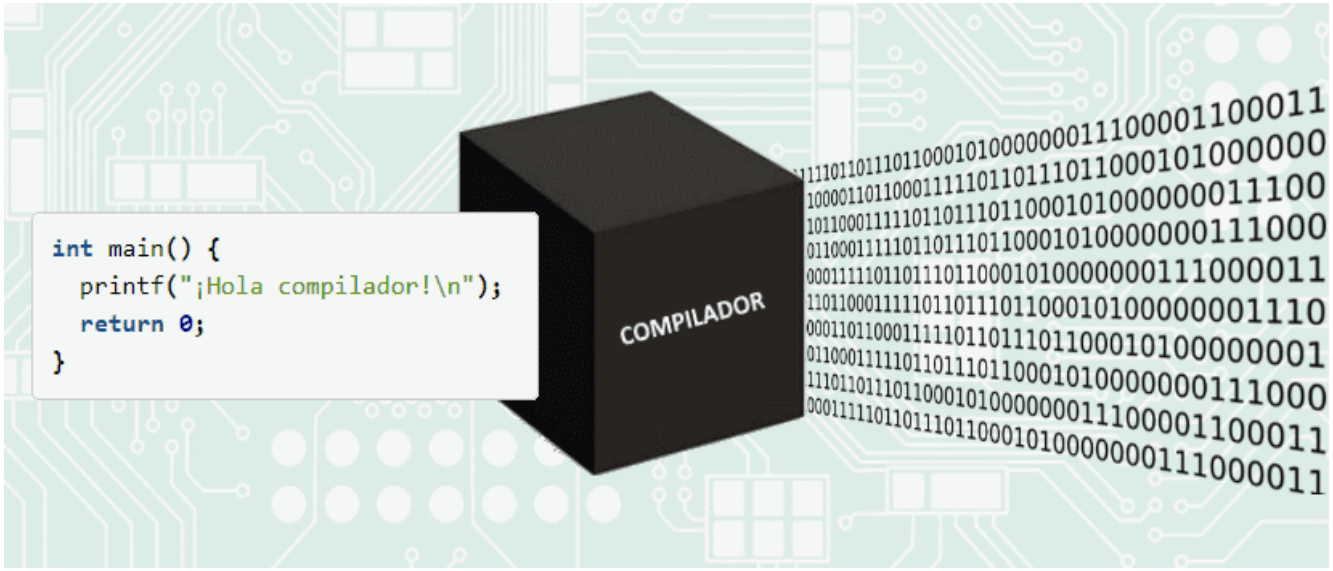
Diseño de Compiladores 1

Grupo 10 - Tp 1 y 2



Integrantes

Medioli Nicolas - mediolifnicolas@gmail.com

Arias Facundo - facundoarias1231@gmail.com

Docente asignado

Jose Fernandez Leon

Temas asignados

6 8 12 14 18 20 21 24 28 29 32 35

índice

Introducción

En este informe se detalla el desarrollo y la implementación de un compilador y las dos primeras partes que lo conforman: Análisis Léxico y Análisis Sintáctico. Las especificaciones del mismo, fueron dadas por la cátedra.

