Trabajo Práctico
Diseño de Compiladores I
Etapa 1 y 2

N° Grupo: 18

Integrantes:

- Borgato, Franco - 249224 - francofederico.6v@gmail.com
- Costes, Juan Mateo - 249313 - mateocostes@hotmail.com

Profesor/a Asignado:

- Marcela Ridao

Temas Asignados:

- 4 8 9 14 17 19 21 25 26

Índice

Introducción

En el siguiente trabajo se desarrolla el analizador léxico y sintáctico de un compilador basado en las características de un lenguaje brindado por la cátedra, quedando pendiente el desarrollo del Generador de Código.

En este informe se describe el desarrollo de lo mencionado anteriormente, considerando las consignas asignadas, buscando ser claros y precisos a la hora de expresar lo que realizamos.

Temas particulares asignados

4. Enteros sin signo (16 bits): Constantes con valores entre 0 y 216 – 1.
Se debe incorporar a la lista de palabras reservadas la palabra ui**16**.

8. Dobles: Números reales con signo y parte exponencial. El exponente comienza con la letra D (mayúscula) y el signo es opcional. La ausencia de signo, implica exponente positivo. La parte exponencial puede estar ausente.
Puede estar ausente la parte entera, o la parte decimal, pero no ambas. El '.' es obligatorio.
Considerar el rango 2.2250738585072014D-308 < x < 1.7976931348623157D+308 U
-1.7976931348623157D+308 < x < -2.2250738585072014D-308 U 0.0

9. Incorporar a la lista de palabras reservadas la palabra **discard**.

gramática correspondiente: Incorporar a la gramática, la o las reglas que permitan reconocer invocaciones a funciones como si fueran procedimientos. Es decir: ID(<lista_de_parametros>); Si el compilador detecta una invocación como la indicada, debe informar Error Sintáctico. Sin embargo, si la invocación es precedida por la palabra reservada discard, la sentencia no generará error, y la invocación podrá considerarse como válida.

14. Incorporar a la lista de palabras reservadas las palabras **do** y **until**.
gramática correspondiente: **Do until con expresión**.
do <bloque_de_sentencias_ejecutables> **until** (<condicion>) : (<asignación>) ;
<condición> tendrá la misma definición que la condición de las sentencias de selección.
<bloque_de_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 „;")

17. Break con etiquetado.
Incorporar, a la sentencia de control asignada (temas 11 al 16), la posibilidad de antecederla con una etiqueta. Incorporar, a la sentencia break, la posibilidad de incluir una etiqueta.

19. Diferimiento
Incorporar, a las sentencias ejecutables, la posibilidad de antecederlas por la palabra reservada defer. Esta funcionalidad podrá aplicarse a estructuras sintácticas de una sola sentencia, como asignación o emisión de mensajes, pero también a estructuras de bloque como sentencias de selección o iteración.
Incorporar a la lista de palabras reservadas, la palabra defer.

21. Conversiones Explícitas: Se debe incorporar en todo lugar donde pueda aparecer una expresión, la posibilidad de utilizar la siguiente sintaxis:
tof64(<expresión>)
Incorporar a la lista de palabras reservadas, la palabra **tof64**.

25. Comentarios multilínea: Comentarios que comiencen con "**<<**" y terminen con "**>>**" (estos comentarios pueden ocupar más de una línea).

26. Cadenas de 1 línea: Cadenas de caracteres que comiencen y terminen con " **'** " (estas cadenas no pueden ocupar más de una línea).

Analizador Léxico

Decisiones de diseño e implementación

El proyecto fue desarrollado en el lenguaje Java en su versión 1.8, por ser un lenguaje utilizado en materias anteriores y en el cual nos sentimos cómodos trabajando, además de ser compatible con las herramientas brindadas por la cátedra.

La implementación parte de la base de que el analizador sintáctico debe invocar al analizador léxico cada vez que requiere un token, y este se lo entrega en cada solicitud, leyendo el código fuente.

