DISEÑO DE COMPILADORES
TRABAJO PRÁCTICO

INTEGRANTES:
Guerrero, Matias
Guidi, Bruno
Iribarren Susino, Federico

GRUPO 15

DOCENTE:
Tommasel, Antonela

RESUMEN

El objetivo del presente informe es analizar las estrategias utilizadas para resolver el problema propuesto por la cátedra de Diseño de Compiladores y las decisiones tomadas.
Se pidió desarrollar un compilador de un lenguaje propuesto por la cátedra. Se decidió escribir el compilador en el lenguaje Java por una cuestión de familiaridad con el mismo.
Es importante resaltar que en el trabajo solamente se consideró el desarrollo del Analizador Léxico y el Analizador Sintáctico quedando pendiente el desarrollo del Generador de Código.

INTRODUCCIÓN

Para hacer un compilador de un lenguaje determinado es necesario conocer la estructura que se esconde detrás y poder entender cómo se compone y qué función cumple cada una de sus partes. Si observamos, para compilar un código de un lenguaje determinado, es necesario atravesar diferentes fases o etapas de compilación.

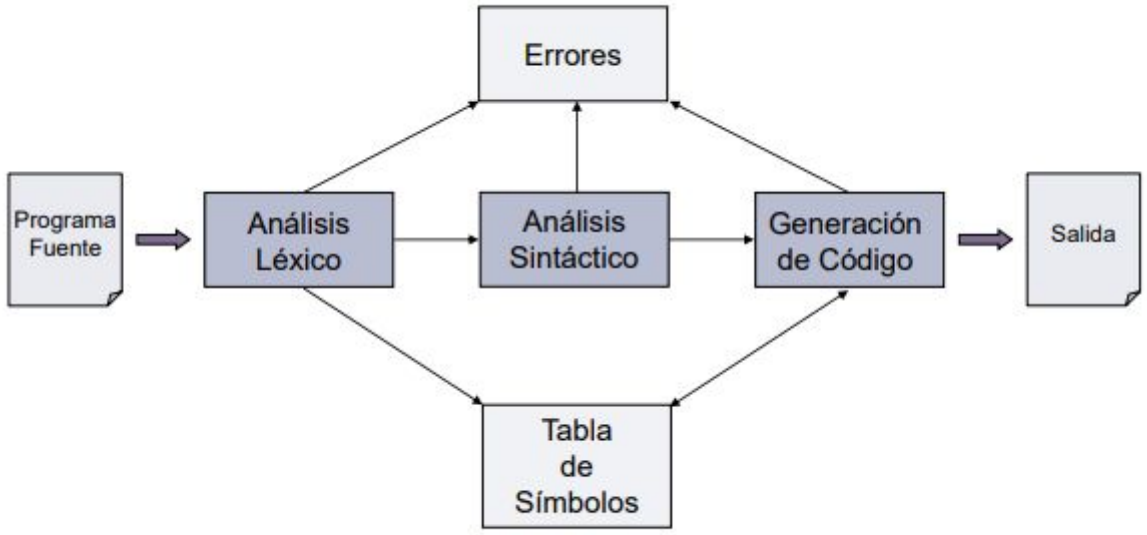

Figura 1. Fases de la Compilación

Si observamos la **Figura 1**, es fácil comprender la estructura general de un compilador. Dado un programa fuente, obtenemos una salida "compilada". A continuación, se mostrarán en detalle cada una de las etapas intermedias que se observan (Análisis Léxico y Análisis Sintáctico, por el momento) y qué decisiones de diseño se tomaron en cada una de las etapas.
Antes de continuar con el desarrollo, es importante remarcar qué objetivos propuso la cátedra para nuestro grupo, ya que estos objetivos son las limitaciones del lenguaje que se desarrolló.

PRIMERA PARTE - ANALIZADOR LÉXICO

Para la primera parte (correspondiente al Analizador Léxico), se pidió resolver los siguientes puntos del enunciado:

El Analizador Léxico debe reconocer:

- Identificadores cuyos nombres pueden tener hasta **20 caracteres de longitud**.
  El **primer carácter debe 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 identificador sólo pueden ser minúsculas)**

- Palabras reservadas (en mayúsculas): **IF, THEN, ELSE, END_IF, OUT, FUNC, RETURN**

- Operadores aritméticos: **"+", "-" , "*", "/"** agregando lo que corresponda al tema particular.

