Diseño de Compiladores I – Cursada 2019

Grupo N°: 19

Docente asignado: José Massa

Integrantes: Freiberger Agustín - agustin.freiberger@gmail.com

San Juan Roque - roque.m.sanjuan@gmail.com

Violante Facundo - facundo.violante@gmail.com

Temas asignados:

- (2) Enteros largos sin signo - **ulong** (0,232–1)
- (6) Palabra reservada - **do y until**
- (12) Palabra reservada **class y extends.**
  Declaración de clase.
  Declaración de objetos de una clase.
  Posibilidad de invocación a métodos de clase.
- (14) Conversiones Explícitas: **to_ulong**
- (17) Comentarios Multilínea: Comienzan con **"/+"** y terminan con **"+/"**
- (19) Cadenas de una línea: Comienzan y terminan con **"%"**.

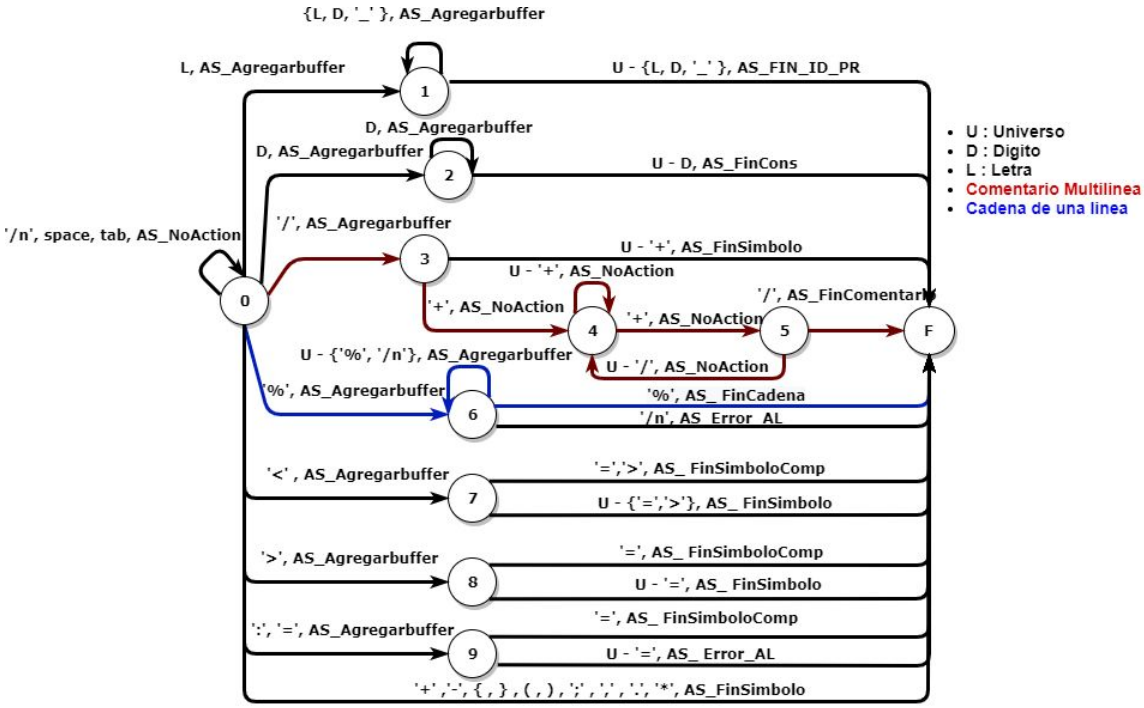
Introducción:

El objetivo de este trabajo práctico es desarrollar un analizador léxico que permita leer texto desde una fuente de entrada y reconozca *tokens*, eliminando espacios en blanco, tabulaciones y saltos de línea. El reconocimiento de *tokens* se realiza mediante un autómata finito que representa la sintaxis que tiene el lenguaje, además de una Tabla de Símbolos para almacenar los identificadores con su correspondiente lexema. Dicho autómata es representado mediante una matriz de estados y asistido por la definición de diferentes acciones semánticas.

Diseño e Implementación:

Primeramente, se decidió leer el archivo por carácter ASCII. Esto fue para identificar correctamente los caracteres "especiales" como saltos de línea, tabulaciones y fin de archivo sin problemas.

Durante la etapa de diseño, se realizó el autómata en el cual se definieron una serie de estados no finales y un estado final (cero). Una vez llegado al estado final, se reconoce o no el *token*.



Se decidió implementar así una matriz de estados, en la que cada casilla indica el estado siguiente, que represente los posibles escenarios y la transición de uno a otro basándose en el siguiente carácter leído.

De esta forma, es posible reconocer el final de cada token y en el caso de recibir un carácter inesperado en algún estado no final, detectar el error léxico correspondiente.