Se comenzó a implementar el analizador léxico, creando la clase **AnalizadorLexico.** La misma, posee los atributos como **codigoFuente** (posee el código fuente), **caracter** (carácter que se está leyendo del código fuente), **cursor** (es el índice del carácter que se está leyendo del código fuente), **linea** (línea que se está leyendo del código fuente), distintos atributos para la construcción e identificación de los diferentes **Tokens** (por ejemplo un atributo para identificar los saltos de línea) y además, posee atributos donde se instancian y crean las clases **MatricesTransicion**, **TablaPalabrasReservadas** y **TablaSimbolos** que posteriormente en este informe se narran sus funciones. Por consiguiente, la clase presenta los distintos métodos:

- getToken: Este método representa el diagrama de transición de estados mostrado posteriormente en este informe. Se utiliza para identificar un token y presentarlo como un número entero. Primero, se identifica el tipo carácter y siempre que sea distinto de "$" se llama al método de **getValorSimbolo** obteniendo el valor del carácter que representa la columna en las matrices. Luego se hace uso de la matriz de transición de acciones semánticas, identificando la acción correspondiente y llamando al método ejecutar, de la accion correspondiente, retornando un token de tipo **Token** (para esto se utilizó la clase abstracta **AccionesSemanticas**, que luego será descrita). También se hace uso de la matriz de acciones estados, la cual nos indica cuál es el próximo estado.
- getValorSimbolo: Se le pasa como parámetro un carácter y retorna un valor asociado a la columna que el mismo representa en las tablas mostradas posteriormente en el informe. (Por ejemplo, detecta el carácter "+" y se retorna el valor 7, que es la columna de ese carácter en nuestra representación). Este método hace uso de la función **tipoCaracter**, para ayudar a identificar el carácter.
- tipoCaracter: Se le pasa como parámetro un carácter y retorna distintos valores constantes si el carácter es un dígito, una letra minúscula o una letra mayúscula. En caso contrario retorna el mismo carácter.

La clase, se instancia en el Main enviándole como parámetro el código fuente leído en la clase **ManejadorArchivo**.

A continuación, se explica en el informe lo que realiza cada clase en el orden que fue mencionada anteriormente. La clase **MatricesTransicion** se desarrolla en el apartado *"Mecanismo empleado para implementar las matrices".*

La clase **TablaPalabrasReservadas** se utiliza, como su nombre menciona, para almacenar las palabras reservadas, para lo cual se utilizó una estructura de tipo mapa. El mapa, llamado símbolos contiene un string el cual es la clave y un entero el cual lo representa numéricamente. En la

construcción de la clase se llama a un método el cual asigna a símbolos, las palabras reservadas junto con su valor, de forma manual comenzando por el número 261, al ser el último disponible. La clase también contiene un método obtenerId al cual le pasas como parámetro una clave string y retorna su identificador numérico.

La clase **TablaSimbolos**, al ser una estructura dinámica, se construyó como un mapa de mapas. El cual contiene un identificador que se incrementa de manera automática y dentro de él un mapa <string, string> el cual contiene los atributos de un determinado símbolo. Esta clase contiene métodos de alta, modificación e impresión del mapa mencionado anteriormente.