- Operador de asignación: **"="** Comparadores: **">=", "<=", ">", "<", "==", "!="** V **"{", "}", "(", ")", "," y ";"**

El Analizador Léxico debe eliminar (reconocer y descartar) de la entrada:

- Caracteres en blanco, tabulaciones y saltos de línea, que pueden aparecer en cualquier lugar de una sentencia.

Temas asignados al grupo:

2. Enteros sin signo: Constantes con valores entre 0 y 2^16 – 1. Estas constantes llevarán el sufijo "_ui". Se debe incorporar a la lista de palabras reservadas la palabra UINT.

6. Dobles: Números reales con signo y parte exponencial. El exponente comienza con la letra d minúscula y llevará signo. La parte exponencial puede estar ausente. Ejemplos válidos: 1. .6 -1.2 3.d–5 2.d+34 2.5d+1 13. 0. Considerar el rango 2.2250738585072014d-308 < x < .7976931348623157d+308 -1.7976931348623157d+308 < x < -2.2250738585072014d-308 0.0 Se debe incorporar a la lista de palabras reservadas la palabra DOUBLE.

13. Incorporar a la lista de palabras reservadas las palabras **LOOP y UNTIL.**

18. Comentarios de 1 línea: Comentarios que comiencen con "%%" y terminen con el fin de línea.

21. 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 un guión " - ". (En la Tabla de símbolos se guardará la cadena sin el guión, y sin el salto de línea.

DECISIONES DE DISEÑO E IMPLEMENTACIÓN

A la hora de trabajar en los problemas propuestos, se presentaron diversas cuestiones que tuvimos que resolver.
A continuación se listan todas las decisiones que se tomaron con respecto al Analizador Léxico:

- La **Máquina de Estados** (ME) tiene un método particular usado para transicionar al llegar al **End of File** (EOF), debido a que no hay un símbolo específico para el mismo

- **Funcionamiento analizador léxico:**
    - **Itera** hasta que la máquina de estados esté en el estado final.
        - **En caso de llegar al EOF**: Se invoca a una transición particular de la ME, y se finaliza la ejecución del método. No se avanza en el código fuente porque no hay nada más que leer.
        - **En cualquier otro caso:** se transiciona teniendo en cuenta el estado actual de la máquina y el input leído. Luego se avanza en el código fuente.
    - **Cuando se termina de iterar**, se retorna el valor del último token generado. Como solo se llega al estado final en los casos donde se debe generar un token, está garantizado que no va a haber tokens duplicados.

- Cuando falla un control en una **Acción Semántica** (AS) de generación de token (Id, PR, Cadena, UINT y Double) se reinicia la máquina de estados, para evitar que la máquina esté parada en el estado final al finalizar una iteración en el léxico. De esta manera, se evita generar un duplicado del último token leído en la iteración anterior. Lo mismo aplica para lectura de símbolos inválidos y lectura de descartables.

- Ante cualquier situación en la cual no se cree un token, la máquina transiciona al estado inicial, para retomar la lectura de símbolos si corresponde.

- Cuando se transiciona a partir de un “\n” o del EOF, no se devuelve el último caracter leído a la entrada, por más que la transición “por defecto” lo indique.

- Si se lee un número literal (por ejemplo: 55) se lo considera como UINT.

- Siempre que se lee un “\n”, se ejecuta la AS para contar saltos de línea.

- Al leer un DOUBLE, su exponente por defecto es 0.

DIAGRAMA DE TRANSICIÓN DE ESTADOS

Por cuestiones de legibilidad y para facilitar la lectura, se propuso dividir la máquina de estados en pequeños diagramas que representan la obtención de cada token.
A continuación se listarán dichos diagramas que representan la máquina de estados utilizada en el Analizador Léxico para el reconocimiento de tokens.

