Diseño de Compiladores I

Compilador - TP 1 y 2

Grupo 13

Rivero Bernardo Jordi

bernardojordir@gmail.com

Pianciola Bartol Galo Emanuel

gpianciolabartol@alumnos.exa.unicen.edu.ar

Velazquez Belén Rocío

velazquezbelenr@gmail.com

UNICEN
Universidad Nacional del Centro de la Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS
UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

Índice

Introducción

El objetivo del trabajo planteado es implementar un compilador en su totalidad. En esta parte del trabajo, tratamos dos aspectos del mismo: analizador léxico y sintáctico.

El primero está encargado de leer el programa fuente y agrupar los caracteres del mismo en unidades llamadas tokens. Dichos tokens son una secuencia de caracteres que conforman una unidad significativa, teniendo cada uno de ellos un ID (representando el tipo de token), y un lexema, siendo ésta la cadena de caracteres que representan. Algunos de estos tokens pueden representar más de un lexema, por lo que el A.L debe enviar información adicional a la tabla de símbolos. Además, existen tokens que no tienen lexema, tales como símbolos reconocidos por el código ASCII o palabras reservadas.

La tabla de símbolos es una estructura de datos que contiene un registro para cada identificador, constante o cadena encontrada en el programa, donde cada registro contiene la información relevante necesaria.

Por otro lado, el sintáctico se encarga de invocar al analizador léxico para que este le entregue tokens (hasta llegar a el final del archivo del código fuente), para que pueda analizar la sintaxis de las estructuras del código y, en caso de detectar errores, informarlos al usuario. Para implementar esta parte del trabajo se generó una gramática, la cual se explicará más adelante.

Temas asignados

- **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 < 1.7976931348623157d+308
  -1.7976931348623157d+308 < x < -2.2250738585072014d-308 0.0
  Se debe incorporar a la lista de palabras reservadas la palabra DOUBLE.

- **8. Control de invocaciones efectuadas – Pasaje de parámetros por referencia**

  Sentencias declarativas:

  - Modificar los encabezados de declaraciones de procedimientos con la siguiente estructura:
    PROC ID (<lista_de_parámetros>) NI=n { … }
    Donde n será una constante de tipo entero (temas 1-2-3-4)
  - Modificar la estructura de cada parámetro, permitiendo usar la palabra reservada REF, delante de cada parámetro.
  - Ejemplos de declaraciones de procedimiento válidas:
    PROC f1(INTEGER x, REF FLOAT y, FLOAT z) NI = 2 {…}
    PROC f2(REF DOUBLE a, REF DOUBLE b) NI = 4 {…}
    // Incorporar NI y REF a la lista de palabras reservadas.

  Sentencias ejecutables:

  - Modificar las invocaciones a procedimientos, incorporando notación explícita (indicando los nombres de los parámetros formales) para los parámetros:
  - La invocación a una procedimiento será:
    ID(<parámetros>)
    Donde cada parámetro de la lista tendrá la siguiente estructura: ID:ID
  - Ejemplos de invocaciones a procedimiento válidas:
    f1(z:i, y:j, x:k) f2(a:m, b:n)
    // Incorporar ":" a la lista de tokens aceptados por el Analizador Léxico.

- **14. FOR**

  FOR (i = n; <condicion> ; <incr_decr> j) <bloque_de_sentencias>

  i debe ser una variable de tipo entero (1-2-3-4).
  n y j serán constantes de tipo entero (1-2-3-4).
  <condicion> 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).
  <incr_decr> puede ser UP o DOWN
  // Incorporar UP y DOWN a la lista de palabras reservadas.

- **16. Conversiones Explícitas:** Se debe incorporar en todo lugar donde pueda aparecer una expresión, la siguiente sintaxis:

  <tipo> (<expresion>), donde <tipo> será: DOUBLE

- **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

Como primera decisión se determinó utilizar el lenguaje de programación Java, principalmente debido a que el equipo de trabajo se desenvuelve bien y se siente cómodo, cuenta con experiencia en su uso y entiende que es compatible con este trabajo.