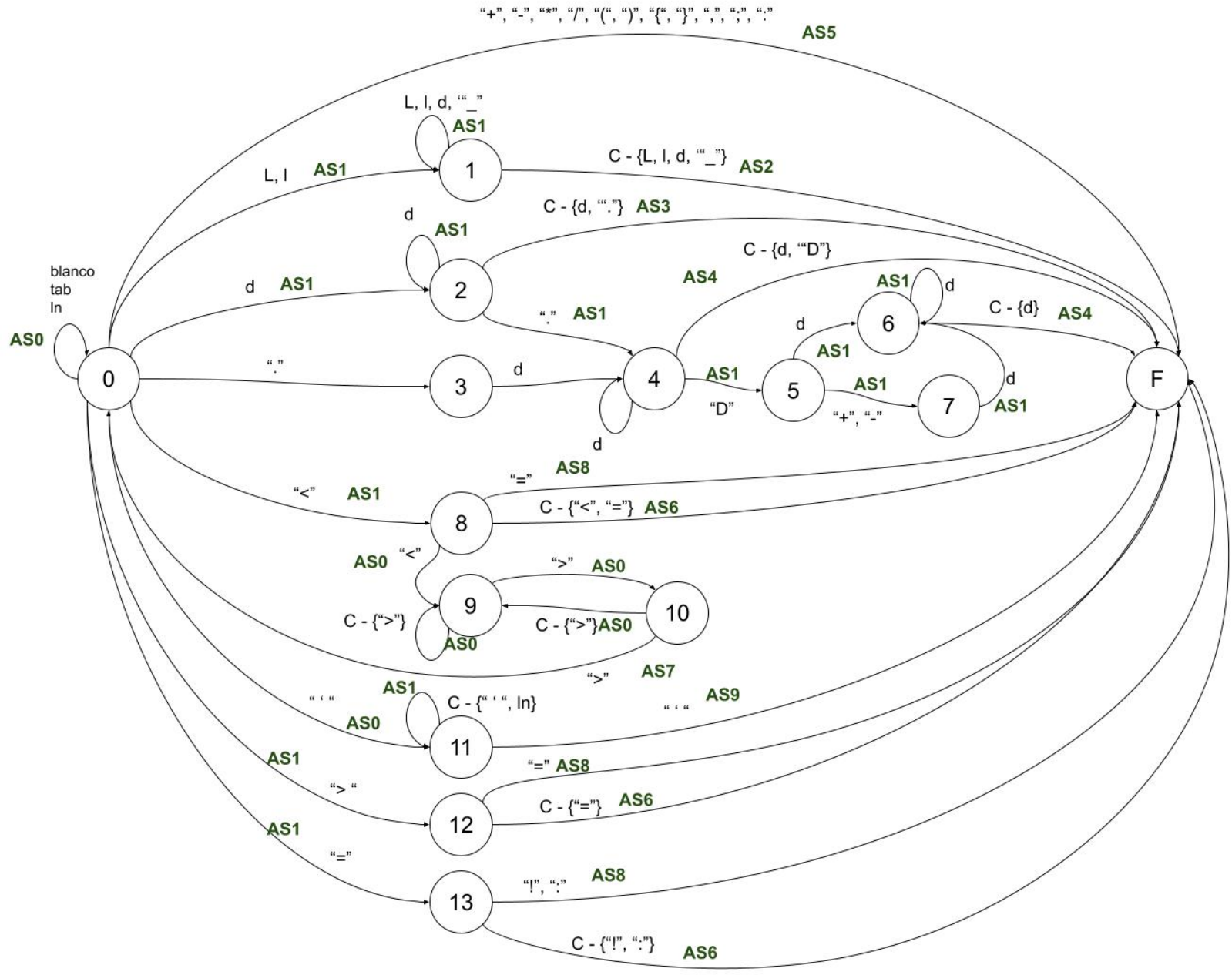
La clase **Token**, se utiliza para representar al token. La misma, contiene el atributo el cual representa numéricamente al token y un lexema. La clase contiene los métodos de consulta para cada atributo.

La clase **AccionesSemanticas**, se desarrolla en la sección “Acciones semánticas”.

La clase **ManejadorArchivo**, se utiliza para leer el código fuente, enviado a través de una dirección y almacenarlo en un StringBuilder, agregando para representar el final del código el símbolo “$”. Nos pareció que esta forma era más practica que ir leyendo los caracteres del código fuente y generando tokens.

Por último, se creó una clase Main en la cual se importó una herramienta de Java para abrir una pantalla y así seleccionar un archivo.txt en el cual se almacena el código fuente. Una vez seleccionado el archivo, se pasa como parámetro la dirección a la clase ManejadorArchivo, contruyendo el StringBuilder del código. Luego se crea el analizador léxico con el StringBuilder y un parser, que representa al analizador sintáctico, pasándole como parámetro el analizador léxico, para luego llamar al método parser.run de la clase Parser, que más adelante será explicado en este informe. Finalmente, como se pedía en el enunciado, se imprimen los tokens almacenados junto con sus errores, las estructuras sintácticas almacenadas, junto con sus errores y la tabla de símbolos.