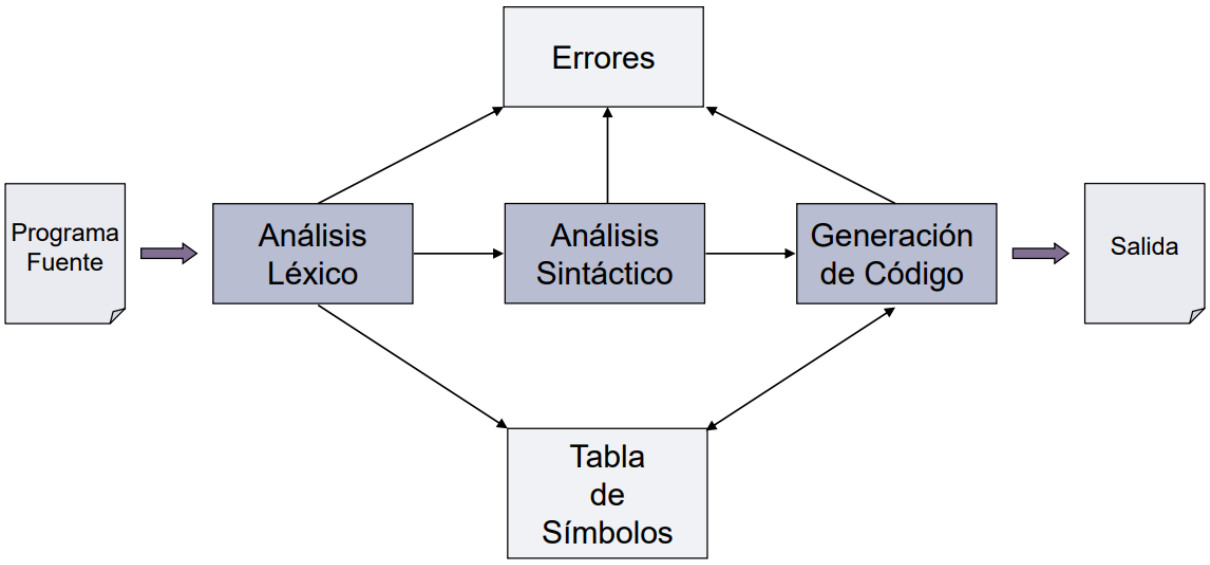


Figura 1: "Estructura general del compilador"

Especificaciones asignadas

6. Enteros largos (32 bits): Constantes enteras con valores entre –2 31 y 231 – 1. Estas constantes llevarán el sufijo "_l".

8. Punto flotante de 64 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra "D" (mayúscula o minúscula) y el signo del exponente es obligatorio. Puede estar ausente la parte entera o la parte decimal, pero no ambas. El „." es obligatorio.

Ejemplos válidos: 1. .6 -1.2 3.d–5 2.D+34 2.5D-1 13. 0. 1.2d+10

Considerar el rango 2.2250738585072014D-308 < x < 1.7976931348623157D+308
-1.7976931348623157D+308 < x < -2.2250738585072014D-308

Se debe incorporar a la lista de palabras reservadas la palabra DOUBLE.

12. En las sentencias de asignación, puede utilizarse el operador '-=' en lugar del operador de asignación '='. Por ejemplo: a -= 10_i, z -= a * b,

14. DO UNTIL DO UNTIL ( ), tendrá la misma definición que la condición de las sentencias de selección. podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

18. Herencia por Composición - Uso con nombre

Incorporar, como sentencia declarativa, la declaración de clases con la estructura que se muestra en los siguientes ejemplos:

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {   // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,     // declaración de atributo
   FLOAT a,     // declaración de atributo
   VOID n() {   // declaración de método
     …
   },
   ca,          // nombre de clase
}
…
```

Incorporar, como sentencia declarativa, la declaración de objetos de una clase determinada:

```
ca a1; a2,
cb b1; b2; b3,
```

Incorporar, dentro de las sentencias ejecutables, la posibilidad de utilizar referencias a métodos y atributos de objetos indicando explícitamente cuando el atributo o método corresponden a la clase heredada por composición.

Ejemplos:

```
a1.a = 3_i,
b1.ca.a = 3_i,
b1.a = 2.3,
b1.b = 1.2,
b1.ca.c = 1_i,
b1.ca.m(),
b1.n(),
```

(todos los chequeos semánticos para estas referencias, se efectuarán en la etapa 3 del TP)

20. Declaración de métodos distribuida La implementación de los métodos de una clase puede efectuarse mediante una cláusula IMPL como se indica en el siguiente ejemplo

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {   // declaración de método
     …
   },



    VOID p(),    // prototipo de método
}

IMPL FOR ca: {
   VOID p() {  // implementación del método p de la clase ca
      ….
    },
   …
}
```

// LÉXICO: Incorporar a los símbolos detectados, el símbolo ':'.