A su vez, por cada transición de estado, se asoció una acción semántica las cuales son acciones que realiza el analizador léxico para la inicialización, finalización y validación de los *tokens* leídos. Estas se representaron también en una matriz de iguales dimensiones que la anterior.

Tanto la matriz de estados como la matriz de acciones semánticas se encuentran anexadas a este informe.

Acciones semánticas:

- **AS_NoAction**: No realiza ninguna acción.

- **AS_AgregarBuffer**: Agrega el carácter al buffer, todavía no se ha identificado un token.

- **AS_Fin_ID_PR**: Reconoce el fin de un token. Verifica si el token pertenece a la tabla de palabras reservadas, en el caso de que no exista, verifica su longitud y lo agrega a la tabla de símbolos.

- **AS_FinCadena**: Reconoce el fin de una cadena y, si no se encuentra en la tabla de símbolos, la agrega.

- **AS_FinComentario**: Reconoce que la secuencia leída fue un comentario, no realiza ninguna acción.

- **AS_FinConst**: Reconoce que termino un token de tipo INT, verifica los límites, si está dentro de los valores permitidos, lo agrega a la tabla de símbolos si no se encuentra aún.

- **AS_FinSimbolo**: Identifica que el token es un símbolo, devolviendo su valor. Esta acción semántica se utiliza solo para símbolos de un solo carácter. Ejemplo: “*”.

- **AS_FinSimboloCompuesto**: Identifica que el token es un símbolo con más de un carácter. Como por ejemplo “>=”.

- **AS_FinSimboloSimple**: Esta acción semántica es un caso especial de símbolos de un solo carácter, se da en aquellos que pueden pertenecer a un símbolo compuesto

como por ejemplo “>” que es un símbolo por sí solo pero además forma parte de un símbolo compuesto.

- **AS_Error**: Se utiliza para marcar errores generados caracteres inválidos fuera de una cadena de caracteres claro está, cómo por ejemplo ‘_’, ‘@’, etc.
- **AS_ErrorAL**: Esta acción semántica identifica el error que se produce luego de leer los caracteres ‘:’ y ‘=’ que sólo son válidos si son seguidos por un ‘=’ y para esto es necesario acomodar la línea (AL).

En las acciones semánticas se desarrolló un método llamado *acomodarLinea()* que será utilizada en algunas acciones para indicar que el último carácter leído tiene que volver a tomarse en cuenta dentro del autómata.

La Tabla de Símbolos donde se almacenan los lexemas fue implementada como un HashMap<String, Símbolo> donde Simbolo es un tipo de dato con valor, tipo y referencia, y la Key es el lexema correspondiente al símbolo. El valor del Símbolo es el lexema y la referencia lleva la cantidad de referencias a ese registro de la tabla.

Errores léxicos considerados

Los errores léxicos se muestran por pantalla junto con la línea donde se encontraron. Se consideraron los siguientes:

- Carácter inválido: este error denota la presencia de un carácter no válido y se identifica con -2.
- Longitud máxima de identificador excedida: este warning se identifica con -3 y deja al identificar con una longitud de 25.
- Constante fuera de rango: este error notifica sobre una constante entera o ulong fuera del rango permitido. Si el entero se pasa de rango positivo se cambia a ulong y si se pasa de rango negativo se trunca al máximo posible. Se identifica con -4.

Trabajo Práctico N° 2

El desarrollo de la gramática tuvo varias idas y vueltas. Particularmente el mayor inconveniente surgió a la hora de identificar la línea en que ocurrían los errores. En primer medida se modificó el tipo de dato ParserVal para que incluya un atributo Token dentro de los que ya tiene y así poder ir llevando un registro particular del Token en el cual se está parado a través de la variable yylval de Parser.

Esto posibilitó la correcta identificación de terminales a través de las referencias tipo ‘$0’ que provee Yacc.

Por otra parte, y sintetizando un proceso con muchos shift/reduce y reduce/reduce en el camino, lo que se hizo fue dividir los bloques ejecutables y declarativos en sentencias y declaraciones respectivamente. A su vez, cada sentencia puede ser de control, una llamada a método, una asignación o una llamada a la función Print y una declaración puede ser de variable, clase o de método.