El proyecto consta de cuatro clases principales: *Lexico*, *Main*, *Token* y *TablaSimbolos* y una clase abstracta denominada *AccionSemantica*. Dentro de la clase Lexico se encuentran las declaraciones de la matriz de transición de estados, de la matriz de acciones semánticas y de cada uno de los tokens y palabras reservadas. Además es la clase que posee el método yylex() que retorna un valor entero para luego ser usado por el analizador sintáctico. La elección de matrices estáticas para las recién mencionadas se basa en que ninguna de las dos necesita modificar su tamaño a lo largo de la ejecución, y además permite fácil acceso mediante el conocimiento de los valores de filas y columnas; en este caso, estado actual y símbolo entrante.

Por otro lado, la clase *TablaSimbolos* contiene una estructura dinámica con el objetivo de almacenar los tokens que lee el analizador léxico y permitir el acceso a estos al analizador sintáctico. Para ello, utilizamos un Hashtable tomando como key un String que representa al lexema propio de cada token y un entero que representa el tipo del token en cuestión. Cabe aclarar que no se almacenan en dicha estructura tokens que no poseen lexema.

La clase abstracta *AccionSemantica* posee un atributo de tipo String utilizado como buffer (o string temporal) para ir generando la secuencia de símbolos que el analizador léxico lee y un método run que deben implementar cada una de las clases que la heredan. En este caso decidimos que tanto las acciones semánticas como los errores hereden de dicha clase, lo cual permite que ante la entrada de un símbolo nuevo, se ejecute el método *run* correspondiente al error o a la acción semántica propio de la coyuntura entre dicho símbolo y el estado actual en la matriz de acciones semánticas.

Por otro lado, se implementó el Analizador Sintáctico, para construirlo se generó la gramática requerida llamada *gramática.y* y utilizando la herramienta Yacc se generaron las clases *Parser* y *ParserVal*.

En el proceso de construcción de la gramática, se tuvo en cuenta que se reconozcan todas las sentencias enunciadas en el trabajo, haciendo énfasis en la definición de las reglas y la detección de los conflictos shift-reduce y reduce-reduce, para luego tomar medidas y solucionarlos. Finalmente al definir la gramática no se encontraron conflictos a solucionar.

Con respecto al manejo de errores sintácticos, se buscó detectar aquellas sentencias con elementos faltantes y, teniendo en cuenta que al detectar un error el compilador debe continuar, se agregaron reglas de error orientadas a permitir que en el caso de detectar un error se imprima por consola una descripción del mismo y en qué línea fue detectado y continuar compilando. Por ejemplo: en el caso de la sentencia IF, cuya estructura es la siguiente:

IF (<condicion>) <bloque_de_sentencias> ELSE <bloque_de_sentencias> END_IF

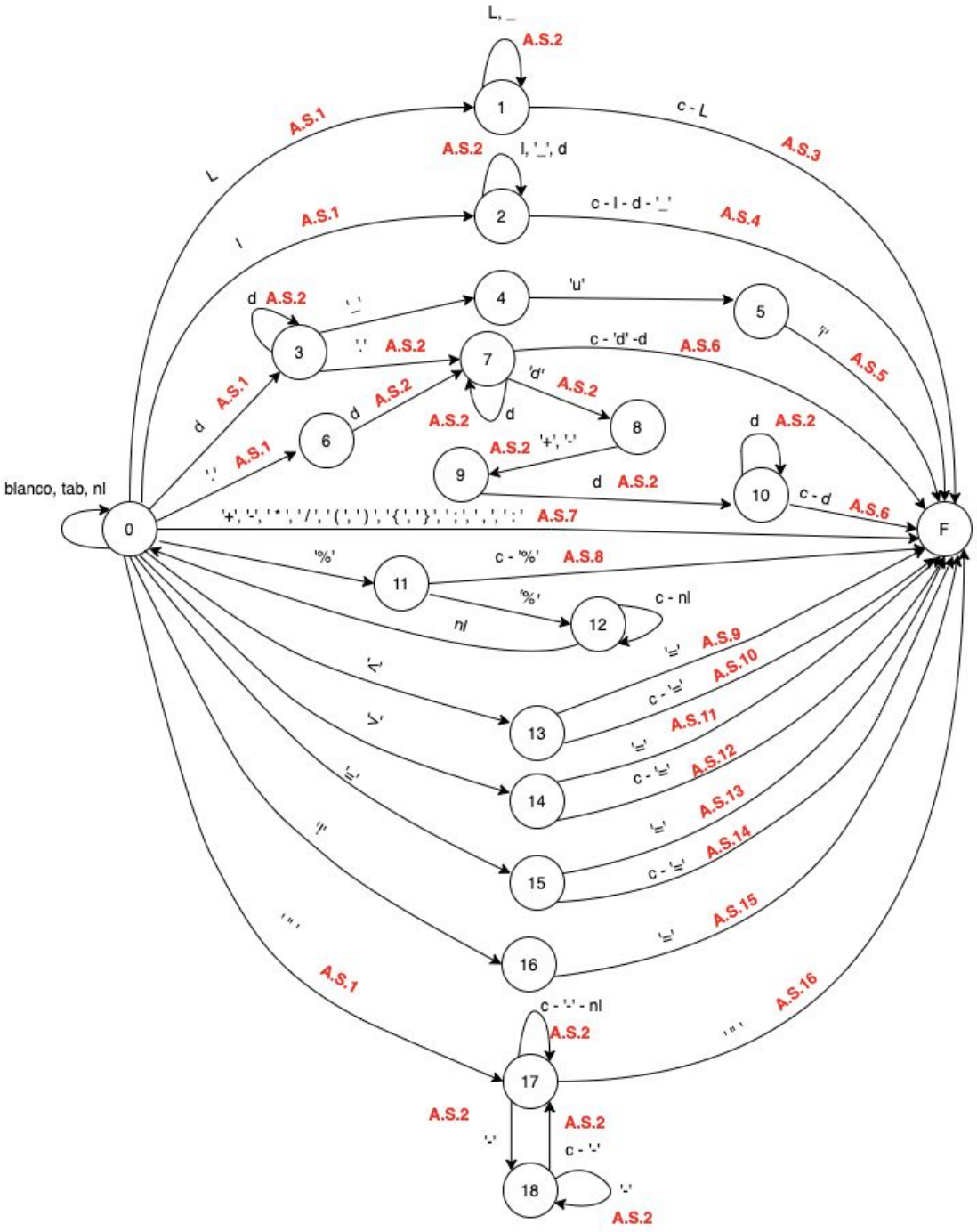
, se agregaron reglas de error para detectar una sentencia en la cual falte la condición, el bloque de sentencias, los paréntesis o alguna palabra reservada requerida. Habiendo explicado el enfoque con el que realizamos el manejo de errores, es necesario informar que existe un caso de prueba (AsignacionMalDefinida4.txt) en el que no pudimos salvar el error sintáctico, en caso de declarar una asignación sin ‘;’ al final de la sentencia el compilador detecta el error pero no logra retomar su compilación.

Además de lo anteriormente mencionado, se considera necesario informar como es el manejo de los comentarios en este compilador, se determinó que al detectarlos se descarten directamente, sin impresiones por pantalla ni guardado alguno. Sin embargo, el compilador es capaz de detectar falencias en la declaración de un comentario y en tal caso informarlo por pantalla.