21. Forward declaration

Incorporar la posibilidad de declarar las clases con posterioridad a su uso.
Por ejemplo:

```
CLASS ca {
   VOID m() {
      …
      }
   ...
   }
CLASS cc,
CLASS cd {
   cc c1,
   ca a1,
   VOID j() {
     c1.p(),   // será un error, pero se detectará en la siguiente etapa
     a1.m(),   // ok
   }
}
CLASS cc {     // declaración de clase cc
  INT z,
  VOID p() {
     …
     }
  …
}
```

24. Sobreescritura de atributos La semántica de este tema se explicará en la etapa 3 del Trabajo Práctico

28. La semántica de estos temas se explicará en la etapa 3 del Trabajo Práctico

29. Conversiones Explícitas: Se debe incorporar en todo lugar donde pueda aparecer una expresión, la posibilidad de utilizar la siguiente sintaxis:

TOF () // para grupos que tienen asignado el tema 7

TOD() // para grupos que tienen asignado el tema 8

//LÉXICO: Incorporar a la lista de palabras reservadas, la palabra TOF o TOD según corresponda.

32. Comentarios de 1 línea: Comentarios que comiencen con "**" y terminen con el fin de línea.

35. Cadenas multilínea: Cadenas de caracteres que comiencen y terminen con " % " . Estas cadenas pueden ocupar más de una línea. (En la Tabla de símbolos se guardará la cadena sin los saltos de línea).

Ejemplo: % ¡Hola

mundo! %

Parte 1 - Analizador Léxico

Para desarrollar el analizador léxico tuvimos en cuenta el objetivo y las especificaciones que proporciona la cátedra:

Objetivo

Desarrollar un Analizador Léxico que reconozca los siguientes tokens:

- Identificadores cuyos nombres pueden tener hasta 20 caracteres de longitud. El primer puede ser una letra o '_', 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.
- 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 postergará 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 mayúsculas):
  IF, ELSE, END_IF, PRINT, CLASS, VOID
- y demás símbolos / tokens indicados en los temas particulares asignados a cada 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.

Analizador Léxico. Especificaciones

a) El Analizador Léxico deberá leer un código fuente, identificando e informando:
   - Tokens detectados en el código fuente. Por ejemplo:
     Palabra reservada **IF**
     (
     Identificador **var_x**
     +
     Constante entera **25**
     Palabra reservada **ELSE**
     etc.
   - Errores léxicos detectados en el código fuente, indicando: nro. de línea y descripción del error. Por ejemplo:
     Línea 24: Constante entera fuera del rango permitido
   - Contenidos de la Tabla de Símbolos.

Se sugiere la implementación de un consumidor de tokens que invoque al Analizador Léxico solicitándole tokens. En el trabajo práctico 2, esta funcionalidad estará a cargo del Analizador Sintáctico.

b) El código fuente **debe ser leído desde un archivo**, cuyo nombre **debe poder ser elegido** por el usuario del compilador. Se espera que el compilador pueda ejecutarse desde línea de comandos.

c) La numeración de las líneas de código debe comenzar en 1. De este modo, la información de cada error coincidirá con el número de línea en el archivo del código fuente.

d) Para la programación se podrá elegir el lenguaje. Para esta elección, tener en cuenta que el analizador léxico se integrará luego a un Parser (Analizador Sintáctico) generado utilizando una herramienta tipo Yacc. Por lo tanto, es necesario asegurarse la disponibilidad de dicha herramienta para el lenguaje elegido.