Lista de errores:

Los errores considerados por el analizador sintáctico se detallan claramente en la gramática ajunta. Debajo se detallan los más importantes:

Programa:

- Falta Bloque ejecutable (BEGIN END)
- Error en el Bloque ejecutable
- Error en el Bloque Declarativo

Bloque Declarativo:

- Falta ‘;’
- No se puede declarar clases dentro de otra
- Falta nombre de la Clase
- Falta lista de clases padre
- Falta nombre de la Clase
- Reconoce declaración de Clase

Declaración de Método:

- Reconoce declaracion de Metodo
- Parámetros erróneos
- Falta ')'
- Falta '('
- Falta BEGIN
- Reconoce declaracion de Metodo
- Falta ')'
- Falta '('
- Falta BEGIN

Bloque Ejecutable:

- Falta 'END' para cerrar el bloque de sentencias

- Falta ‘;’
- Error en la sentencia ejecutable

Sentencia ejecutable:

- Falta “;”

Sentencia PRINT:

- Falta cadena para imprimir
- Falta paréntesis de cierre en la sentencia de impresión
- Falta paréntesis de apertura en la sentencia de impresión
- CADENA faltante en la sentencia de impresión
- 'PRINT' faltante en la sentencia de impresión

Sentencia IF:

- Reconoce sentencia IF
- Falta condición
- Falta '('
- Falta ')'
- Falta condición
- Falta '('
- Falta ')'

Condición:

- Falta condición de comparación
- Mala condición de comparación

Llamada a Método:

- Reconoce llamada a Método
- Falta '('
- Falta ')'
- No se permite pasaje de parámetros

Asignación:

- Operador de asignación erróneo

Lista de no terminales:

programa: Reconoce si el programa fue bien escrito.

bloqueDeclarativo: Se utilizó este no terminal para identificar todas las sentencias declarativas que deben estar antes de las sentencias ejecutables. Entre

estas sentencias declarativas se encuentran todas las declaraciones de clases, métodos, variables y objetos.

listaDeclaraciones: conjunto de sentencias similar al bloqueDeclarativo con la excepción que este conjunto de declaraciones es utilizado en las declaraciones de Clase, sin permitir declarar clases dentro de la misma. Permite declarar atributos y métodos.

declaracionClase: No terminal utilizado para la declaración de clases y clases con herencia múltiple. (Tema asignado **(12)**)

declaracion: Utilizado en la declaración de variables. Contempla la declaración de una o muchas variables.

declaracionMetodo: Utilizado en la declaración de métodos. Permite la declaración de métodos con o sin sentencias ejecutables.

tipo: Puede ser *int* o *ulong. U*tilizado en la declaración de variables.

lista_de_ID: Pueden ser dos ID separados por una coma (ID , ID) o una lista de ID separados por comas.

bloqueEjecutable: Análogamente a bloqueDeclarativo, este no terminal define las posibles sentencias ejecutables que permite el programa. Permite la creación de bloques ejecutables de una o más sentencias ejecutables, siempre que, en caso de ser más de una, se encuentren definidas entre BEGIN y END.

sentenciasEjecutables: Una o más sentencias ejecutables. Fue necesario crear este nivel para poder diferenciar entre sentencias ejecutables que llevan punto y coma al final (asignación) y las que no (sentencias de control).

sentenciaEjecutable: Define los tipos de sentencias ejecutables que llevan punto y coma (**';'**) al final. (Asignaciones, sentencia Print, llamadas a métodos)

sentenciaControl: No terminal que define los tipos de sentencias de control (**IF**, Tema asignado **(6) - do until**)

condición: Define la sintaxis de las condiciones utilizadas por las sentencias de control. Permite todas las comparaciones (<, >, <>, <=, >=, ==) entre dos expresiones.

llamadaMetodo: No terminal que define la sintaxis de las llamadas a métodos. No permite pasaje de parámetros.

asignacion: Define las reglas para reconocer la sentencia ejecutable de asignación. Permite realizar la asignación a cualquier identificador ya sea declarado en el programa o en una clase del mismo.

expresión: Define las reglas de suma y resta como expresiones, además de identificar la conversión explicita TO_ULONG () (Tema asignado **(14)**)

termino: Define las reglas de división y multiplicación como términos.

constante: Se considera como constante a los Token CTE y también a los identificadores. Esto es así para permitir la suma, resta, división, multiplicación y conversión a ULong de variables.

Identificador: Puede ser un identificador simple (**ID**) o un identificador de un atributo de una clase (**ID.ID**).

Conclusiones:

Para la realización del primer trabajo práctico, el analizador léxico, surgieron varios desafíos de implementación que mediante sucesivas correcciones sobre las matrices y la implementación del getToken pudieron ser resueltos. Para la segunda parte del compilador, nos encontramos con ciertos asuntos con el manejo de las constantes negativas, esto se debió a que en la primera parte habíamos contemplado el signo menos en el autómata para que se lo considere parte de la constante. Para solucionar esto fue necesario realizar muchos ajustes y sobre todo, una implementación de un *check_rango* el cual fue implementado en el parser ya que debería ser ejecutado al reconocer la constante junto al token **‘-‘.**