Diagrama de transición de estados



Matrices de transición de estados

E\S	Bl	Tab	Ln	L	l	d	"_"	"+"	"*"	"/"	"("	")"	"{"	"}"	","	";"	"<"	">"	"="	"!"	" ' "	"."	"D"	" - "	":"	Otro
0	0	0	0	1	1	2	1	F	F	F	F	F	F	F	F	F	8	12	13	E	11	3	1	F	F	E
1	F	F	F	1	1	1	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	1	F	F	E
2	F	F	F	F	F	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	4	F	F	F	F
3	E	E	E	E	E	4	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
4	F	F	F	F	F	4	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	5	F	F	F
5	E	E	E	E	E	6	E	7	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	7	E	E
6	F	F	F	F	F	6	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
7	F	F	F	F	F	6	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	9	F	F	F	F	F	F	F	F	F
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	10	9	9	9	9	9	9	9	9
10	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	0	9	9	9	9	9	9	9	9
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	F	11	11	11	11	11
12	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
13	F	F	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\S	Bl	Tab	Ln	L	l	d	"_"	"+"	"*"	"/"	"("	")"	"{"	"}"	","	";"	"<"	">"	"="	"!"	" ' "	"."	"D"	" - "	":"	Otro
0	AS0	AS0	AS0	AS1	AS1	AS1	ASE0	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS5	AS1	AS1	AS1	ASE0	AS0	AS1	AS1	AS5	AS5	ASE1
1	AS2	AS2	AS2	AS1	AS1	AS1	AS1	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS2	AS1	AS2	AS2	ASE1
2	AS3	AS3	AS3	AS3	AS3	AS1	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS3	AS1	AS3	AS3	AS3	AS3
3	ASE2	ASE2	ASE2	ASE2	ASE2	AS1	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2	ASE2
4	AS4	AS4	AS4	AS4	AS4	AS1	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS1	AS4	AS4	AS4
5	ASE3	ASE3	ASE3	ASE3	ASE3	AS1	ASE3	AS1	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	AS1	ASE3	ASE3
6	AS4	AS4	AS4	AS4	AS4	AS1	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4	AS4
7	ASE3	ASE3	ASE3	ASE3	ASE3	AS1	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3	ASE3
8	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS0	AS6	AS8	AS6	AS6	AS6	AS6	AS6	AS6	AS6
9	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0
10	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS7	AS0	AS0	AS0	AS0	AS0	AS0	AS0	AS0
11	AS1	AS1	ASE4	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS1	AS9	AS1	AS1	AS1	AS1	AS1
12	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS8	AS6	AS6	AS6	AS6	AS6	AS6	AS6
13	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS6	AS8	AS6	AS6	AS6	AS6	AS8	AS6

Mecanismo empleado para implementar las matrices

Para la construcción de las matrices se utilizó la clase MatricesTransicion. La misma contiene dos atributos que identifican la cantidad de estados y la cantidad de símbolos, que representan las filas y columnas en nuestras matrices; y, además, contiene dos atributos para la declaración y construcción de las matrices de estados y de acciones. En el método constructor de la clase se invoca a dos metodos setMatrizEstados, en el cual se almacenan los estados de la tabla mostrada anteriormente, un -1 si es un estado final y -2 si es estado error, y setMatrizAcciones, en el cual se crean las distintas clases de acciones semánticas para ser almacenadas en la matriz como también se presentó anteriormente. La clase, también contiene los metodos getEstado, el cual retorna el valor del estado en una fila y columna determinada y getAcciones, el cual retorna una acción en una fila y columna determinada.

Acciones semánticas

A continuación se listan las acciones semánticas asociadas a las transiciones del automata del analizador Léxico, con una breve descripción de cada una.

AS0: Se encarga únicamente de leer el siguiente token en el código del programa.

AS1: Se lee el siguiente carácter y lo concatena.

AS2: Destinada principalmente al enviar tokens de identificadores y palabras reservadas. Revisa si lo leído hasta el momento un identificador o palabra reservada.

Si es ID, verifica su longitud y en caso de superar lo pedido, se acorta y se envía un mensaje de error. También se lo agrega a la tabla de símbolos y devuelve su identificación. Si es una palabra reservada, solo devuelve el identificador de token de la misma.