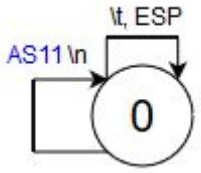

Figura 2.1 Descartables

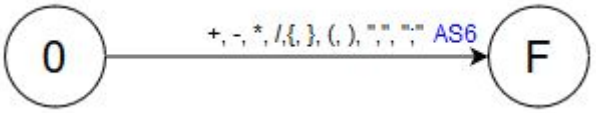

Figura 2.2 Tokens literales

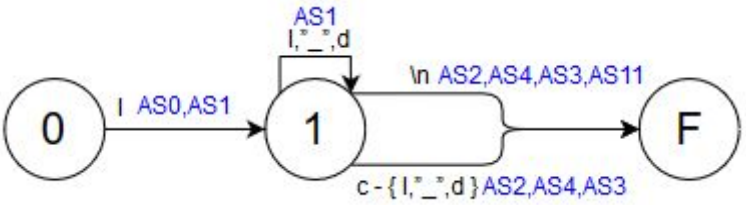

Figura 2.3 Identificadores

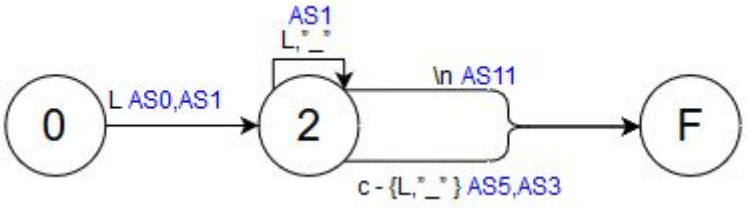

Figura 2.4 Palabras reservadas

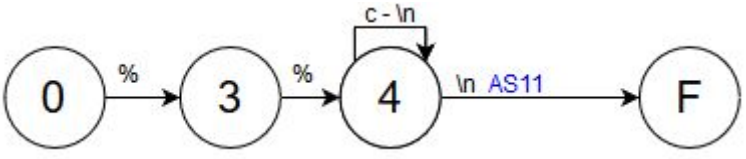

Figura 2.5 Comentarios

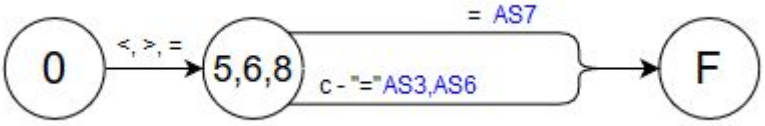

Figura 2.6 Mayor, Menor, Igual, Asignación

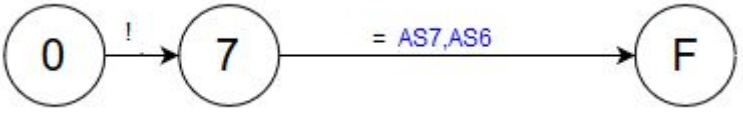

Figura 2.7 Distinto

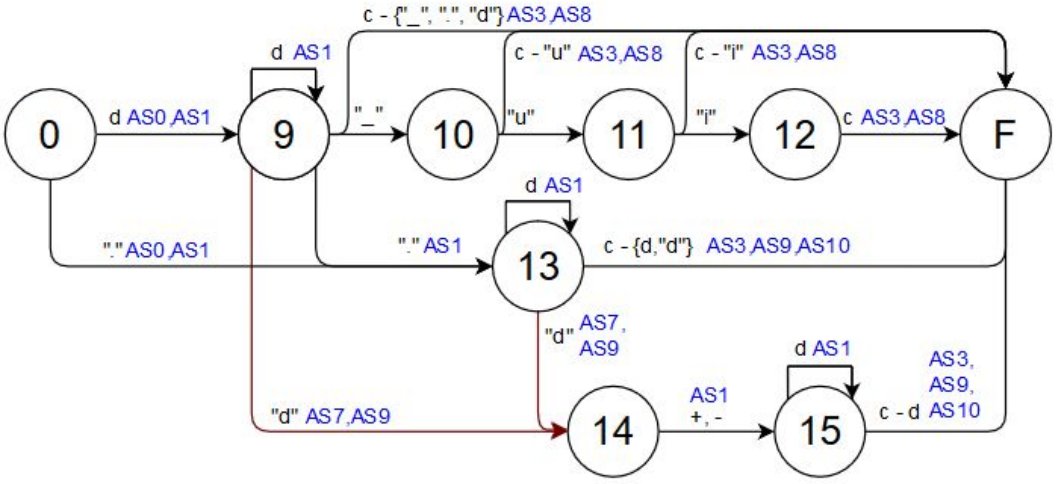

Figura 2.8 UINT, DOUBLE

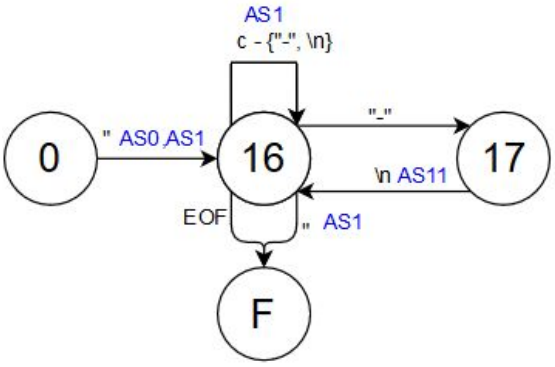

Figura 2.9 Cadena multilínea

MATRIZ DE TRANSICIÓN DE ESTADOS

Si tomamos el diagrama de transición de estados, podríamos representarlo en forma de matriz. Donde las columnas representan los símbolos leídos y las filas los estados de donde partimos y el contenido; el estado a donde nos dirigimos junto con la acción semántica a ejecutar si es que corresponde.
Con respecto a la implementación esto se llevó a cabo creando la clase *MaquinaEstados* que tiene la matriz donde cada celda es de tipo *TransicionEstado* que contiene el estado a transicionar, y el conjunto de acciones semánticas a ejecutar al realizarse la transición. Esta última tiene un método particular para gestionar la transición al alcanzar el EOF.

Por una cuestión de legibilidad, se deja el link de acceso al Google Sheet dónde está la matriz de transición de estados y la matriz de acciones semánticas. Ambas matrices fueron representadas en una sola con el único propósito de simplificar el entendimiento y la lectura de las mismas.

LINK A MATRIZ DE TRANSICIÓN DE ESTADOS Y ACCIONES SEMÁNTICAS

LISTA DE ACCIONES SEMÁNTICAS