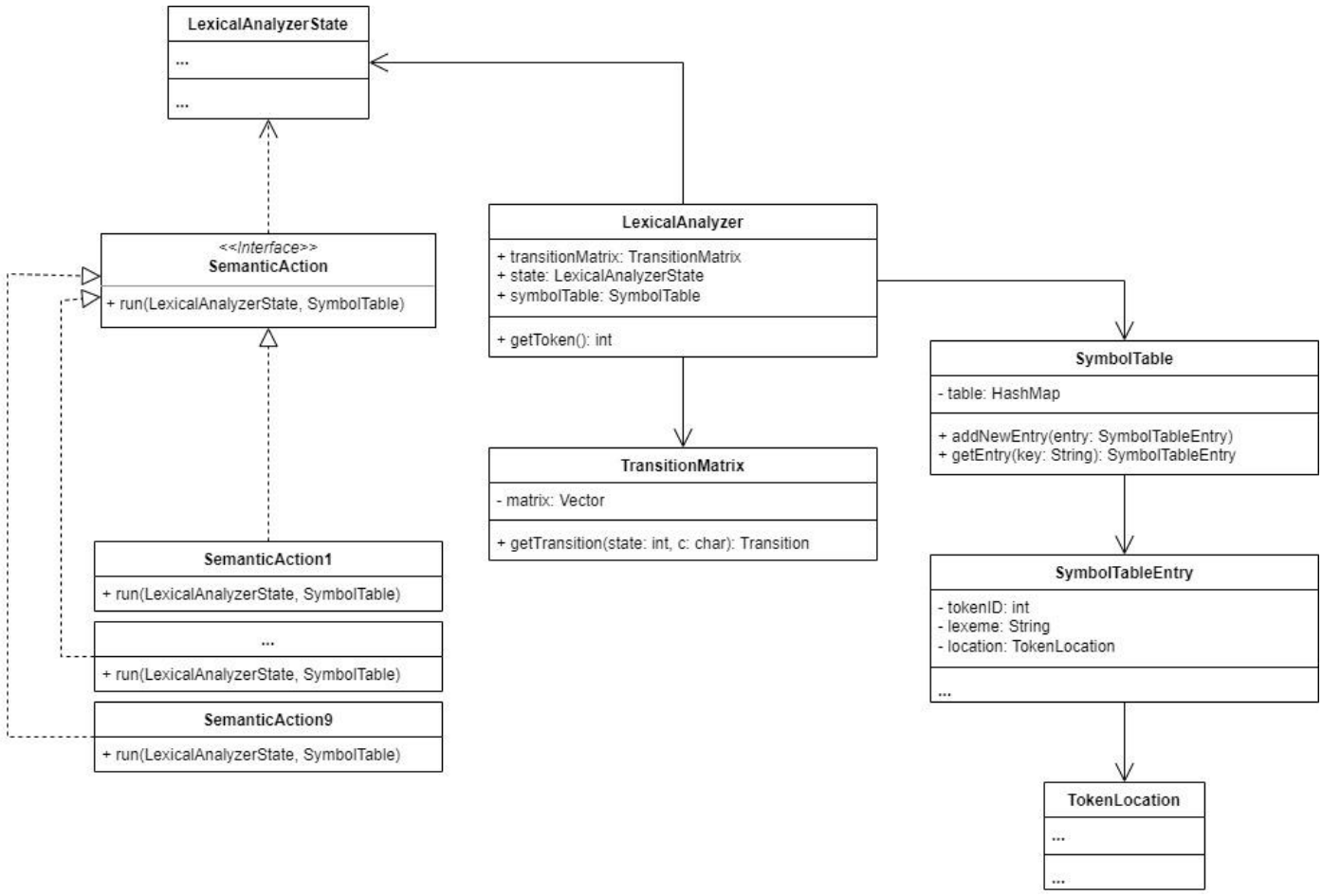
e) El Analizador Léxico deberá implementarse mediante una matriz de transición de estados y una matriz de acciones semánticas, de modo que cada cambio de estado y acción semántica asociada, sólo dependa del estado actual y el carácter leído.

f) Implementar una Tabla de Símbolos donde se almacenarán identificadores, constantes, y cadenas de caracteres. Es requisito para la aprobación del trabajo, que la tabla sea implementada con una estructura dinámica.

g) La aplicación deberá mostrar, además de tokens y errores léxicos, los contenidos de La Tabla de Símbolos.

Decidimos utilizar el lenguaje Java porque es el lenguaje que más conocemos en común y se complementa fácilmente con Yacc, teniendo mucha documentación a disposición.

Estructura general del analizador léxico



Implementación general - LexycalAnalizer

En primera instancia creamos una clase “Compilador” que llama a otra clase “LexicalAnalyzer”, esta última tiene como atributos la matriz de transición, la tabla de símbolos, un arreglo con acciones semánticas y un estado actual, entre otros. Luego se inicia la clase (asignando sus respectivos valores cada atributo). Esta, tiene como método principal “getToken”, el mismo, mientras esté leyendo el token, hará una serie de funciones, entre ellas, pasar al siguiente estado, ejecutar acciones semánticas, etc.