AS3: Agrega el valor de la constante entera leída a la tabla de símbolos si no está en ella, también verifica el rango entero y devuelve el identificador del Token CTE_INT.

AS4: Agrega el valor de la constante flotante leída a la tabla de símbolos si no está en ella, también verifica el rango correspondiente y devuelve el identificador del Token CTE_DBL.

AS5: Se encarga de leer los símbolos: '+' '-' '*' '/' '(' ')' '{' '}' ',' ';' ':' , reconocer el literal, y devolver Token del mismo.

AS6: Se encarga de leer el símbolo, retrocediendo el cursor para no perder el carácter encontrado actualmente. (Por ejemplo, devuelve el símbolo '<').

AS7: Esta acción está destinada a los comentarios. Suprime el token leído hasta el momento, y lee el siguiente carácter. El comentario es ignorado.

AS8: Se encarga de identificar operadores de dos caracteres, retornando su token correspondiente.

AS9: Se encarga de identificar cadena caracteres, retornando su token correspondiente.

ASE0: Se encarga de reconocer los caracteres que no llegan a ningún estado. Se notifica del error y se agrega el error a la lista de errores.

ASE1: Se encarga de reconocer los caracteres no reconocido por la gramática. Se notifica del error y se agrega el error a la lista de errores.

ASE2: Se encarga de la construcción del double en caso de que no venga un digito. Se notifica del error y se agrega el error a la lista de errores.

ASE3: Mala construcción parte exponencial.

Se encarga de la construcción de la parte exponencial del double en caso de que no venga un digito, '+' o '-'. Se notifica del error y se agrega el error a la lista de errores.

ASE4: Se encarga de la construcción de la cadena en caso que venga un salgo de línea y la misma no se haya cerrado. Se notifica del error y se agrega el error a la lista de errores.

Errores léxicos considerados

Se analizaron ciertos errores que pueden surgir cuando se está intentando leer un token. Se consideró el truncamiento de las constantes flotantes y enteras, si se encuentran fuera del rango mencionado en el enunciado, y el truncamiento de los identificadores si superan los 25 caracteres. Además, se crearon métodos de acciones semánticas que distinguen errores en la construcción de los tokens (ASE0, ASE1, ASE2, ASE3 y ASE4) mencionados anteriormente.

Analizador Sintáctico

Descripción del proceso de desarrollo

Se comenzó con esta segunda etapa, describiendo las reglas gramaticales correspondientes con lo solicitado en las consignas generales y particulares a cada grupo. Las mismas fueron revisadas y corregidas varias veces para lograr el resultado pedido. En el proceso, se buscó simplificar y desglosar lo máximo posible las descripciones de las mismas para una mayor comprensión, una posible futura corrección y evitar conflictos shift-reduce y/o reduce-reduce. También se trató de describir paso a paso cada regla y compilar la gramática, en orden a lo que era solicitado en el enunciado.
Por otra parte, también se describieron los diferentes errores sintácticos que podrían detectarse por el compilador. Para esto se contemplaron diferentes casos tales como: falta de comas, paréntesis, número máximo de parámetros posibles, variables fuera de rango, entre muchos otros.
A la hora de utilizar la herramienta yacc para java, la misma generó los archivos Parser.java y ParserVal.java, por lo que en nuestra clase Main, tuvimos que instanciar una variable de tipo Parser a la cual le pasamos el analizadorLexico, previamente instanciado con el código fuente a analizar.
Para finalizar se plantearon varios ejemplos de código fuente para testear nuestra descripción de la gramática, junto con la descripción de errores.

Problemas surgidos y solución de conflictos

Algunos inconvenientes que surgieron en el desarrollo fueron debido a los anidamientos y a las pequeñas diferencias entre algunas de las descripciones, por ejemplo, los parámetros presentes en la declaración de una función (tipo id), y los empleados en una invocación a una función (id), para este y otros casos se tuvo que replantear las descripciones y separarlas. Además de tener cierta imaginación y suspicacia a la hora de contemplar los casos de error.
Avanzando con la fase de testeo de la gramática, ante la obtención de resultados inesperados a la hora de compilar los casos de prueba, se detectaron errores tanto en la tabla de transición de estados como en la tabla de acciones semánticas de la etapa 1, correspondiente al analizador léxico, que en su momento no se observaron y que finalmente fueron corregidos.