En este apartado se describen brevemente las acciones semánticas definidas. Todas las acciones semánticas se basan en la clase AccionSemantica, que define un método *hook* que debe ser sobrescrito por cada implementación particular, según la acción que deba realizarse.

AS0 - InicStringVacio: Inicia un string vacío para almacenar símbolos leídos por el Analizador Léxico. El string en cuestión, es usado por las clases *ConcatenaChar*, *TruncaId*, *GeneraTokenTS*, *GeneraTokenPR* y detección de constantes numéricas.
AS1 - ConcatenaChar: Agrega el último símbolo leído a un string temporal.
AS2 - TruncaId: Trunca el string temporal en caso de que se supere un límite impuesto.
AS3 - RetrocedeFuente: Retrocede en el código fuente, permitiendo volver a leer el último símbolo leído.
AS4 - GeneraTokenTS: Agrega una nueva entrada a la TS, usando como lexema el contenido de un string temporal y agrega el token correspondiente a la lista de tokens.
AS5 - GeneraTokenPR: Corrobora que el contenido del string temporal se corresponda con una palabra reservada. De ser así, agrega el token correspondiente a la lista de tokens. Sino, notifica el error.
AS6 - GeneraTokenParticular: Agrega un token específico a la lista de tokens.
AS7 - ConsumeChar: Ignora el último símbolo leído.
AS8 - GeneraTokenUINT: Parsea el string temporal, chequea el rango del valor almacenado y lo inserta en la TS en caso de que corresponda.
AS9 - ParseBaseDouble: Parsea el string temporal y lo asigna a un atributo asociado a la base de un número double.
AS10 - GeneraTokenDouble: Parsea el string temporal y lo asigna a un atributo asociado al exponente de un double. Luego construye y normaliza el double a partir de la base y exponentes almacenados. Si el double cumple los requisitos se inserta en la TS.
AS11 - CuentaSaltoLinea: Incrementa un contador asociado a la cantidad de líneas del fuente.

ERRORES LÉXICOS CONSIDERADOS

Los errores léxicos considerados fueron 3.