Otra clase que es importante mencionar es “LexicanAnalyzerState”, esta es una instancia del analizador léxico que tiene como atributos el índice del archivo que se está leyendo, estado y lexema actual, una variable booleana que te indica si estas leyendo el token, el token a devolver, etc. En resumen, funciona como un registro que contiene los datos actuales del token que se está leyendo.

La tabla de símbolos es implementada con un HashMap, que tiene dos atributos: String y Token. El token es un objeto a parte que tiene un ID y una variable booleana “predefinided” que indica si ese token está predefinido o no desde un inicio. La tabla está precargada con caracteres y palabras reservadas. Elegimos este tipo de implementación, porque tiene mucha flexibilidad a la hora de buscar y poner nuevas duplas siendo muy eficiente para su uso.

Diagrama de transición de estados

En primer lugar, debíamos realizar el diagrama de transición de estados. Utilizando daw.io creamos todos los posibles estados y sus respectivas transiciones hasta llegar al nodo final (por una cuestión de mejor legibilidad, hay dos nodos finales).

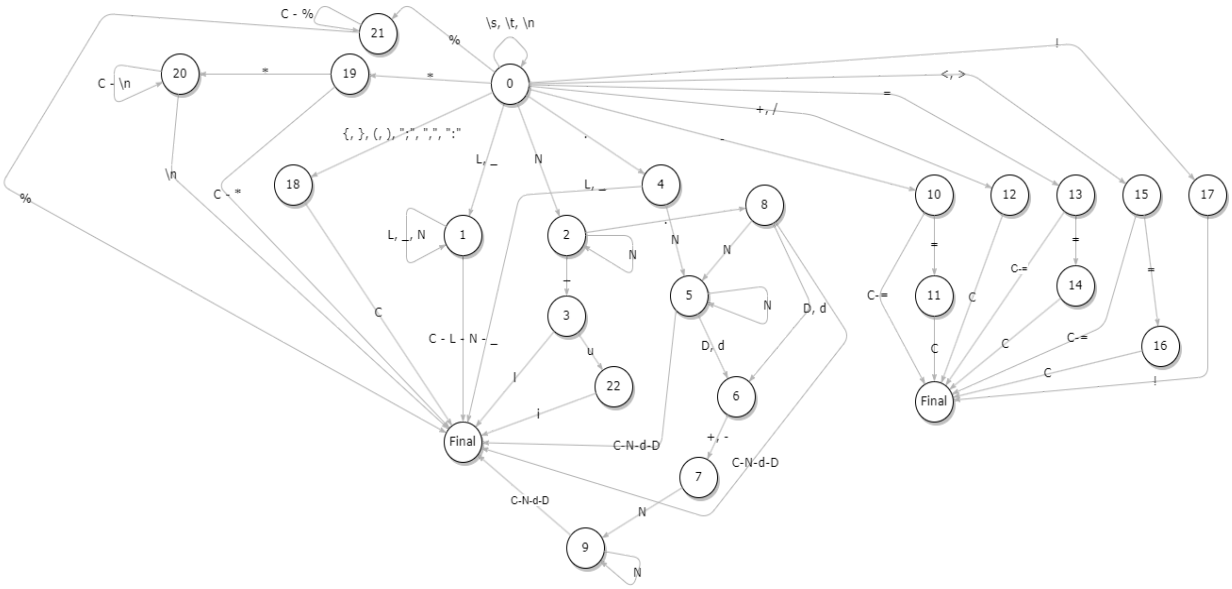


figura 2: "Diagrama de transición de estados"

Para implementar este diagrama en código, se podría haber utilizado una matriz de transición de estados como la que aparece en la teoría, pero tomamos la decisión de volver a utilizar un HashMap como aparece a continuación:

```
public class TransitionMatrix {

    private static Vector<HashMap<Character, Transition>> matrix;
```

Luego, cargamos el HashMap con los datos que aparecen en el diagrama (*figura 2*). Decidimos utilizar esta estructura porque nos pareció eficiente y por las mismas razones que se dieron anteriormente en la tabla de símbolos.