Lista de no terminales

Programa: se encarga de controlar que el código inicie con un identificador de programa seguido por un conjunto de sentencias.
conjunto_sentencias: chequea que el bloque de sentencias dentro del programa esté dentro de una llave de apertura y otra de cierre.
sentencias: Acepta sentencias de tipo declarativas o ejecutables, cualquiera sea el orden.
declarativas: Identifica una sentencia o bloque de sentencias de tipo declarativas.
ejecutables: Identifica una sentencia o bloque de sentencias de tipo ejecutables.
declarativa: Determina la sintaxis de una sentencia declarativa.
tipo: describe los dos tipos solicitados, ui16 para los enteros y f64 para los flotantes.
lista_de_variables: Determina que en una declaración de variables las mismas corresponden a identificadores separados por una ‘,’.
funcion: Plantea la sintaxis con la que se declara una función.
lista_parametros: Determina que los parámetros que se le pasan a una función se separan con ‘,’.
parametros: Se utiliza para controlar que el máximo de parámetros de una función sea de dos.
parametro: Define la conformación de un parámetro como un tipo seguido de un identificador.
cuerpo_funcion: Describe la conformación de un bloque dentro de una función como un bloque de sentencias, tanto declarativas como ejecutables, y un retorno o simplemente un retorno.
retorno: Controla la descripción correcta de la sentencia de retorno. La misma debe comenzar con la palabra reservada return y finalizar con ‘;’.
ejecucion_retorno: Determina que es lo que puede retornar una función, siendo posibles opciones una condición o una expresión entre paréntesis.
condicion: Establece la conformación de la condición como dos expresiones separadas por un comparador y todo lo mencionado entre un paréntesis de apertura y cierre.
expresion: Plantea a la expresión como un término, una expresión más o menos un término o una conversión explícita tof64.
termino: Plantea al término como un término multiplicado o dividido por un factor o simplemente como un factor.
factor: Puede ser una constante tanto entera como flotante, un identificador o una invocación a una función.
invocacion: Establece cómo se conforma una invocación de una función.
lista_parametros_reales: Determina que los parámetros utilizados en la invocación de una función se separan con ‘,’, además también establece que pueden no emplearse parámetros.
parametros_reales: Se utiliza para controlar, al igual que en las declaraciones, que en las invocaciones a funciones se aceptan un máximo de 2 parámetros.
parametro_real: Plantea que un parámetro utilizado en una invocación puede ser una constante entera o flotante o un identificador.

comparador: Establece las distintas opciones de comparadores aceptados por la gramática. Estos son: ‘<=’; ‘>=’; ‘=’; ‘<’; ‘>’; ‘=!’.
ejecutable: Clasifica a las sentencias ejecutables como ejecutable_comun o ejecutable_defer.
ejecutable_comun: Describe a las sentencias ejecutables, estas pueden ser: asignacion; seleccion; mensaje_pantalla; invocacion_discard o expresion_dountil.
ejecutable_defer: Describe la sintaxis de las sentencias ejecutables de diferimiento las cuales son una ejecutable_comun precedida por la palabra reservada defer.