1- Cada vez que se lee una constante literal en el código fuente, se toma como UINT.
2- El exponente por defecto de los doubles es 0, es decir si leemos:
9.5d+, lo que vamos a obtener es 9,5d0.
3- Si en una cadena nos encontramos con el final del archivo (EOF) se produce un error.
4- Si ante un salto de línea en una cadena, no se encuentra el '-', el léxico no genera un error, sino que toma como válido el salto de línea y genera un warning.

SEGUNDA PARTE - ANALIZADOR SINTÁCTICO

Para la segunda parte (correspondiente al Analizador Sintáctico), se pidió resolver los siguientes puntos del enunciado:

a) Utilizar **YACC** u otra herramienta similar para construir el parser.
b) Adaptar el Analizador Léxico del Trabajo Práctico 1 para convertirlo en el método o función int **yylex()** (o el nombre que el Parser generado requiera). Tener en cuenta que el léxico deberá devolver al parser, en cada invocación, un token. Para los identificadores, constantes y cadenas, deberá devolver además, la referencia a la entrada de la Tabla de Símbolos donde se ha registrado dicho símbolo, utilizando yylval para hacerlo.
c) Para aquellos tipos de datos que permitan valores negativos **(INTEGER, LONGINT, FLOAT y DOUBLE)** deberán detectar constantes negativas, modificando la tabla de símbolos según corresponda. Será necesario volver a controlar el rango de las constantes, ya que un valor aceptado para una constante por el Analizador Léxico, que desconoce su signo, puede estar fuera de rango si la constante es positiva. Ejemplo: Las constantes de tipo **INTEGER** pueden tomar valores desde –32768 a 32767. El Léxico aceptará la constante 32768 como válida, pero si se trata de una constante positiva, estará fuera de rango.
d) Cuando se detecte un error, la compilación debe **continuar**.
e) Conflictos: **Eliminar TODOS LOS CONFLICTOS SHIFT-REDUCE Y REDUCE-REDUCE** que se presenten al generar el parser.

DESCRIPCIÓN DEL PROCESO DE DESARROLLO DEL ANALIZADOR SINTÁCTICO

El proceso de desarrollo del analizador sintáctico se divide en dos etapas. En la primera, se definió la gramática del lenguaje, teniendo en cuenta los tipos de datos, tipos de sentencia y demás requerimientos asignados por la cátedra. Luego, se generó el parser a partir de la gramática antes mencionada usando la herramienta YACC. Dicha herramienta crea una máquina de estados de pila, la cual permite reconocer los terminales y no terminales definidos en la gramática.
La gramática puede dividirse en tres secciones: terminales, regla de inicio y reglas de la gramática.
Los terminales, también llamados tokens, representan los distintos elementos que considera la gramática, como por ejemplo operadores aritméticos, identificadores, palabras reservadas, entre otros.
La regla de inicio es la raíz del programa.
Las reglas de la gramática definen no terminales a partir de combinaciones de terminales y no terminales. Sirven para dar forma al lenguaje y así reconocer las estructuras y sentencias requeridas.

LISTA DE NO TERMINALES

A continuación se realiza una breve descripción de todos los NO TERMINALES utilizados en la gramática.

programa: Es la raíz del programa
bloque_sentencias: Compone al programa, describe conjuntos de sentencias.
sentencia: Describe los tipos de sentencia que conforman un bloque.
sentencia_declarativa: Describe declaración de procedimientos o variables.
nombre_proc: Describe la sintaxis de un procedimiento.
params_proc: Describe los parametros utilizados en un procedimiento.
cuerpo_proc: Describe el cuerpo de un procedimiento que está formado por sentencias.
bloque_estruct_ctrl: Un bloque estructural de control contiene sentencias o bloques de sentencias (solo ejecutables).
lista_params: Contiene una lista de parámetros.
param: Describe los posibles tipos de parametros.
param_var: Describe un parámetro pasado por referencia.
param_comun: Describe un parámetro pasado por copia-valor.
tipo: Describe el tipo de una variable.
lista_variables: Describe un conjunto de ID's.
sentencia_ejecutable: Describe sentencias ejecutables: invocación, asignación, loop, if y print.
bloque_sentencias_ejec: Describe un conjunto de sentencias ejecutables.
invocacion: Describe como invocar un procedimiento.
params_invocacion: Describe los parametros utilizados en una invocación.
asignacion: Describe una asignación.
expresion: Describe una expresión y sus operaciones.
termino: Describe un término y sus operaciones.
factor: Describe un factor y sus operaciones.
sentencia_loop: Describe un loop.
cuerpo_loop: Describe el cuerpo de un LOOP.
cuerpo_until: Describe el cuerpo de un UNTIL.
sentencia_if: Describe una sentencia IF.
encabezado_if: Describe un IF y su condicion.
rama_then: Describe la rama THEN de una sentencia IF.
rama_else: Describe la rama ELSE de una sentencia IF.
condicion: Describe una condicion.
comparador: Describe un comparador que se utiliza en una condicion.
print: Describe la sintaxis de un print.
imprimible: Describe todo lo que se puede imprimir por pantalla.

