexión entre ambos para el correcto funcionamiento del compilador.
Esto fue una experiencia óptima para practicar el desarrollo de un proyecto en equipo, utilizando herramientas como Git y Github para el manejo de versiones y de la herramienta Yacc para la generación del parser.