asignacion: Determina la construcción de las asignaciones, las mismas cuentan con un identificador seguido del símbolo ‘=:’ y una expresión, finalizando con ‘;’.
seleccion: Establece la construcción de las selecciones. La conformación es la siguiente: if condicion then '{' bloque_de_sent_ejecutables '}' else '{' bloque_de_sent_ejecutables '}' end_if ';'
o bien if condicion then '{' bloque_de_sent_ejecutables '}' end_if ';'.
bloque_de_sent_ejecutables: Define el bloque de sentencias empleado en las selecciones, las cuales son todas ejecutables.
mensaje_pantalla: Describe la estructura de la sentencia para mostrar un mensaje por pantalla. La misma consiste de la palabra reservada out seguida de una cadena cerrada entre paréntesis, la cual puede ser vacía.
invocacion_discard: Describe la invocación de una función discard, la cual es igual a la invocación de una función normal pero precedida por la palabra reservada discard.
expresion_dountil: Define la sintaxis de una expresión dountil, la cual tiene la siguiente conformación: do '{' bloque_de_sentencias_ejecutables '}' until condicion ':' asignacion_do_until ';' o también se le puede añadir una etiqueta, resultando de la siguiente forma: etiqueta ':' do'{' bloque_de_sentencias_ejecutables_etiqueta '}' until condicion ':' asignacion_do_until ';'.
asignacion_do_until: Determina la conformación de la asignación utilizada en la expresión do until, la misma posee la misma descripción que una asignación común, pero agrega un paréntesis de apertura y de cierre a la misma.
etiqueta: Consiste en un identificador empleado por las expresiones do until con etiquetado.
bloque_de_sentencias_ejecutables: Describe un bloque de sentencias ejecutables que puede estar seguido de la palabra reservada break y un ‘;’ o simplemente break y ‘;’.
bloque_de_sentencias_ejecutables_etiqueta: Posee la misma descripcion que el caso anterior, agregando una etiqueta precedida por un ‘:’ luego de la palabra break.

Lista de errores

error_programa: Describe los errores como falta del bloque de sentencias o nombre del programa.
error_conjunto_sentencias: Define los errores como falta de llaves de apertura y cierre o bloque de sentencias vacío de un programa.
error_declarativa: Describe errores en la declaración de variables, tales como falta de ';' al final, falta del tipo de la variable o falta de la propia variable.
error_lista_de_variables: Define posibles errores como la ausencia de una variable al inicio previo a una ',' o al final.
error_funcion: Establece posibles errores en la declaración de una función tales como: falta de la palabra reservada fun; ausencia del identificador de la función; entre otros.
error_lista_parametros: Controla que los parámetros se separen con ',' y que no falte algún parámetro al inicio o al final de la lista.
error_parametro: Chequea que no se defina un parámetro sin su tipo o identificador.
error_bloque_funcion: Describe la imposibilidad de definir un bloque de función vacío.
error_retorno: Controla la descripción de la sentencia de retorno.
error_retorno_expresion: Describe errores posibles en la expresión definida dentro de una sentencia de retorno.
error_condicion: Plantea errores probables en una condición.
error_expresion: Describe los posibles errores en la descripción de una expresión, tales como ausencia de un término o paréntesis en la expresión tof64.
error_termino: Controla errores en los términos de una expresión, tales como ausencia de factores.
error_invocacion: Chequea la falta de paréntesis de cierre en la lista de parámetros de una invocación a función.
error_lista_parametros_reales: Controla el máximo de parámetros permitidos en la invocación a una función además de la separación de los mismos con ','.
error_asignacion: Describe errores en las sentencias de asignación, como falta de identificador a la izquierda del símbolo de asignación, falta del propio símbolo de asignación, falta de la expresión o ';' al final.
error_seleccion: controla los posibles errores sintácticos en la descripción de una selección.
error_mensaje_pantalla: Verifica errores en la sentencia encargada de mostrar un mensaje por pantalla.
error_invocacion_discard: Controla los errores en el llamado a una función discard.
error_dountil: Describe errores en una sentencia do until y do until con etiquetado.
error_asignacion_do_until: Verifica errores en la asignación perteneciente a la sentencia do until.
error_bloque_sent_ejecutables: Chequea la sintaxis del bloque de sentencias ejecutables dentro de un do until.
error_bloque_de_sentencias_ejecutables_etiqueta: Chequea la sintaxis del bloque de sentencias ejecutables dentro de un do until con etiquetado.

Conclusiones

Finalmente, mediante la elaboración de este trabajo, se pudo comprender y abordar de manera práctica el funcionamiento de las primeras etapas de un compilador, la cual resultó desafiante a la hora de la implementación y el testeo de su correcto funcionamiento. Esto nos llevó a reflexionar y dimensionar el trabajo que conlleva realmente realizar un compilador para un lenguaje completo y más complejo que el propuesto por la cátedra. Considerando todo lo mencionado, nos resultó de gran interés y nos plantea un próximo desafío para la segunda entrega de este proyecto