Por otro lado, en el analizador sintáctico, hizo falta distinguir la entrada del token ‘-’ como negador de un factor con respecto a la que funciona como operando. Para ello se creó en la gramática una regla que interpreta un posible factor negado, el cual ejecuta una acción en la cual se controla el tipo de factor. Ante esto se forman dos posibles caminos, el primero es que dicho factor sea una constante de tipo entero sin signo y por lo tanto se informa un error y se elimina dicho token de la tabla de símbolos. El segundo es que sea una constante de tipo DOUBLE, para la cual se chequea que el negado de la misma esté dentro del rango, de ser así se modifica su lexema en tabla de símbolos y si no cumple dicho requisito se elimina e informa el error. También, por cómo está implementada la gramática, existe el caso en que como factor negado se genere un factor “- IDE” el cual consideramos válido a la hora de implementar el trabajo en cuestión.

Diagrama de transición de estados

c: Conjunto de caracteres aceptados por el lenguaje, incluye L, l, d, '.', '%', ‘<’,’>’,’=’,’!’, ' " ', nl, blanco, tab , '+', '-', ' * ', ' / ', ' (', ') ', ' { ', ' } ', ' ; ', ' , ', ‘:’



Matriz de transición de estados

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	L	l	d	.	%	<	>	=	“	!	+	-	_	‘u’	‘i’	blanco , tab	‘d’	otro	\n	$
0	1	2	3	6	11	13	14	15	17	16	F	F	-1	2	2	0	2	F	0	F
1	1	F	F	F	F	F	F	F	F	F	F	F	1	F	F	F	F	F	F	F
2	F	2	2	F	F	F	F	F	F	F	F	F	2	F	F	F	F	F	F	F
3	-1	-1	3	7	-1	-1	-1	-1	-1	-1	-1	-1	4	-1	-1	-1	-1	-1	-1	F
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	-1	-1	-1	-1	-1	F
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F	-1	-1	-1	-1	F
6	-1	-1	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F
7	F	F	7	F	F	F	F	F	F	F	F	F	F	F	F	F	8	F	F	F
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9	9	-1	-1	-1	-1	-1	-1	-1	F
9	-1	-1	10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F
10	F	F	10	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
11	F	F	F	F	12	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	0	F
13	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
14	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
15	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
16	-1	-1	-1	-1	-1	-1	-1	F	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	F
17	17	17	17	17	17	17	17	17	F	17	17	18	17	17	17	17	17	17	-1	F
18	17	17	17	17	17	17	17	17	17	17	17	18	17	17	17	17	17	17	17	F

Aclaración:

- El estado -1 se considera un error.
- Otro es el conjunto de caracteres de c menos los que están en las columnas de la tabla.

Descripción del mecanismo empleado para implementar la matriz de transición de estados y la matriz de acciones semánticas

Para este aspecto del compilador se implementaron en la clase *Lexico* las matrices de transición de estados y de acciones semánticas con dimensiones 19 x 20 (filas x columnas, estados x carácter entrante). En ambas, la metodología se basa en representar de forma matricial lo que figura en el diagrama de transición de estados mostrado anteriormente, en el cual se muestra para cada transición la acción semántica a ejecutar en caso de ser necesario. Además se incorporan a la matriz de acciones semánticas los posibles errores asociados a distintas transiciones de estados

En la primera matriz, simplemente se tiene en cuenta en qué estado se encuentra actualmente y, dependiendo de qué carácter es el entrante, se determina el estado próximo. En la estructura de acciones semánticas la metodología es similar, ya que se tiene en consideración el estado actual y el símbolo entrante, pero aquí, una vez que se tiene la posición indicada en la matriz, se ejecuta el método *run* del objeto correspondiente a dicha celda.