LISTA DE ERRORES SINTÁCTICOS

A continuación se muestran todos los NO TERMINALES que implementan errores. Para poder comprenderlos fácilmente se mostraran tal cual se definieron en la gramática. Los errores se notifican con acciones que se ejecutan en el método *yyerror* acompañado de una breve descripción del mismo.

bloque_sentencias: sentencia *{yyerror("Falta ';' al final de la sentencia");}*
| sentencia ';'
| sentencia ';' bloque_sentencias
;

sentencia: sentencia_declarativa
| sentencia_ejecutable
| error *{yyerror("Sentencia mal definida");}*
;

nombre_proc: PROC ID
| PROC *{yyerror("Procedimiento sin nombre");}*
;

params_proc: '(' lista_params ')'
| '(' lista_params *{yyerror("Falta parentesis de cierre de parametros");}*
;

ni_proc: NI '=' CTE_UINT
| error *{yyerror("Formato incorrecto de NI. El formato correcto es: 'NI = CTE_UINT'");}*
;

lista_params:
| param
| param ',' param
| param ',' param ',' param
| param ',' param ',' param ',' param ',' lista_params *{yyerror("El procedimiento no puede tener mas de 3 parametros");}*
;

param_var: VAR tipo ID
| VAR ID *{yyerror("Falta el tipo de un parametro");}*
| VAR tipo *{yyerror("Falta el nombre de un parametro");}*
;

param_comun: tipo ID
| ID *{yyerror("Falta el tipo de un parametro");}*
| tipo *{yyerror("Falta el nombre de un parametro");}*
;

lista_variables: ID
| ID ',' lista_variables
| error *{yyerror("Lista de variables mal definida");}*
;

params_invocacion: '(' ')'
| '(' lista_variables ')'
| '(' *{yyerror("Falta parentesis de cierre");}*
| '(' lista_variables *{yyerror("Falta parentesis de cierre");}*
;

asignacion: ID '=' expresion
| ID '=' *{yyerror("Falta expresion para la asignacion");}*
;

cuerpo_loop: LOOP bloque_estruct_ctrl
| LOOP *{yyerror("Cuerpo LOOP vacio");}*
;

sentencia_if: encabezado_if rama_then rama_else END_IF
| encabezado_if rama_then END_IF
| encabezado_if rama_then rama_else *{yyerror("Falta palabra clave END_IF");}*
| encabezado_if rama_then *{yyerror("Falta palabra clave END_IF");}*
;

encabezado_if: IF condicion
| condicion *{yyerror("Falta palabra clave IF");}*
;

rama_then: THEN bloque_estruct_ctrl
| THEN *{yyerror("Cuerpo THEN vacio");}*
| bloque_estruct_ctrl *{yyerror("Falta palabra clave THEN");}*
;

rama_else: ELSE bloque_estruct_ctrl
| ELSE *{yyerror("Cuerpo ELSE vacio");}*
;

condicion: '(' ')' *{yyerror("Condicion vacia");}*
| '(' expresion comparador expresion ')'
;

CONCLUSIONES

Gracias a las pruebas realizadas y solicitadas por la cátedra se observa que el Analizador Léxico y Sintáctico funcionan correctamente. En esta entrega, se desarrolló un generador de tokens y una gramática que permiten que nuestro “compilador” entienda un código y sus diferentes contextos en los que más adelante será ejecutado.