El componente Transition que tiene el HashMap tiene dos atributos, uno es el estado nuevo al que se accede cuando recibe un carácter y el otro es una lista de acciones semánticas que se ejecutan.

```java
public class Transition {

    private Integer newState;
    private List<Integer> semanticActionList;
```

figura 3: "Atributos de la clase Transition"

Por ejemplo:

si el analizador detecta el carácter '_' se pasará al estado 1 (como se marca en el diagrama de estados) ejecutándose la acción semántica 2:

```java
HashMap<Character, Transition> s0 = new HashMap<>();
s0.put(key:'_', new Transition(newState:1, List.of(e1:2)));
```

figura 4: inicialización de HashMap y ejemplo de inserción de dupla

Las acciones semánticas están cargadas en un arreglo en la clase "LexicalAnalyzer.java", entonces cuando se quiera ejecutar alguna acción, simplemente hay que acceder al arreglo en el índice que indica la transición y ejecutarla.

Acciones semánticas

En total generamos 9 acciones semánticas, estas, implementan un método que implementan la interfaz SemanticAction (método run), que recibe como parámetro el estado actual del léxico (anteriormente mencionado) y la tabla de símbolos.

- ***Acción semántica 1:*** Se encarga de decrementar el índice de lectura del archivo.

- ***Acción semántica 2:*** Se encarga de resetear el lexema actual sumándole el último carácter leído.

- ***Acción semántica 3:*** A diferencia de la acción anterior, esta, simplemente le suma el último carácter leído al lexema actual. Pero si el carácter leído es un salto de línea, se omite.

- ***Acción semántica 4:*** Si el lexema tiene más de 20 caracteres, es truncado avisando al usuario mediante un warning. Luego lo busca en la tabla de símbolos predefinidos, y si está, devuelve el token junto a su ubicación. En caso de que no se encuentre, se agrega a la tabla de símbolos dinámica (de ahora en adelante, simplemente tabla de símbolos) el token 'ID', su ubicación y el lexema. Finalmente se retorna el token y su referencia en la tabla de símbolos

- ***Acción semántica 5:*** Verifica el rango de la constante y la da de alta en la tabla de símbolos, luego devuelve el token y su referencia en la tabla.

- ***Acción semántica 6:*** Devuelve un token predefinido

- ***Acción semántica 7:*** Resetea el lexema actual para comenzar con una nueva lectura.

- ***Acción semántica 8:*** Agrega el string (presente en el lexema actual) a la tabla de símbolos y retorna el token 'CTE_STRING' junto a su referencia en la tabla.

- ***Acción semántica 9:*** Incrementa el índice de línea que se está leyendo del archivo. Para incluir la información de ubicación de los tokens en la tabla de símbolos.

Errores léxicos considerados

- Enteros largos de 32 bits que no tengan como sufijo “_l”
- Punto flotante:
    - no contener “.”
    - si se declara la parte exponencial, no contener “d” o “D”
- Comentarios:
    - no comenzar con “**”
    - no finalizar con un salto de línea
- Las cadenas de caracteres que no empiezan y terminan con “%”
- Chequeo de rangos para las constantes.
- Identificadores cuyos nombres tengan más de 20 caracteres (Warning).
- Palabras reservadas en minúsculas.

Parte 2 - Analizador Sintáctico

Al igual que el analizador léxico, tuvimos en cuenta las especificaciones generales dadas por la cátedra para realizar la parte sintáctica:

OBJETIVO

Construir un Parser (Analizador Sintáctico) que invoque al Analizador Léxico creado en el Trabajo Práctico Nº 1, y que reconozca un lenguaje con las siguientes características:

SINTAXIS GENERAL:

Programa:

- Programa constituido por un conjunto de sentencias, que pueden ser declarativas o ejecutables.
  Las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control.
  Los elementos declarados sólo serán visibles a partir de su declaración (esto será chequeado en etapas posteriores).
- El programa estará delimitado por llaves '**{**' y '**}**'.
- Cada sentencia debe terminar con coma "**,**".