Esos objetos que se tienen en la tabla de acciones semánticas, son de la clase *AccionSemanticaX*, siendo X el número de la misma. Cada una de estas clases hereda de la clase abstracta *AccionSemantica* descrita previamente, dicha clase posee un *buffer* y una declaración abstracta del método *run* mencionado anteriormente para procurar que en toda celda de la matriz de acciones haya una acción para ejecutar (siempre y cuando el objeto no sea nulo). Con respecto al atributo *buffer,* existe para que las acciones semánticas heredantes de ella, lo usen para almacenar los caracteres entrantes y formar el token.

Acciones semánticas

- **Acción semántica 1:** Inicializa buffer y agrega el primer carácter al mismo.
- **Acción semántica 2:** Está implementada para agregar un carácter al buffer.
- **Acción semántica 3:** Devuelve a la entrada el último carácter leído. Verifica en la tabla de Palabras Reservadas si está el contenido del buffer, en caso de que esté devuelve el id de la palabra reservada e informa por pantalla haber leído una palabra reservada. En otro caso imprime un error.
- **Acción semántica 4:** Devuelve a la entrada el último carácter leído. Se encarga de controlar que el identificador tenga un tamaño menor a 20 caracteres, de no ser así imprime un warning y lo retorna truncado. Luego verifica si está en la tabla de símbolos, si está devuelve su lexema, si no está lo da de alta en la tabla y devuelve el identificador del token junto con su lexema.
- **Acción semántica 5:** Verifica que la constante se encuentre dentro del rango (0, (2^16) - 1), de ser así lo da de alta en tabla de símbolos y retorna el token. En caso de no cumplir con dicha condición genera un error.
- **Acción semántica 6:** Devuelve a la entrada el último carácter leído. Verifica rango de la constante: (2.2250738585072014d-308 < x < 1.7976931348623157d+308 U -1.7976931348623157d+308 < x < -2.2250738585072014d-308 U 0.0). Finalmente la da de alta en tabla de símbolos y devuelve el token.
- **Acción semántica 7:** Se encarga de generar un token con el código ASCII correspondiente al símbolo entrante.
- **Acción semántica 8:** Devuelve a la entrada el último carácter leído. Genera un error debido a que se leyó un comentario con formato erróneo.
- **Acción semántica 9:** Retorna token con el número de token correspondiente al ‘<=’.
- **Acción semántica 10:** Genera un token con el código ASCII correspondiente al símbolo ‘<’ y devuelve a la entrada el último carácter leído.
- **Acción semántica 11:** Retorna token con el número de token correspondiente al ‘>=’.
- **Acción semántica 12:** Genera un token con el código ASCII correspondiente al símbolo ‘<’ y devuelve a la entrada el último carácter leído.
- **Acción semántica 13:** Genera token con el número de token correspondiente al ‘==’.

- **Acción semántica 14:** Genera un token con el código ASCII correspondiente al símbolo ‘=’ y devuelve a la entrada el último carácter leído.
- **Acción semántica 15:** Retorna token con el número de token correspondiente al ‘!=’.
- **Acción semántica 16:** Se encarga de dar de alta en la tabla de símbolos el token correspondiente a la cadena y devolver su identificador su identificador junto con su lexema.

Lista de no terminales

- **programa:** regla inicial de la gramática
- **bloque:** contenedor de un bloque de sentencias (entre ‘{ }’)
- **bloque_sentencias:** puede ser una sola sentencia o un bloque de sentencias
- **sentencia:** puede ser una sentencia declarativa o de ejecución
- **declaracion:** puede ser una declaración de función, o de una lista de variables
- **lista_de_variables:** lista que contiene uno o más identificadores de variable
- **procedimiento:** declaración de un procedimiento (con su bloque dentro)
- **lista_de_parametros:** lista de hasta 3 parámetros (los permitidos)
- **param:** parámetro singular que conforma la lista anterior
- **tipo:** tipo de la variable a especificar (UINT y DOUBLE posibles en nuestro caso)
- **ejecucion:** sentencias de control, selección, salida, asignación, invocación
- **control:** sentencia FOR
- **condicion:** compone la sentencia IF
- **expresion**
- **termino**
- **factor**
- **comparador:** regla que detecta los símbolos de comparación ‘>’, ‘>=’, ‘<’, ‘<=’, ‘==’, ‘!=’
- **inc_decr:** sentencia que modifica el accionar de un FOR
- **seleccion:** sentencia IF
- **salida:** sentencia de print
- **asignacion**

- **invocacion**
- **parametros:** regla que relaciona parámetros singulares formales y reales en invocaciones.