Sentencias declarativas:

- Sentencias de declaración de datos para los tipos de datos correspondientes a cada grupo según la consigna del Trabajo Práctico 1, con la siguiente sintaxis:

  <tipo> <lista_de_variables>,

  Donde <tipo> puede ser (Según tipos correspondientes a cada grupo): **SHORT, INT, LONG, USHORT, UINT, ULONG, FLOAT, DOUBLE**
  Las variables de la lista se separan con punto y coma (";")
- Incluir declaración de funciones **VOID**, con la siguiente sintaxis:

  VOID ID (<parametro>)

  {

  <cuerpo_ de_la_funcion>

  }

  Donde:
  - <parametro> será un identificador precedido por un tipo.:

    <tipo> ID

    Se permite hasta un parámetro, y puede no haber parámetros. **Este chequeo debe efectuarse durante el Análisis Sintáctico**
  - <cuerpo_de_la_funcion> es un conjunto de sentencias declarativas (incluyendo declaración de otras funciones) y/o ejecutables, incluyendo sentencias de retorno con la siguiente estructura:

    RETURN,

Ejemplos válidos:

```
VOID f1 (INT y)
{
        INT x,
        x = y,
        …
        RETURN,
}
```

```
VOID f2 (LONG x)
{
        INT x,
        x = y,
        …
        RETURN,
}
```

```
VOID f3 ()
{
        FLOAT x,
        x = 1.2,
        …
        IF (x > 0.0)
                RETURN,
        ELSE
            {
                x = 2.0,
                RETURN,
                }
        END_IF,
}
```

Sentencias ejecutables:

- Asignaciones donde el lado izquierdo puede ser un identificador, y el lado derecho una expresión aritmética. Los operandos de las expresiones aritméticas pueden ser variables, constantes, u otras expresiones aritméticas.
  No se deben permitir anidamientos de expresiones con paréntesis.
- invocación a una función, con el siguiente formato:

  ID(<parametro_real>), // Función con parámetro

  o

  ID ( ), // Función sin parámetro

  El parámetro real puede ser cualquier expresión aritmética, variable o constante.
  Ejemplos:

  ```
  f1(a),        f2(),        f3(a+b),        f4(5_i),
  ```

- Cláusula de selección (**IF**). Cada rama de la selección será un bloque de sentencias. La estructura de la selección será, entonces:

  IF (<condicion>) <bloque_de_sent_ejecutables> **ELSE** <bloque_de_sent_ejecutables> **END_IF,**

  El bloque para el **ELSE** puede estar ausente.
  La condición será una comparación entre expresiones aritméticas, variables o constantes, y debe escribirse entre **"(" ")"**.
  El bloque de sentencias ejecutables puede estar constituido por una sola sentencia, o un conjunto de sentencias ejecutables delimitadas por llaves.
- Sentencia de salida de mensajes por pantalla. El formato será

  PRINT<cadena>,

  Ejemplos:

  ```
  PRINT#Hola mundo#,          //Tema 34
  PRINT %Hola                 //Tema 35
        Mundo%,
  ```

Implementación General - Yacc

Para implementar el analizador sintáctico, utilizamos Yacc para analizar estructuralmente una entrada, esta herramienta requiere 3 especificaciones:

- Una gramática (Conjunto de reglas)
- Código a ser invocado cuando una regla es reconocida
- Un analizador léxico que le provea los tokens

Gramática

Tuvimos que realizar una gramática que reconozca las reglas, esta está contenida en un archivo llamado "compilador.y".

En primer lugar declaramos como tokens las palabras reservadas que actúan como estados finales (además de los tokens con relación 1:1 con ascii) y luego definimos la gramática.

```
%token IF ELSE END_IF PRINT CLASS VOID ID
       LONG UINT DOUBLE STRING
       CTE_LONG CTE_UINT CTE_DOUBLE CTE_STRING
       CMP_GE CMP_LE CMP_EQUAL CMP_NOT_EQUAL
       SUB_ASIGN
       DO UNTIL IMPL FOR RETURN TOD
```

"Figura 5: Definición de palabras reservadas"

Para poder mostrar por pantalla el conjunto de reglas detectados en el código, utilizamos una cola de prioridad en la que se almacenan las diferentes estructuras sintácticas encontradas (SyntacticStructureResult) en base a su ubicación en el archivo.