Errores léxicos considerados

- **Error 1:** El carácter leído no es válido porque no pertenece al lenguaje.
- **Error 2:** Se esperaba un dígito, '_' o '.' y llegó otro carácter.
- **Error 3:** Se esperaba una 'u' después de leer un dígito y llegó otro carácter.
- **Error 4:** Se esperaba una 'i' después de leer una 'u' y llegó otro carácter.
- **Error 5:** Se esperaba un dígito y llegó el carácter.
- **Error 6:** Se esperaba un '+' ó un '-' y llegó otro carácter.
- **Error 7:** Se esperaba un '=' después de leer '!' y llegó otro carácter.
- **Error 8:** No está permitido realizar un salto de línea antes de un '-'.
- **Error 9:** La constante se encuentra fuera de rango.
- **Error 10:** El lenguaje no reconoce la palabra reservada.
- **Error 11:** Se esperaba un salto de línea al terminar el comentario.
- **Error 12:** La cadena no se cerró correctamente.

Tener en cuenta que a la hora de hacer las pruebas del analizador léxico, en caso de ser un token válido o no se informará por pantalla, por como están escritas dichas pruebas siempre darán un error sintáctico.

Errores sintácticos considerados

- **error_bloque**: Bloques de sentencias mal declarados, faltan las llaves que cierran o abren el bloque.
- **error_declaracion**: Sentencia declarativa mal definida, falta ‘;’ al final de la misma.
- **error_lista_de_variables**: Faltan las ‘,’ que separan las variables en una sentencia declarativa.
- **error_proc**: Declaraciones de procedimientos mal definidos.
- **error_lista_de_parametros**: errores en la definición de los parámetros de un procedimiento, exceden la cantidad máxima o faltan las ‘,’ entre los mismos.
- **error_ejecución**: Mal declarada una sentencia ejecutable, falta ‘;’ al final de la misma.
- **error_for**: Sentencia de control mal declarada, faltan elementos en la misma.
- **error_if**: Sentencia de selección mal declarada, faltan elementos en la misma.
- **error salida**: Sentencia de salida de mensajes por pantalla mal declarada, faltan elementos en la misma.
- **error_asignacion**: Sentencia de asignación mal declarada, faltan elementos en la misma.
- **error_invocacion**: Sentencia de invocación a procedimientos mal declarada, faltan elementos en la misma.
- **error_parametros**: errores en los parámetros de una sentencia de invocación, faltan elementos en su estructura, por ejemplo ‘:’ ó ‘,’ entre los parámetros.

Conclusión

A lo largo del desarrollo de esta primera parte del compilador, implementando analizadores léxico y sintáctico, se pudieron materializar todos los conceptos que se fueron viendo a lo largo de la cursada hasta el momento. Este aspecto nos ayudó a que dichos temas no queden sólo en lo teórico, sino también en lo práctico de forma tangible.

Creemos que el hecho de haber implementado cada aspecto (léxico y sintáctico) de estos 2 primeros trabajos prácticos por separado, y después relacionarlos, hace que se entienda de mejor manera el trabajo en su conjunto, y esperamos que así también nos resulte en las siguientes etapas.

Para concluir lo dicho, consideramos que lo experimentado a lo largo del desarrollo de este trabajo con respecto a las dificultades y obstáculos con los que nos interceptamos y, en su gran mayoría, pudimos solucionarlos nos hace pensar que el objetivo del mismo fue alcanzado.