Esta implementación nos permite imprimir de forma ordenada el conjunto de reglas sintácticas que reconoce el parser.

Lista de no terminales

- **programa:** *encierra todo el programa entre {}.*
- **comparador:** *operadores de comparación.*
- **condicion:** *comparación de expresiones.*
- **tipo:** *incluye palabras reservadas de los tipos de datos.*
- **lista_identificadores:** *lista recursiva de identificadores.*
- **sentencia_declarativa:** *incluye las sentencias de:*
  - declaración de variable
  - definición de función y clases
  - implementación de métodos distribuidos.
- **sentencia_ejecutable:**
  - asignación a variables/atributos
  - invocación a función/método
  - sentencia if, do until, PRINT y RETURN
- **lista_sentencias:** *lista de sentencias de cualquier tipo.*
- **lista_sentencias_ejecutables:** *sólo sentencias ejecutables.*
- **invocacion_funcion:** *invocación a función con o sin parámetro.*
- **op_asignacion_aumentada:** *asignación con = o -=.*
- **sentencia_if:** *sentencia condicional.*
- **constante:** *constante de cualquier tipo de dato.*

- **expr:** *expresión con casting.*
- **basic_expr:** *expresión sin anidamiento.*
- **term:** *término de una expresión.*
- **factor:** *factor de una expresión.*
- **parametro_formal:** *sintaxis de parámetro en declaración de funciones.*
- **parametro_real:** *sintaxis de parámetro en invocación a funciones.*
- **definicion_funcion:** *definición de función.*
- **procedimiento:** *definición de procedimiento (función o método)*
- **do_until:** *sentencia de bucle condicional*
- **metodo:** *definición de método.*
- **acceso_atributo:** *acceso a atributos mediante operador de acceso.*
- **definicion_clase:** *definición de clase.*
- **cuerpo_clase:** *cuerpo de clase según la especificación.*
- **clase_lista_atributos:** *lista de atributos dentro de clase.*
- **clase_lista_metodos:** *lista de métodos dentro de clase.*
- **clase_lista_composicion:** *lista de clases para composición.*
- **implementacion:** *implementación de métodos de forma distribuida.*

Errores sintácticos considerados

- Sentencias que no terminen con coma ‘,’

- Programa que no esté delimitado por llaves ‘{’ ‘}’
- Sentencias declarativas:
    - lista de variables declaradas que no estén separadas con un punto y coma ‘;’
    - Declaración de funciones:
        - Parámetro que no sea del tipo <tipo> ID
        - Cuerpo de la función que no esté delimitado con llaves ‘{’ ‘}’
        - No se incluya en el cuerpo de la función la sentencia “RETURN”
- Sentencias ejecutables:
    - Asignaciones donde el lado izquierdo no sea un identificador
    - Anidamiento de expresiones con paréntesis
    - Cláusula de selección **IF:**
        - Ramas de selección que no sean bloques de sentencias
        - Condición que no esté delimitada con paréntesis ‘(’ ‘)’
        - Bloques de sentencias que no estén delimitadas por llaves ‘{’ ‘}’

Conclusión

Al principio de la cursada no sabíamos cómo se podría traducir un código fuente a un código ejecutable, pero a medida que avanzabamos en este trabajo empezamos a comprender cómo se implementa un compilador, profundizando en su estructura y sus partes, tuvimos que informarnos sobre ciertos temas, tomar decisiones de implementación y aprender a usar herramientas nuevas. Si bien nos quedan dos etapas más que debemos aprender y profundizar, ya hemos podido realizar las dos primeras partes del compilador, hemos aplicado técnicas aprendidas en la carrera, manejamos los errores de manera efectiva y documentamos el proceso.

Para finalizar, la experiencia de crear un software complejo como este, nos puso a prueba y hemos podido mejorar nuestras habilidades y capacidades adquiriendo conocimientos nuevos.